Tree automata, approximations, and constraints for verification : Tree (Not quite) regular model-checking

Vincent Hugot

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

N° 6 6 6

THÈSE présentée par

VINCENT HUGOT

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Tree Automata, Approximations, and Constraints for Verification

Tree (Not-Quite) Regular Model-Checking

Soutenue publiquement le 27 Septembre 2013 devant le Jury composé de :

PHILIPPE SCHNOEBELEN	Rapporteur	Directeur de recherche, CNRS
JEAN-MARC TALBOT	Rapporteur	Professeur à l'Université d'Aix-Marseille
PIERRE-CYRILLE HÉAM	Co-Directeur	Professeur à l'Université de Franche-Comté
OLGA KOUCHNARENKO	Directeur de thèse	Professeur à l'Université de Franche-Comté
FLORENT JACQUEMARD	Examinateur	Chargé de recherche HDR, INRIA
JEAN-FRANÇOIS RASKIN	Examinateur	Professeur à l'Université libre de Bruxelles
SOPHIE TISON	Examinateur	Professeur à l'Université de Lille

Version of the document: **d361**,
dated *2013-12-30 05:08:42+01:00* ,
compiled on *December 30, 2013*.

Table of Contents

List of Figures

— Part I —

Motivations and Preliminaries

Chapter 1
Formal Tools for Verification

Contents

—Where we are reminded that bugs are bad, and that formal methods are good.

Ariane 5's 1996 maiden flight "501" enjoys the dubious distinction of being remembered as one of the most expensive fireworks displays in the history of mankind. Yet its most striking feature lies not in the spectacular character of the failure, but in how it came to pass. The cause was not a structural flaw of the rocket, but a software bug. The ruinous error lay in a single line of Ada code in the inertial navigation system, a fairly simple conversion from 64-bit to 16-bit that should have checked for overflows but did not. Misled by erroneous navigation data, the rocket veered hopelessly off course, and self-destructed. There may be imponderables in rocket design, but that was not one of them. The range check had actually been deliberately deactivated for this conversion, as a performance optimisation made under the belief that there was an ample margin of error. This may have been true for Ariane 4, from which the navigation system was copied directly, but the greater acceleration of Ariane 5 turned out to be beyond the scope of the 16-bit variable.

Not all individual software bugs cost a few hundred million to one billion dollars – as did flight 501 – but they are pervasive and the costs accrue over time. However, there is no intrinsic difference between (mostly) harmless, everyday bugs and catastrophic ones, as a quick look at some of the most publicised incidents shows. Similar to the Ariane 5 incident is the loss of the Mariner I space probe in 1962: an error in some rarely-used part of the navigation software of the Atlas-Agena rocket resulted in the unrecoverable failure of its guidance system. The Phobos I probe on the other hand was launched successfully in 1988, but a malformed command sent from earth forced the unexpected execution of a test routine that was supposed to be dead code. The year 1999 saw the loss of two probes to software errors: the Mars Climate Orbiter likely disintegrated in Mars's atmosphere because the ground-control computer was using imperial units while the probe itself used metric units; the Mars Polar Lander nearly made it to the ground, but invalid touchdown detection logic prompted it to cut thrusters 40 meters above the ground. After a decade of loyal services, the Mars Global Surveyor was lost in 2006 because of an error causing data to be written in the wrong memory address.

The Cost of Software Bugs

A study conducted by the NIST in 2002 concluded that software bugs cost the US economy about $59 billion each year, or about 0.6% of the GDP. A 2013 Cambridge University study estimated a global annual cost of $312 billion. . . for the constant debugging activity alone.

In the medical domain, the Therac-25 radiation therapy machine is infamous for having killed three patients and balefully irradiated at least three others between 1985 and 1987. Its predecessor, the Therac-20, used a mechanical safety interlock that prevented the high-powered electron beam from being used directly. The Therac-25 used a software interlock instead, which a race condition could disable if the machine's operator was fast enough. Another race condition, this time in the alarm routines of a XA/21 energy management system, escalated what was a minor power failure into the USA & Canada Northeast blackout of 2003, depriving 55 million people of electricity for up to two days. In 1991, a MIM-104 Patriot anti-ballistic battery correctly detected an incoming Al-Hussein missile, but after 100 hours in operation, cumulative rounding errors had caused its internal software clock to drift by one third of a second; using this erroneous time to predict the missile's trajectory yielded an error of about 600 meters. Unopposed, the missile hit its mark, killing 28 soldiers.

Total Recall

In 2010, Toyota recalled 133 000 Prius hybrids and 14 500 Lexus hybrids. In 2004–05, Mercedes recalled almost two million SL500 and E-Class vehicles. Both cases were warranted by faults in the braking software.

Range checks, race conditions, access to dead code, dimensional clashes, clock synchronisation problems… Despite their dramatic consequences, those are all perfectly mundane bugs of the same kinds that plague desktop computers on a daily basis, from word processors to card games. But when software controls rockets, missiles, or any sort of critical equipment, bugs are more than mere annoyances. Increasingly, sophisticated software replaces simpler mechanical systems and specialised circuits. Embedded systems are everywhere, from pocket watches to microwave ovens to phones to cars to planes to rockets. But regardless of what it is that the software controls, preventing, handling and fixing bugs is not rocket science, but computer science.

verification
formal methods

There are many approaches dedicated to the end-goal of reliable software; this thesis only concerns itself with the field of *verification* (or *formal methods*), whose aims can be broadly defined as proving that a given piece of software or hardware is correct with respect to a given specification. The rise of embedded systems not only makes such methods more necessary than ever, but also contributes to create a "target-rich" environment for the field. The high cost of formal methods is indeed offset by the much higher cost of bugs in embedded systems: even if a bug is caught in time and causes no damage, recalling and fixing entire lines of products is prohibitively expensive. Furthermore, embedded systems are often smaller and more specialised than modern desktop software, which makes them easier to specify and to check. Thus the current technological advances provide a strong impetus for software verification.

That field is quite vast, however; this chapter provides a very succinct and mostly informal overview of the techniques and traditions within which our work is inscribed.

1.1 Model-Checking: Simple, Symbolic & Bounded

Hoare logic

One of the first approaches to program verification is *Hoare logic*, introduced in [Hoare, 1969] and further refined by many other researchers, notably Floyd and Dijkstra. The basic idea is to enclose a program code C between two assertions p

and q expressed in standard mathematical logic, the first being called *precondition* and the second *postcondition*. The resulting triplet is often written $\{ p \}\ C\ \{ q \}$ and interpreted as the statement "if p holds before the execution of C, then q holds after its execution". For instance, it holds that $\{ \top \}\ x := y\ \{ x = y \}$. A set of axioms and rules of composition make it possible to cover entire programming languages, provided that their semantics are clearly defined. Thus the proof of correctness of a program becomes a mathematical theorem.

temporal logics

A related development is the use of *temporal logics* in software verification, proposed in [Pnueli, 1977]. Temporal logics were originally developed within and for the philosophy of human language, but they turned out to be well-suited to the task of specifying concurrent as well as sequential programs. Properties such as mutual exclusion, absence of deadlocks and absence of starvation are most conveniently expressed under those formalisms.

Nevertheless, whatever the kind of logic in use, in the absence of automatic theorem provers every proof of correctness had to be hand-crafted. This of course required of lot of time, and often a lot of mathematical ingenuity as well. Despite this, human error remained frequent, especially given the intrinsically repetitive and tedious character of many program proofs. Rightly sceptical about the scalability of manual proofs, Clarke & Emerson combined temporal logic with the state-exploration approach proposed – but heretofore largely ignored – by Gregor Bochmann and others. The result was the seminal paper [Clarke & Emerson, 1981] which, along with the similar but independently researched [Queille & Sifakis, 1982], is considered the foundation of *model-checking* as it is now understood. It is often the case that, when the time is ripe for a good idea, it is discovered independently and simultaneously by several people; this undoubtedly happened for model-checking. Although published in 1982, the work of Queille & Sifakis first appeared as a technical report version in 1981, and is similar in many respects to that of Clarke & Emerson. In all cases, the problem solved by model-checking is the following:

model-checking

Clarke, Emerson and Sifakis continued to refine model-checking over the years, and jointly received the Turing award in 2007 for their overall contributions to that domain.

▽ Definition 1.1: Model-Checking Problem

Let M be a finite *Kripke structure* – i.e. a finite state-transition graph. Let φ be a formula of temporal logic (i.e. the specification). Find all states s of M where φ holds true, i.e. such that $M, s \models \varphi$.

Kripke structure

Oftentimes, the real question is more simply to determine whether $M, s_0 \models \varphi$, where s_0 is singled out as an initial state of the system. Note that the name "model-checking" refers to determining whether M is a model – in the logical sense – of φ, and not to the dictionary meaning of "model" as an abstraction of the system under study. [Clarke & Emerson, 1981] presented a fixpoint characterisation of and model-checking algorithm for a new variety of branching-time temporal logic called *Computation Tree Logic* (*CTL*), first defined and linked to μ-calculus in [Emerson & Clarke, 1980]. The model-checking algorithm recursively broke down the formula φ into its sub-formulæ, and incrementally labelled each state of the system according to which subformulæ they satisfied or violated. For instance, $\neg\psi_1$ and $\psi_1 \wedge \psi_2$ being subformulæ of φ, if at pass k a state s was labelled by ψ_1

Computation Tree Logic
CTL

and ψ_2, then at pass $k+1$, the state s could and should additionally be labelled by $\psi_1 \wedge \psi_2$ and $\neg(\neg\psi_1)$.

This method, while limited to finite systems, presented great advantages over manual proofs, the most obvious being that it was entirely mechanical and required no user input whatsoever – besides the system M and its specification φ, obviously. As a non-negligible bonus, the method could be extended to allow the generation of counterexamples and witnesses, a functionality first implemented in 1984 by Michael C. Browne – then a student of Clarke – in his `MCB` verifier, and a staple of all model-checkers since then.

automata-theoretic model-checking

Linear Temporal Logic
LTL

Büchi automaton
FSA
ω-word
ω-language

It is notable that, in this approach, the system and the specification are handled very differently. In another viewpoint, often called *automata-theoretic model-checking* and spearheaded in [Aggarwal, Kurshan & Sabnani, 1983], both the system and the specification are represented by automata; this idea was applied to *Linear Temporal Logic* (*LTL*) [Pnueli, 1977] in [Vardi & Wolper, 1986]. In this framework, the model-checking problem becomes formulated in terms of language containment, and the verification algorithm is reduced to automata-theoretic constructions. First, the model M is transformed into a *Büchi automaton*, that is to say a kind of finite-state automaton (*FSA*) with an acceptance condition adapted so that they work on *ω-words* (infinite words), and thus accept an *ω-language*. In this step, the labels of the states of M become the new transition labels, and thus a word of this new automaton $\mathcal{B}_M$ corresponds to an execution of the system, which is either valid or invalid wrt. the LTL formula φ. Second, φ itself is converted into a Büchi automaton $\mathcal{B}_\varphi$, which accepts precisely the set of executions satisfying φ. Then the question of whether the system M is a model of the property φ becomes equivalent to the language containment $\mathcal{L}(\mathcal{B}_M) \subseteq \mathcal{L}(\mathcal{B}_\varphi)$. This is the method used by the `Spin` verifier.

In practice one actually prefers to solve the equivalent problem $\mathcal{L}(\mathcal{B}_M) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi}) = \varnothing$. This is a technical detail, however.

symbolic model-checking

ordered binary decision diagrams
OBDD

The greatest limitation of model-checking up to that point was what is known as the *state explosion problem*. The number of global states of a system can be gigantic, especially in the case of concurrent systems involving a great many different processes. The original `EMC` model-checker, which implemented an improved version of the CTL verification algorithm mentioned above, was used successfully to check real-world systems with up to 10^5 states, and further improvements pushed the limits to 10^8 states. This is a rather small number, many orders of magnitude below what can be expected of a highly concurrent system: in general the number of global states grows exponentially with that of simultaneously executing components. In late 1987, McMillan, then a graduate student of Clarke, set out to solve the problem by replacing the explicit representation of the state-transition graph by a symbolic one, giving rise to *symbolic model-checking* [Burch, Clarke, McMillan, Dill & Hwang, 1992]. To illustrate the gains of symbolic representations, consider for instance the explicit $\{1, 2, 3, \ldots, 999, 1000\}$ (but without the ellipsis), as opposed to the symbolic $\{n \mid n \geqslant 1 \wedge n \leqslant 1000\}$. More precisely, in symbolic model-checking, sets of states and transitions – and therefore the Kripke structure – are represented by propositional formulæ, or equivalently, boolean functions. This opens up a number of possibilities, the first of which is the use of *ordered binary decision diagrams* (*OBDD*) to represent the formulæ, and therefore, the system. OBDD often provide extremely compact representations which capture some of the underlying regularities of the system. The use of OBDD enables the application of CTL

model-checking to systems with up to 10^{20} states. In 2008, further refinements of the technique had pushed that number to 10^{120}.

To put this number of 10^{120} states in perspective, the estimated number of atoms in the entire observable universe is about 10^{80}. Non-trivial concurrent systems *still* routinely exceed such numbers of configurations, however. Put another way, it is less than the number of configurations of a system with 50 octets of memory.

Another successful symbolic technique is *bounded model-checking* [Biere, Cimatti, Clarke & Zhu, 1999], which takes advantage of the impressive advances in efficiency of *boolean satisfiability problem* solvers (*SAT solvers*) to find counter-examples of fixed length for LTL safety properties. There are many other related techniques, the discussion – or even enumeration – of which falls outside the scope of this short introduction. The interested reader is invited to consult [Clarke, 2008] for a more complete high-level historical survey of the field from 1980 to 2008.

bounded model-checking
boolean satisfiability problem
SAT solvers

1.2 Regular Model-Checking

The key assumption underlying model-checking as seen in the previous section is the finiteness of the state space. This assumption is challenged in many circumstances, especially by *parametrisation*. Consider some communication protocol involving an arbitrary number of simultaneously connected clients; that number is a *parameter* of the system. In the absence of an upper bound on that parameter, the set of possible configurations is infinite, and therefore correctness of the system cannot be checked by the techniques seen in the last section.

The solution lies again in symbolic representations of sets of states, the difference with the notion of symbolic model-checking presented in the previous section being that this time, the sets are infinite. In *regular model-checking* (*RMC*) [Kesten, Maler, Marcus, Pnueli & Shahar, 1997] states are represented by finite words over finite alphabets, sets of states are regular word languages, represented by finite-state automata, and the actions of the system are captured by a finite-state transducer (*FST*). Among other techniques, this kind of representation lends itself well to a form of verification called *reachability analysis*, which focuses on safety properties of the form "no bad state is ever reached", for some definition of "bad". A significant portion of this thesis is inscribed in a closely related context; therefore, to prepare for further discussions, this section presents and semi-formally justifies the key intuitions under the challenges facing this family of verification methods.

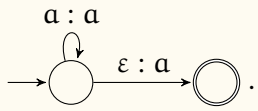
regular model-checking
RMC
FST
reachability analysis

The general idea of reachability analysis with regular model-checking is to start with: an initial language S_0, that is to say the set of possible initial states of the system, represented by a finite-state automaton; a set $\mathcal{B}$ of so-called "bad states", also given as an automaton; and a finite-state transducer $\mathcal{T}$ representing the system, that is to say a relation encoding the transitions from one state to another. Then an execution of the system looks like this:

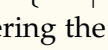
$$S_0 \xrightarrow{\mathcal{T}} S_1 \xrightarrow{\mathcal{T}} S_2 \xrightarrow{\mathcal{T}} S_3 \xrightarrow{\mathcal{T}} \cdots \xrightarrow{\mathcal{T}} S_n \xrightarrow{\mathcal{T}} \cdots ,$$

where $S_k = \mathcal{T}(S_{k-1}) = \mathcal{T}^k(S_0)$ is the regular set of states in which the system may be after exactly $k \geqslant 1$ transitions by $\mathcal{T}$. Therefore, the question of whether the system can ever reach a bad state is expressed as "$\exists k : S_k \cap \mathcal{B} \neq \varnothing$"?, or equivalently, "$\bigcup_{k=0}^{\infty} S_k \cap \mathcal{B} \neq \varnothing$"?, or also, using the standard notation for transitive and reflexive closure, "$\mathcal{T}^*(S_0) \cap \mathcal{B} \neq \varnothing$"?. It is clear that a purely iterative algorithm, computing and storing reached states transition after transition, and hoping to get

to a fixed point such that $S_{n+1} \subseteq \bigcup_{k=0}^{n} S_k$, has absolutely no guarantee of ever terminating. To take the simplest possible example, consider $S_0 = \{\varepsilon\}$ and the following transducer:

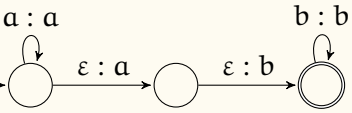
$\mathcal{T} =$ 

.

We have $\{\varepsilon\} \xrightarrow{\mathcal{T}} \{a\} \xrightarrow{\mathcal{T}} \{aa\} \xrightarrow{\mathcal{T}} \{aaa\} \xrightarrow{\mathcal{T}} \cdots$, and thus the state space is infinite, making even this trivial system unsuitable for the methods of the previous section. Fortunately, having symbolic representations does afford advantages: in a number of cases – such as this one – one can mechanically build the transducer $\mathcal{T}^*$, and then use it to build the automaton accepting $\mathcal{T}^*(S_0) = \bigcup_{k=0}^{\infty} S_k$. In the case of our very trivial example, $\mathcal{T}^*(S_0) = \left\{ a^k \mid k \geqslant 0 \right\}$, and the automaton is of course 

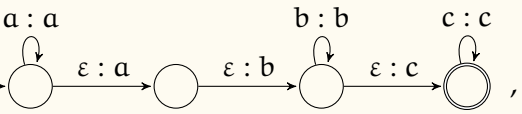
. From that point, answering the original question is two easy computations away: an FSA intersection and emptiness test.

A crucial point of regular model-checking, however, is that it is not always possible to compute $\mathcal{T}^*$: it is well-known that FST are closed by finite composition, so that $\mathcal{T}^k$ – and therefore $\bigcup_{n=0}^{k} S_n$ – can be built for arbitrary k, but are *not* closed by transitive and reflexive closure. To be convinced that $\mathcal{T}^*$ may not exist, let us consider a new transducer, a slight extension of the previous one:

$\mathcal{T}_2 =$ 

.

The transitions yield $\{\varepsilon\} \xrightarrow{\mathcal{T}_2} \{ab\} \xrightarrow{\mathcal{T}_2} \{aabb\} \xrightarrow{\mathcal{T}_2} \{aaabbb\} \xrightarrow{\mathcal{T}_2} \cdots$, thus $\mathcal{T}_2^*(\{\varepsilon\}) = \left\{ a^k b^k \mid k \geqslant 0 \right\}$. This is the archetype of non-regular languages, so $\mathcal{T}_2^*$ cannot be a FST. The resulting language is still context-free in that example, however even that property is easily disposed of with another transducer

$\mathcal{T}_3 =$ 

,

yielding $\mathcal{T}_3^*(\{\varepsilon\}) = \left\{ a^k b^k c^k \mid k \geqslant 0 \right\}$, the archetypal non-context-free language. While it is still context-sensitive, one could very well go even further down the Chomsky hierarchy, all the way to recursively enumerable languages, for instance by encoding the transitions of a Turing machine with a transducer. Those examples should however suffice to convey the notion that the general reachability analysis problem for infinite-state systems, even in the restricted context of linear languages and transitions – as defined above, is computationally difficult. It is actually undecidable [Apt & Kozen, 1986].

The literature follows three main approaches to deal with this fact. The first approach focuses on identifying special classes of systems (i.e. of transducers) that do preserve regularity through transitive and reflexive closure. Using such classes, reachability analysis goes smoothly; however the difficulty lies in expressing the system to verify in terms of such classes, which is of course not always possible. Indeed, the example of $\mathcal{T}_2$ and $\mathcal{T}_3$ shows that one does not need to look very far to find systems that fall squarely outside of those classes.

The second approach focuses on checking a greater but bounded number of steps by using *accelerations*. The gist of it is to break up the relation defined by $\mathcal{T}$ into smaller, disjoint chunks $\mathcal{T}_k$ such that $\mathcal{T} = \biguplus_{k=1}^{n} \mathcal{T}_k$, the chunks being individually more digestible than the whole in the sense that for as many k as possible, $\mathcal{T}_k^*$ is computable or, failing that, $\mathcal{T}_k^n$ is pre-computed for some large n. Then, by carefully choosing this partitioning and the order in which to use the $\mathcal{T}_k$, one may "skip steps" in the iterative algorithm described above, thus going much farther with the same computational resources and proportionally increasing the odds of detecting non-compliant traces, or even of reaching a fixpoint.

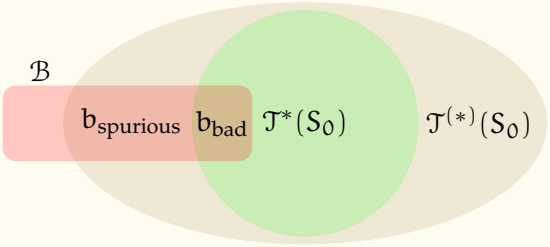
accelerations

Approximated Procedures

A *positive approximated procedure* is an algorithm – it always terminates – that answers "yes" or, if it fails to determine the answer, "maybe". A *negative approximated procedure* would answer "no" or "maybe".

The third approach, which is the one considered later in this thesis, is the use of over-approximations in order to obtain a *positive approximated procedure*. The idea is that, while the exact computation of $\mathcal{T}^*(S_0)$ may not be possible – because this set is not regular, and may even not be computable at all – it is possible to compute a regular set $\mathcal{T}^{(*)}(S_0)$ such that $\mathcal{T}^*(S_0) \subseteq \mathcal{T}^{(*)}(S_0)$. Then if it holds that $\mathcal{T}^{(*)}(S_0) \cap \mathcal{B} = \varnothing$, it follows that $\mathcal{T}^*(S_0) \cap \mathcal{B} = \varnothing$, and therefore the system is safe. On the other hand, if there exists a "bad" state $b \in \mathcal{T}^{(*)}(S_0) \cap \mathcal{B}$, then b may genuinely be reachable, in which case the system is not safe ($b_{\text{bad}} \in \mathcal{T}^*(S_0)$ on the figure), or it may simply be an artefact of the over-approximation ($b_{\text{spurious}} \in \mathcal{T}^{(*)}(S_0) \setminus \mathcal{T}^*(S_0)$ on the figure), signifying nothing.

positive approximated procedure
approximated procedures
negative approximated procedure



So, when the over-approximation intersects with the set $\mathcal{B}$ of bad states, there is no direct way to determine whether those are spurious or real counter-examples to the safety of the system. One technique to deal with that is to refine the abstraction underlying the over-approximation technique and try again. Thus the usefulness of the approximated procedure is directly dependent upon the quality of the approximation: the coarser the approximation, the less useful the method. Every set of real numbers can trivially be over-approximated by $\mathbb{R}$ – and very efficiently at that – but that is hardly helpful. On the other hand, the finer the approximation, the less likely it is to perform well. Finding suitable approximations is most often an empirical matter, informed by the exact question which needs to be answered, the nub of which suggests an abstraction keeping just the required information and discarding the rest. Considerable research work has gone into finding good approximations for the transitive and reflexive closure, and this method has been used successfully to check a large variety of infinite state systems. See [Abdulla, Jonsson, Nilsson & Saksena, 2004] for a survey of regular model-checking approaches.

There are many variants of those techniques: the choice of regular sets and finite-state transducers is absolutely not etched in stone. All that is required is that the class of languages involved support necessary properties of closure and decidability, as discussed on [Fisman & Pnueli, 2001], where context-free languages are used on the last step. As always, choosing appropriate representations is an exercise in compromise, where algorithmic complexity and decidability considerations must be

balanced against the expressive power required to encode the desired systems and properties. A widespread variant of regular-model-checking is its generalisation from regular word languages to regular tree languages [Kesten et al., 1997; Abdulla, Jonsson, Mahata & d'Orso, 2002], referred to as *tree regular model-checking* (*TRMC*). Most of this thesis is placed within the context of TRMC, thus the notion of tree is central to what follows.

tree regular model-checking
TRMC

Before saying more about trees in the next section, let us mention that there are other techniques dedicated to the study and verification of infinite-state systems. For instance, *well-structured transition systems* (*WSTS*) are transitions systems equipped with some well-quasi-ordering over the – infinite – set of states. This ordering is generally an abstraction of the structure of some particular class of transition systems such as Petri nets, lossy systems, process algebras, string rewriting systems and more, which are naturally well-structured. Furthermore, any transition system can be well-structured [Finkel & Schnoebelen, 2001, Thm. 11.1]. If certain properties are satisfied, many useful problems become decidable, such as termination, boundedness, eventuality, coverability, simulation of and by a finite automaton, etcetera [Finkel & Goubault-Larrecq, 2012, Sec. 3.1].

well-structured transition systems
WSTS

1.3 Tree Automata in Verification

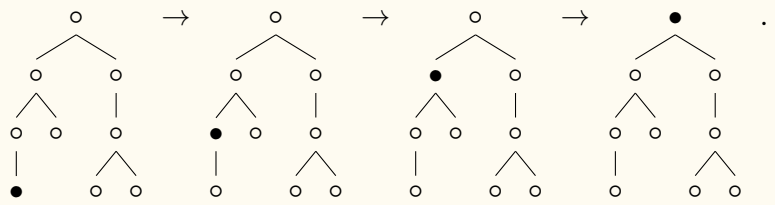
Terms, trees, tree languages and tree automata will be formally introduced in the next chapter. This section only provides a first intuition about what they are and why they are useful. We shall also briefly steer the discussion away from model-checking by pointing out other kinds of formal verification that may be carried out using a formal notion of trees. Let us proceed by example: below are three different representations of the same tree t:

$$t = f(a_1, g(a_2, a_3)) = \begin{array}{ccc} & f & \\ a_1 & & g \\ & & a_2 \quad a_3 \end{array} = \begin{array}{ccc} f & & a_1 \\ & g & \begin{array}{c} a_2 \\ a_3 \end{array} \end{array} .$$

The first representation is what is usually called a term; we equate terms and trees in this thesis, although a distinction exists, which will be made clearer in the next chapter. The second is the usual, top-down representation of trees in computer science; children are ordered left-to-right. The third is a slightly less usual horizontal representation, where the children are ordered top-to-bottom. Trees generalise the words recognised by finite-state automata: for instance the word abc can be represented as the tree $a(b(c(\bot)))$, using $\bot$ as an arbitrary leaf. As trees generalise words, so do tree languages generalise word languages, and finite tree automata, finite-state word automata.

The jump from words to trees affords extended expressive power, in that they allow simple representations of hierarchical structures. In the context of model-checking, this is most useful to encode systems which may be intrinsically simple, but which operate on a non-linear topology. Consider for instance a simple token-passing

protocol between processes, serving as something of a running example in this and the next chapter. Processes are organised according to a natural tree-like topology, with each process communicating directly with its parent and children, and only with them. The aim of our little protocol is to pass the token (of which there is and must always be exactly one) from whichever process holds it at the moment, to the most ancient process. Below is an example execution, with • representing the process with the token, and ∘ any process without the token:



It should be clear that such a protocol is extremely simple, yet regular model-checking, as defined on word languages, is incapable of expressing operations such as these. Going beyond regular word languages might solve the problem, but at an unreasonable cost: on top of representing the system, one would need to encode its topology in its states, with all the problems that supplementary complexity entails. Using trees instead of words is much more convenient, and, as we shall see, tree automata have all the nice closure properties required, while keeping generally manageable decision complexities. The very common occurrence of hierarchical structures in verification makes those techniques essential. Indeed, as best we could find in the literature, TRMC is not a generalisation that was thought of and developed after RMC, but instead it seems that they both originate from the same paper [Kesten et al., 1997], highlighting the natural need for tree-based methods.

The applications of tree automata to verification extend beyond the context of TRMC, though. An increasingly popular domain making extensive use of tree formalisms is that of so-called (semi-)structured documents, web databases etcetera, of which XML is one of the best and richest exemplars. Let us consider the following XML document:

```
<crew>
  <team name="Command">
    <member> Kirk </member> <member> Spock </member>
    <starship> NCC-1701[-A] </starship>
  </team>
  <team name="Science">
    <member> Spock </member> <member> McCoy </member>
    <starship> NCC-1701[-A] </starship>
  </team>
  <team name="Command">
    <member> Picard </member> <member> Riker </member>
    <starship> NCC-1701-D </starship>
  </team>
</crew>
```

At its core, the entire structure of any XML document is that of a tree. Each node has a "tag", or "label", and is classically referred to in XML parlance as an *element*. A node can have any number of children, the order of which is significant. Finally,

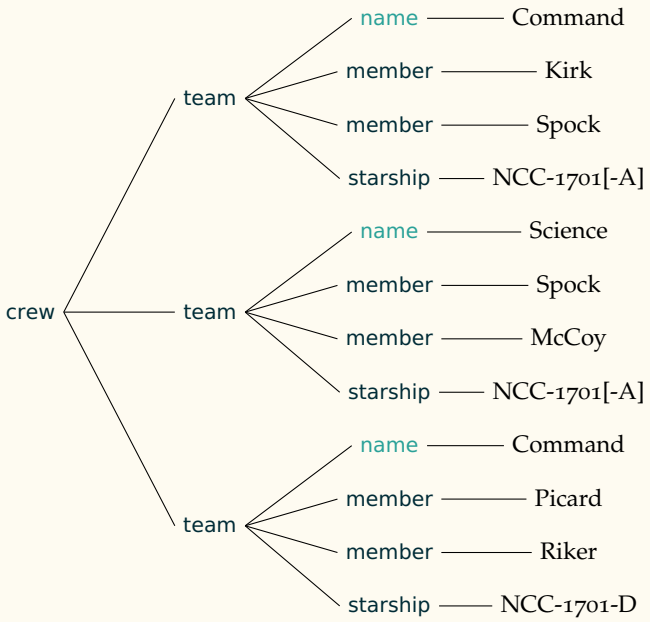


Figure 1.1: Tree representation of "Star Trek" XML document.

the leaves of the tree must be text strings. Figure 1.1 gives the tree representation of the example document. The alert reader will notice that we cheat a little by representing attributes simply as other elements; attribute processing is actually a little tricky because they are in reality unordered and non-duplicable. This thesis is not at all concerned with the details of XML processing, however, and minutiæ such as attributes and namespaces will be abstracted in the discussion. These little omissions notwithstanding, the very few constraints given above suffice a priori to describe XML documents. But the main selling point of XML is that, on the basis of this very simple structure, more precise constraints – called *schemas* – can be defined and enforced as needed. A schema defines which elements are expected, in what order, how many children they may have, and so forth. In that respect, a schema defines a data format, and XML itself is less of a data format and more of a meta-data format, or a format of data formats. For added recursive fun, schemas themselves may be written in XML.

schema

Taking a step back, what a schema defines is essentially a means of either accepting or rejecting trees. In other words, a schema defines a tree acceptor, and herein lies an important connexion between XML and tree automata theory: sufficiently expressive schemas can be encoded into finite tree automata. Thus, asking whether a given document belongs to some document format is equivalent to asking whether it satisfies some schema, which is in turn equivalent to testing whether the document's tree is a member of the language accepted by some tree automaton. With this correspondence in mind, a vast number of common questions and manipulations on structured documents are immediately expressed in terms of tree languages, and thereby in terms of closure constructions and decision problems on tree automata. By way of example, let $S_0, S_1, \ldots$ be schemas, represented as tree languages; then a database aggregator receiving data from sources conforming to either S_1 or S_2 will check the data against the schema $S_3 = S_1 \cup S_2$. Suppose now that the aggregator needs to consider only data that satisfies certain constraints, given by another schema S_0: then it actually needs to check conformance to $S_0 \cap S_3$.

One may then ask whether S_0 is actually a reasonable requirement, in the sense of it being compatible with the format of the input data: this amounts to the non-emptiness test $S_0 \cap S_3 \neq \emptyset$. If the intersection is empty, it means that no possible input data may conform to the aggregator's schema; which is probably a sign that it should be amended. Should the aggregator finally decide to amend the old S_0 schema into the new and improved S'_0, checking whether this new format is liable to invalidate old data corresponds to the containment check $S_0 \subseteq S'_0$. The list of applications goes on. While some of those problems look easy enough that one may cynically wonder why have a theory at all, others are quite difficult. For instance, complexity theory reveals containment checks, and consequently tests of safe evolution of a schema, type-checking etc, to be intrinsically expensive. Designing systems capable of answering those questions for real, highly voluminous data therefore requires careful abstract analysis. To conclude this short introduction, XML processing is one of those fields where theory and practice are very closely and visibly coupled. A reader interested in a very extensive survey of tree-automata theoretic foundations for XML processing is invited to consult [Hosoya, 2010].

Containment for finite tree automata is ExpTime-complete. Decision problems are discussed in the next chapter.

1.4 Outline and Contributions

This thesis revolves around the use of tree automata – in their various incarnations – not only for the verification of systems, but also for that of queries and other aspects of semi-structured documents or databases. The primary focus of our research is the verification of infinite-state systems. More precisely, the end-goal is to develop a fully functional verification chain based on a specific method of tree model-checking, at the confluence of tree regular model-checking, reachability analysis and rewriting logic.

The idea under this method was originally presented with hand-crafted proofs on examples in [Courbis, Héam & Kouchnarenko, 2009]. It combines aspects of tree regular model-checking and reachability analysis with the verification of properties expressed in temporal logic. Our goal is to generalise this process to a fragment of LTL, and to accomplish this we use and study tree automata with global equality constraints, a powerfully expressive model of tree automata originally developed in the context of logics for XML queries.

A secondary goal for this thesis is the improvement of algorithmic methods for tree-walking automata, a computational model with strong connections to semi-structured documents and, in particular, their navigational languages.

Part II forms the core of our contributions, as it deals with the model-checking method itself. It provides a positive answer to the question of whether the idea of [Courbis et al., 2009] can be generalised and extended into an automatic verification framework for a fragment of linear temporal logic. This is done by means of two distinct translation steps, for which we provide sets of automatic translation rules. The temporal specification is first converted into an intermediate form – a formula of propositional logic whose atoms are comparisons of rewrite languages, which we call a *rewrite proposition* – disregarding all properties of the system. Then, the intermediate form is turned into a (possibly) approximated procedure – the

general problem is undecidable in general – based on *tree automata with global equality constraints*; in this step, the specific properties of the system are taken into account, and affect the quality of the resulting approximated procedure. As a means of solving the rather difficult problem of the mechanical translation of a temporal specification into an equivalent rewrite proposition, we introduce the notion of *signatures*, which provide a linear model of some temporal formulæ. The part ends with a discussion of the fragment of linear temporal logic covered by our methods, in terms of the popularity – according to surveys – of the classes of properties which the automatic method may be able to handle. We also scour the literature for systems of interest modelled as term-rewriting systems, and examine their properties with respect to the second step. Some of the material in this part has been published in [Héam, Hugot & Kouchnarenko, 2012a], and most of the remainder is currently in submission [Héam, Hugot & Kouchnarenko, 2013].

The order of the authors in our publications is alphabetical.

The use of tree automata with global equality constraints (TAGE), superior in expressive power to the standard models of bottom-up tree automata, improves the precision of the approximations generated by the verification framework. However, this enhanced expressive power comes at the cost of high algorithmic complexities for many important decision problems. Furthermore, this is a relatively new class of automata and, although they have rich theoretical connections and multiple applications to XML and model-checking, there have been, to the best of our knowledge, no studies beyond our own geared towards achieving efficient decision procedures for them.

Part III focuses on TAGE and their decision problems; the aim is to obtain efficient algorithms for some common and useful decision problems, such as membership and emptiness tests, as well as to improve our general understanding of what it is that makes those problems complex. We provide a SAT encoding for membership testing (a NP-complete problem) and study the effect of bounds on the number of constraints, showing membership to be polynomial for any bound, and emptiness and finiteness to be at full complexity with as few as two constraints. The study on bounds has been published in [Héam, Hugot & Kouchnarenko, 2012c], and the SAT encoding in [Héam, Hugot & Kouchnarenko, 2010b]. In the same domain, we have also worked on providing heuristics and random generation schemes for emptiness testing (ExpTime-complete); while this work does not appear as part of this thesis, some of it is available as a research report [Héam, Hugot & Kouchnarenko, 2010a].

Part IV is linked to another kind of verification using tree automata, in relation to semi-structured documents. The focus in this part is a study of tree-walking automata (TWA), especially with respect to their conversion into bottom-up tree automata. Introducing the notion of *tree overloops* enables us to considerably reduce the size of the generated automata in the deterministic case, roughly from 2^{x^2} to $2^{x \log x}$. Furthermore, we present efficient algorithms for deciding membership, and a polynomial positive approximated procedure – generalisable to a class of increasingly precise such procedures – for emptiness testing, the decision of which is ExpTime-complete. This scheme is tested against uniformly random generated TWA, and turns out to be surprisingly accurate. This work has appeared in conference proceedings [Héam, Hugot & Kouchnarenko, 2011] and in an extended journal version [Héam, Hugot & Kouchnarenko, 2012b].

The next chapter introduces the technical notions and notations necessary for all parts of the thesis.

Chapter 2
Some Technical Preliminaries

Contents

—Where we are buried under definitions and examples.

Whereas the last chapter was geared towards a general and historical view of the field, this one provides a dryer formal exposition of the relevant technical concepts. Its contents are general prerequisites for all parts of this thesis – though each part puts emphasis on a different domain – and as such, should not be skipped. This being said, a reader already well-versed in the subject matters might be content merely to skim over it, in which case the index and marginal annotations should prove sufficient to find specific definitions, keeping in mind that less than usual mathematical symbols and notations do appear at the beginning of the index. In addition to what follows, the opening chapter of each part contains further preliminaries and historical as well as state-of-the art surveys relevant only to that part, making the present chapter necessary, yet not sufficient. The dependencies are summarised in Figure 2.1, where the styles of the nodes correspond to

Contribution Original Survey Technical Preliminaries .

A large proportion of the material presented here about trees and bottom-up tree automata is greatly inspired by [Comon, Dauchet, Gilleron, Löding, Jacquemard, Lugiez, Tison & Tommasi, 2008], and the reader is encouraged to refer to this book for a much deeper presentation of those topics. The remainder is either common scientific folklore or indebted to specific papers, in which case they are generally cited towards the end of the sections that make use of them. Other references include [Dershowitz & Jouannaud, 1990; Kirchner & Kirchner, 1996; Baader & Nipkow, 1998] for term-rewriting systems.

notational choices

For convenient reference, below is a small table of usual *notational choices*; they are used quite uniformly throughout the document, though Part II recycles some notations.

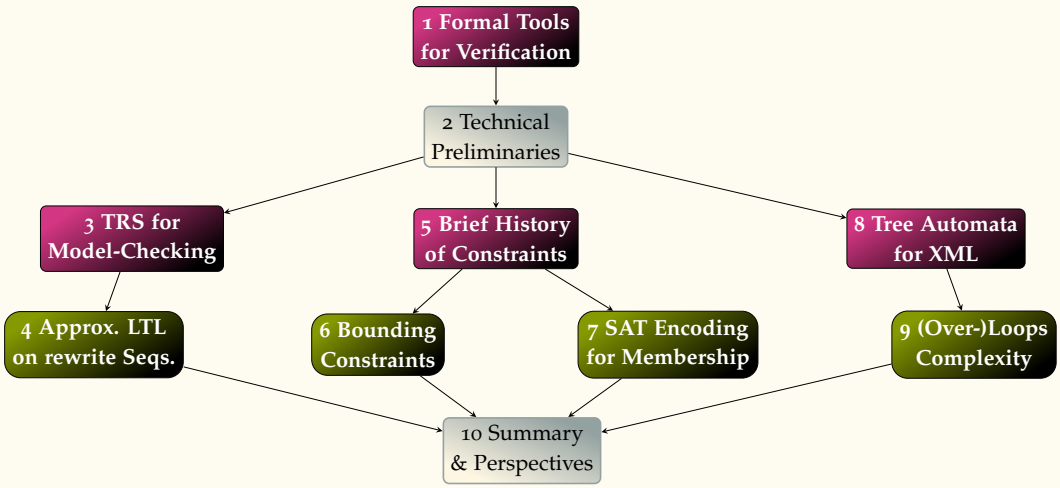


Figure 2.1: Reading dependencies between chapters.

$\mathcal{A}, \mathcal{B}, \mathcal{C}$	automata of all kinds	a, b, c	constant symbols
t, u, v	trees, subtrees	f, g	k-ary symbols, $k \geqslant 1$
p, q	automata states	σ	symbols, substitutions
w, v	words	ρ	run of an automaton
α, β	nodes, positions,	ε	empty word
σ, ρ	signatures *(Part II)*	λ	empty word *(Part II)*

2.1 Pervasive Notions and Notations

Sets, & Intervals. Inclusion is written $\subset$ if it is strict, and $\subseteq$ otherwise. The set $\mathbb{N}$ of the natural integers is extended in the usual way into $\overline{\mathbb{N}} = \mathbb{N} \cup \{+\infty\}$, with $+\infty > x$ for all $x \in \mathbb{N}$. For any $k \in \mathbb{N}$, we let $\mathbb{N}_k = [k, +\infty) \cap \mathbb{N}$ and $\overline{\mathbb{N}}_k = \mathbb{N}_k \cup \{+\infty\}$. For $n, m \in \mathbb{Z}$, the integer interval $[n, m] \cap \mathbb{Z}$ is written $[\![n, m]\!]$, with the convention that $[\![n, +\infty]\!] = [n, +\infty) \cap \mathbb{Z}$. The powerset of S is written $\wp(S)$. The *disjoint union* of two sets X and Y is written $X \uplus Y$, and is the same object as $X \cup Y$, with the added underlying assertion that $X \cap Y = \varnothing$, or that X and Y can trivially and safely be chosen disjoint.

$[\![i, j]\!]$: *integer interval*

$X \uplus Y$: *disjoint set union*

Relations & Functions. Let $R \subseteq S^2$ be a binary relation on a set S; we denote by R^+, R^* and $R^{\equiv}$ its transitive, reflexive-transitive, and equivalence closure (symmetric-reflexive-transitive), respectively, and we may write xRy for $(x, y) \in R$. Unless explicitly stated otherwise – e.g. page 169 – reflexive closures are taken on the domain $\mathrm{dom}(R) = \{ x \mid \exists y : xRy \text{ or } yRx \}$, even if R has been introduced as a relation on the larger set S. A *partial function* $f : D \to C$ from a domain D to a codomain C is a relation $f \in \wp(D \times C)$ with the functional property: $\forall x \in D; a, b \in C;\ [(x, a) \in f \wedge (x, b) \in f] \Rightarrow a = b$. The set of partial functions from D to C is written $D \nrightarrow C$. A *total function* f is a functional relation such that $\forall x \in D,\ \exists a \in C : (x, a) \in f$. The set of total functions is written $D \to C$ or C^D. The domain of a function $f : D \to C$ is defined differently from that of the underlying relation, as the largest subset $X \subseteq D$ such that the restriction $f|_X$ is total.

$R^{\equiv}$: *equivalence closure*

domain of a relation

partial function

total function

domain of a function

Function Application, Substitutions. Function application is most often denoted by $f(x)$, meaning the application of f on x. Occasionally, and in particular when there is a need to distinguish function application from the construction of terms, this is simply written $f\,x$, with a thin typographical space. The image of a subset $S \subseteq \operatorname{dom} f$ is written directly as the application of f on S, unless there is a risk of confusion. The postfix application notation common for substitutions in the literature – about term rewriting in particular – is not used in this document, with one exception: in relatively informal contexts, substitution is written in the commonplace notation $\varphi[v/X]$, meaning "v replaces X in the expression φ". In the usual notation of section 2.3[p27], this would become $\{X \mapsto v\}\varphi$. We let, for all $x \in \mathbb{R}$, $|x|_0 = \frac{1}{2}(x + |x|)$.

$f(x)$, $f\,x$: *function application*

$\{X \mapsto v\}\varphi$, $\varphi[v/X]$: *substitution*

$|x|_0$: *positive or zero*

Quotient Sets. Let $(\sim) \subseteq S^2$ be an equivalence relation over a set S; that is to say, $\sim$ is reflexive, symmetric and transitive. Given an element $x \in S$, the *equivalence class of* x with respect to $\sim$ is the set $[x]_\sim = \{y \in S \mid x \sim y\}$. The *quotient set* of S with respect to $\sim$ is the set $S/_\sim = \{[x]_\sim \mid x \in S\}$ of all $\sim$-classes.

$[x]_\sim$: *equiv. class of* x *wrt.* $\sim$

$S/\sim$: *quotient set of* S *by* $\sim$

Words, Kleene Closure. Let $\mathbb{A}$ be an alphabet, in other words, a set of symbols, or letters. Then the *Kleene Closure* over this alphabet is the set of finite words over $\mathbb{A}$. The empty word is written ε (most of the time), or λ (in Part II[p41], which has specific notational needs). The concatenation of two words $v, w \in \mathbb{A}$ is traditionally represented either implicitly by the juxtaposition vw, or explicitly as $v.w$. For the most part we shall favour the latter convention for arbitrary words, and the first for letters. The length of a word w is written $\#w$.

Kleene Closure

λ, ε: *empty word*

$\#w$: *length of word* w

2.2 Ranked Alphabets, Terms, and Trees

The previous chapter touched briefly upon the notions of term and tree. In this section we shall define both notions properly, see that they can be considered equivalent in the context which interests us, and then promptly forget the distinction between the two: they will be conflated in the rest of the document. Just as words are defined over a given alphabet, so are terms defined over a *ranked alphabet*. Let $\mathbb{A}$ be a finite alphabet, i.e. , a finite set of symbols, and $\mathsf{arity} : \mathbb{A} \to \mathbb{N}$ the *arity function*; intuitively, this function associates to a functional symbol $\sigma \in \mathbb{A}$ the number of arguments which it may take. The couple $(\mathbb{A}, \mathsf{arity})$ then forms a ranked alphabet. By abuse of notation the arity function will most often be kept implicit, and $\mathbb{A}$ will stand for the ranked alphabet. The arities will then be given together with the symbols using the shorthand $\sigma/_k$ for a symbol σ of arity k, and $\sigma_1, \ldots, \sigma_n/_k$ for a list of symbols $\sigma_1/_k, \ldots, \sigma_n/_k$. The set of all symbols of $\mathbb{A}$ whose arity is k is written $\mathbb{A}_k = \{\sigma \in \mathbb{A} \mid \mathsf{arity}(\sigma) = k\}$. A symbol of arity k is said to be k-ary, or nullary, unary, binary, and ternary in the cases where $k = 0, 1, 2$, and 3, respectively. Alternatively, one may speak of adicity instead of arity, and the symbols are then called k-adic, or medadic, monadic, dyadic, and triadic, for $k = 0 \ldots 3$. Nullary symbols are also referred to as *constant symbols*, or *constants*, and it is convenient to assume of any ranked alphabet $\mathbb{A}$ that it contains at least a constant, that is to say, $\mathbb{A}_0 \neq \varnothing$.

ranked alphabet

Varieties of Trees

What we refer to simply as *trees* should more precisely be called *finite ordered ranked trees*. Trees in general may be infinite, unranked, or both. The alphabet need not be finite either. This being said, most of those possibilities are outside the scope of this thesis. The same applies to terms. Unranked trees are presented section 8.2.1[p148].

arity function

The set of *terms* over the ranked alphabet $\mathbb{A}$ is written $\mathcal{T}(\mathbb{A})$, or simply $\mathcal{T}$ when

term

specifying the alphabet explicitly is not useful, and defined inductively as the smallest set such that

(1) $\mathbb{A}_0 \subseteq \mathcal{T}(\mathbb{A})$ and

(2) if $k \geqslant 1$, $\sigma \in \mathbb{A}_k$ and $t_1, \ldots, t_k \in \mathcal{T}(\mathbb{A})$, then $f(t_1, \ldots, t_k) \in \mathcal{T}(\mathbb{A})$.

In the context of terms, we sometimes find it convenient to write a constant a as $a()$, seeing it as having an empty list of children. This has the advantage, besides pure consistency of expression, to fuse base and inductive cases into one. For instance, the inductive definition given above can now be expressed in one statement:

$$\forall k,\ \sigma \in \mathbb{A}_k \wedge t_1, \ldots, t_k \in \mathcal{T}(\mathbb{A}) \implies \sigma(t_1, \ldots, t_k) \in \mathcal{T}(\mathbb{A})\ .$$

tree

Defining trees requires a bit more work. A *tree* t over a ranked alphabet $\mathbb{A}$ is a mapping from a finite non-empty set $S \subseteq \mathbb{N}^*$ into $\mathbb{A}$. Each element $\alpha \in S$ is called a node, and S itself is called the tree structure, or *set of positions*. There are structural constraints over S in order for the mapping to qualify as a tree. As a first requisite, it must be *prefix-closed*, which is to say that for any word $w \in S$, all prefixes of w are also in S. Specifically, whenever S contains a position $\alpha = k_1 k_2 \ldots k_{n-1} k_n$, then it must also contain $\beta = k_1 k_2 \ldots k_{n-1}$. By analogy with family trees, β is said to be the father, or parent, of α. Additionally, S must be closed with respect to little brothers, which means that it must also contain the positions $\beta.k$, for all $0 \leqslant k \leqslant k_n$. While this is the most classical definition, in practice it is often more convenient to index nodes from 1 instead of 0, in which case this definition is altered in the obvious way for $S \subseteq \mathbb{N}_1^*$; this is the convention we shall take in most of this thesis. As a last condition, arities must be observed, which means that for any position α, writing $a = \mathsf{arity}(t(\alpha))$, it must hold that $\alpha.a \in S$ but $\alpha.(a+1) \notin S$. In clearer words, combining the last two conditions, the number of children of a node α is equal to the arity of the symbol at position α.

set of positions

prefix-closed

It is then easy to see that, as announced, terms can be viewed as trees and vice-versa. Indeed, let us define the set of positions $\mathcal{P}(t)$ of any term $t \in \mathcal{T}(\mathbb{A})$; the definition is inductive on the structure of terms, as follows:

$\mathcal{P}(t)$: positions of a term

Note that "$f(t_1, \ldots, t_n)$" entails $f \in \mathbb{A}_n$, and that, with the $a = a()$ convention, **(2)** implies **(1)**.

(1) $\mathcal{P}(a) = \{\varepsilon\}$ if $a \in \mathbb{A}_0$, and

(2) $\mathcal{P}\big(f(t_1, \ldots, t_n)\big) = \{\varepsilon\} \cup \{\, k.\alpha_k \mid k \in [\![1, n]\!],\ \alpha_k \in \mathcal{P}(t_k)\,\}$, in general.

Then, if t is a term, a corresponding tree $\gamma(t)$ can be built as a mapping from $\mathcal{P}(t)$ to $\mathbb{A}$ simply by generalising the above construction so as to yield both the position and the corresponding symbol:

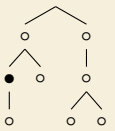
$$\gamma\big(\sigma(t_1, \ldots, t_n)\big) = \big\{(\varepsilon \mapsto \sigma)\big\} \cup \big\{\ (k.\alpha_k \mapsto \sigma_k) \ \big|\ k \in [\![1, n]\!],\ (\alpha_k \mapsto \sigma_k) \in \gamma(t_k)\ \big\}\ ,$$

and the construction can easily be inverted. This justifies the convention taken throughout this document to speak interchangeably of terms and trees, taking whichever viewpoint is the most convenient at the moment. For instance, if t is a term, the symbol at position α is written simply $t(\alpha)$ instead of $\gamma(t)(\alpha)$; in fact, we can forget the notation γ, for it will never have to be written explicitly again. Let us note then that $\mathcal{P}(t)$ and the domain $\mathrm{dom}(t)$ become two different notations for the same object, although in the context of terms and trees the former notation will systematically be preferred. Let us not forget, as was mentioned in the last chapter,

that trees are often represented graphically; representing the positions as well as the symbols, we have for instance, for $t = f(f(a,a), g(a,b))$,

$$t = \begin{array}{l} (\varepsilon \mapsto f) \\ \quad\diagdown\ (1 \mapsto f) \\ \quad\quad\diagdown\ (11 \mapsto a) \\ \quad\quad\diagup\ (12 \mapsto a) \\ \quad\diagup\ (2 \mapsto g) \\ \quad\quad\diagdown\ (21 \mapsto a) \\ \quad\quad\diagup\ (22 \mapsto b) \end{array} .$$

Note: the trees in the previous chapters, such as the one below, were given by abuse of notation: there is a version of • and ∘ for each arity with which they are used.



The existence of this tree implies that $a \in \mathbb{A}_0$ and $f, g \in \mathbb{A}_2$; arities being fixed, it is not stricto sensu possible to have, say, $f(f(a,a))$, because then f would be both unary and binary. Such an expression should be interpreted as $f(f'(a,a))$, with $a/_0$, $f/_1$ and $f'/_2$. To illustrate the functional view of trees, we have on this example $t(\varepsilon) = t(1) = f$ and $t(22) = b$, and the positions are $\mathrm{dom}(t) = \mathcal{P}(t) = \{\varepsilon, 1, 11, 12, 2, 21, 22\}$.

There remains to introduce a bit of vocabulary, and some common operations on trees. The empty position ε, which appears in the structure of all trees by definition, is called the *root* of the tree; nodes that have children are called internal nodes, and node that do not are called leaves. The terminology of computer-science being botanically backwards, "up" and "top" refer to the root, while "down" and "bottom" refer to the leaves. Given a tree t, the *parent function* $\mathsf{parent}(\cdot) : \mathcal{P}(t) \setminus \{\varepsilon\} \to \mathcal{P}(t)$ maps any child node $\alpha.k$ to its father α. The *height* of a tree is written $|t|$ and defined as $|t| = 1 + \max_{\alpha \in \mathcal{P}(t)} \#\alpha$, while its *size* is denoted by $\|t\|$ and defined by $\|t\| = |\mathcal{P}(t)|$. Positions are equipped with a partial order $\trianglelefteq$, such that $\alpha \trianglelefteq \beta$ if and only if β is a prefix of α. When this is the case, we say that α is under β, and β is an ancestor of α. Two positions α and β are *incomparable*, written $\alpha \curlywedge \beta$, if neither $\alpha \trianglelefteq \beta$ nor $\beta \trianglelefteq \alpha$. One can extract a *subterm* (or *subtree*) from a given term: letting $t \in \mathcal{T}(\mathbb{A})$ and $\alpha \in \mathcal{P}(t)$, the subterm of t at position α is denoted by $t|_\alpha$, and defined as follows:

root

parent$(\cdot)$: *parent function*

$|t|$: *height of tree* t

$\|t\|$: *size of tree* t

$\alpha \trianglelefteq \beta$: α *under* β

$\alpha \curlywedge \beta$: *incomparable positions*

$t|_\alpha$: *subtree of* t *under* α

(1) $\mathcal{P}(t|_\alpha) = \{\beta \mid \alpha.\beta \in \mathcal{P}(t)\}$

(2) for any $\beta \in \mathcal{P}(t|_\alpha)$, $t|_\alpha(\beta) = t(\alpha.\beta)$.

By extension of the ordering of positions, terms are submitted to a partial order, also denoted by $\trianglelefteq$, such that $u \trianglelefteq t$ if and only if u is a subterm of t, that is to say iff there exists $\alpha \in \mathcal{P}(t)$ such that $u = t|_\alpha$. This order is compatible with the ordering on positions, in the sense that for all $\alpha, \beta \in \mathcal{P}(t)$, we have $\alpha \trianglelefteq \beta \Rightarrow t|_\alpha \trianglelefteq t|_\beta$. The induced strict orders are additionally defined in the usual way, i.e. $x \triangleleft y$ iff $x \trianglelefteq y$ and $x \neq y$, regardless of whether x and y are both terms or both positions. It must be kept in mind that, taking the functional point of view on trees, a subtree is not simply a functional restriction of the original tree mapping. To illustrate this, let us go back to the example above; we have

$\alpha \triangleleft \beta$: α *strictly under* β

$$t|_2 = \begin{array}{l} (\varepsilon \mapsto g) \\ \quad\diagdown\ (1 \mapsto a) \\ \quad\diagup\ (2 \mapsto b) \end{array} \quad \text{and} \quad t|_2 \neq \begin{array}{l} (1 \mapsto g) \\ \quad\diagdown\ (11 \mapsto a) \\ \quad\diagup\ (12 \mapsto b) \end{array} .$$

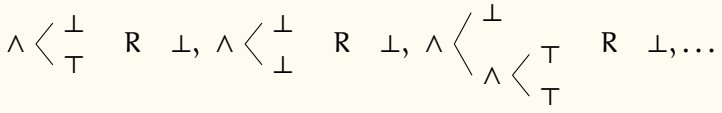
The second object, although represented as a tree, is not actually a tree according to our definitions.

2.3 Term Rewriting Systems

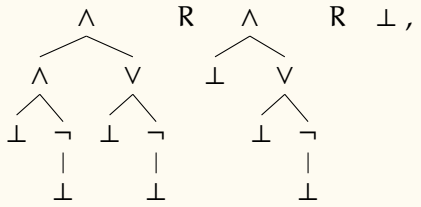
Once data is arranged into trees, it is natural to rearrange it according to certain sets of properties, expressed as (bidirectional) equations or (unidirectional) rules. This is the gist of any of the day-to-day algebraic manipulations on arithmetic and logical formulæ. Consider for instance the associative, commutative, and annihilation properties of logical conjunction:

$$\forall x, y, z; \quad x \wedge (y \wedge z) = (x \wedge y) \wedge z, \quad x \wedge y = y \wedge x, \quad \bot \wedge x = \bot\,.$$

It is immediate that all formulæ of propositional logic can be represented as terms, as can any mathematical or logical expression: it suffices for an operator to be translated into a symbol of corresponding arity. Discounting variables for now, expressions of propositional logic with the traditional operators are therefore terms over the alphabet $\mathbb{A} = \{\wedge, \vee/_2, \neg/_1, \top, \bot/_0\}$. Under that light, the properties above become transformations on trees, or equivalently, relations on trees. Expressed in those terms, the annihilation property $\bot \wedge x = \bot$ may be seen as the symmetric relation $R \subseteq \mathcal{T}(\mathbb{A})^2$, such that



In particular, two successive applications of this relation enable us to write statements such as



which accounts for the lazy evaluation of a specific formula. This section introduces *term rewriting systems* (*TRS*), that provide a uniform framework in which such relations are expressed and studied, and constitute a general, Turing-complete model of computation. The starting point is the combination of the notions of equations on terms (such as the annihilation property), and of algebraic manipulation. While an equation is bidirectional (by symmetry of equality), any isolated step of a computation actually manipulates symbols using only one direction of the equality. In both steps of the example above, annihilation is used in the direction "$\bot \wedge x \to \bot$", which, once encoded as trees, reads "$\wedge(\bot, x) \to \bot$". This is called a *rewrite rule*, and there remains to assign a precise meaning to the notation. It is clear that x plays the role of a free variable standing for any possible subterm, as in the second step of the example, where x stands for $\vee(\bot, \neg(\bot))$. If such a variable appears more than once, as in the commutative rule $\wedge(x, y) \to \wedge(y, x)$, then it should stand for identical subterms wherever it appears. Furthermore, the first step of the example illustrates the fact that such rules do not only operate on the entire tree, but on any suitable subtree as well: indeed, it is the subtree at position 1 that is changed in that step, the rest of the tree – the *context* – remaining untouched. Replacing a

term rewriting systems
TRS

subterm by another is a common operation, with its own notation: let t and u be terms and $\beta \in \mathcal{P}(t)$ some position, the result of the *replacement* by u of the subterm of t at position β is the tree $t[u]_\beta$ defined by

$t[u]_\alpha$: *subtree replacement*

(1) $t[u]_\varepsilon = u$,

(2) $f(u_1, \ldots, u_n)[u]_{k.\alpha} = f(u_1, \ldots, u_{k-1}, u_k[u]_\alpha, u_{k+1}, \ldots, u_n)$.

There remains to clarify the role of variables in rules such as $\wedge(\bot, x) \to \bot$. Note that, according to the definition of terms seen in the previous section, $\wedge(\bot, x)$ is not a term of $\mathcal{T}(\mathbb{A})$, since $x \notin \mathbb{A}$. The set $\mathcal{T}(\mathbb{A})$ as defined is more precisely known as the set of *ground terms* over $\mathbb{A}$, to indicate that it does not use any variables, although we rarely need to make that distinction explicit in this thesis. Variables are simply special constant symbols from a set $\mathbb{X}$, chosen disjoint from $\mathbb{A}$, and terms with variables are thus ground terms of $\mathcal{T}(\mathbb{A} \uplus \mathbb{X})$. Despite being otherwise ordinary nullary symbols, variables do hold a special status in several important operations on trees which we have yet to define; in order to make that status clear, terms over the alphabet $\mathbb{A}$ with variables from $\mathbb{X}$ will be written $\mathcal{T}(\mathbb{A}, \mathbb{X})$. Furthermore, it is convenient to consider the set of variables $\mathbb{X}$ to be countably infinite so as never to "run out". The set of variables appearing in a term $t \in \mathcal{T}(\mathbb{A}, \mathbb{X})$ is written $\mathcal{V}(t)$, and defined as

ground terms

$\mathcal{V}(t)$: *variables of a term*

$$\mathcal{V}(x) = \{x\} \text{ if } x \in \mathbb{X} \quad \text{and} \quad \mathcal{V}(f(t_1, \ldots, t_n)) = \bigcup_{k=1}^{n} \mathcal{V}(t_k) \text{ if } f \in \mathbb{A}_n \ .$$

A term t is *linear* if each variable of $\mathcal{V}(t)$ occurs only once in t. The prime purpose of a variable is of course to be replaced by a term, an operation called *substitution*. Strictly speaking, a substitution σ is a mapping from $\mathbb{X}$ to $\mathcal{T}(\mathbb{A}, \mathbb{X})$ such that all but finitely many variables are invariant; in other words, $\mathrm{Card}(\{\, x \in \mathbb{X} \mid \sigma x \neq x \,\}) \in \mathbb{N}$. This set $\{\, x \in \mathbb{X} \mid \sigma x \neq x \,\}$ of variables affected by the substitution σ is called its *domain* $\mathrm{dom}\,\sigma$. As a shorthand, a substitution σ can be written as the set $\{\, x \mapsto \sigma x \mid x \neq \sigma(x) \,\}$. When the co-domain of σ is the set of ground terms $\mathcal{T}(\mathbb{A})$, it is called a ground substitution. Substitutions are transparently extended to endomorphisms on $\mathcal{T}(\mathbb{A}, \mathbb{X})$, as follows:

linear term

substitution

$$\sigma f(t_1, \ldots, t_n) = f(\sigma t_1, \ldots, \sigma t_n), \qquad \forall f \in \mathbb{A}_n,\ t_1, \ldots, t_n \in \mathcal{T}(\mathbb{A}, \mathbb{X}) \ .$$

The set of substitutions on $\mathcal{T}(\mathbb{A}, \mathbb{X})$ is written $\mathcal{S}(\mathbb{A}, \mathbb{X})$, and that of ground substitutions on $\mathcal{T}(\mathbb{A})$ is written $\mathcal{S}(\mathbb{A})$. With this, we can at last define rewrite rules formally: a *rewrite rule* is a couple $(l, r) \in \mathcal{T}(\mathbb{A}, \mathbb{X})^2$ such that $\mathcal{V}(l) \supseteq \mathcal{V}(r)$ and $l \notin \mathbb{X}$, which is traditionally written $l \to r$. Following this notation, we call l the left-hand side of the rule, and r the right-hand side. If l is linear, then the rule is called *left-linear*, and if r is linear, it is called *right-linear*. A rule that is both left- and right-linear is called *linear*. A rewrite system $\mathcal{R}$ is a set of rewrite rules, and determines a corresponding rewrite relation $\to_\mathcal{R}$, written $\to$ when there is no ambiguity as to which rewrite system is involved. The *rewrite relation* is a binary relation between ground terms, and should not be confused with the "$\to$" of rewrite rules, although it shares its notation with them; it is determined as follows:

rewrite rule

The $l \notin \mathbb{X}$ condition can be lifted, but it is often taken in some contexts, such as completion algorithms; cf. 3.2.2[p46].

linearity of rules

rewrite relation

$$t \to_\mathcal{R} s \iff \exists \alpha \in \mathcal{P}(t), (l \to r) \in \mathcal{R}, \sigma \in \mathcal{S}(\mathbb{A}) :\ t|_\alpha = \sigma l \text{ and } s = t[\sigma r]_\alpha \ .$$

In clear, a term t is *rewritten into* a term s by $\mathcal{R}$ if there is some subterm of t that matches the left-hand side of some rule in $\mathcal{R}$, and the result s is obtained by

replacing that subterm within t by the right-hand side of the rule. The variables are replaced by the corresponding subterms in the match of the left-hand side. To summarise this with an arguably pithier, more memorable formula,

$$\forall t \in \mathcal{T}(\mathbb{A}), (l \to r) \in \mathcal{R}, \sigma \in \mathcal{S}(\mathbb{A}), \alpha \in \mathcal{P}(t) : \quad t[\sigma\, l]_\alpha \to_{\mathcal{R}} t[\sigma\, r]_\alpha \,.$$

When a term $t \in \mathcal{T}(\mathbb{A})$ cannot be rewritten, that is, when there does not exist any $s \in \mathcal{T}(\mathbb{A})$ such that $t \to_{\mathcal{R}} s$, it is called *irreducible*, or in *normal form* with respect to $\mathcal{R}$. The set of terms obtained through one step of rewriting of the ground language $\ell \subseteq \mathcal{T}(\mathbb{A})$ by $\mathcal{R}$, is written

normal form

$$\mathcal{R}(\ell) = \{\, s \in \mathcal{T}(\mathbb{A}) \mid \exists t \in \ell : t \to_{\mathcal{R}} s \,\},$$

and a symmetric notation exists in the reverse direction:

$$\mathcal{R}^{-1}(\ell) = \{\, t \in \mathcal{T}(\mathbb{A}) \mid \exists s \in \ell : t \to_{\mathcal{R}} s \,\}.$$

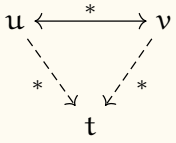
The sets of $\mathcal{R}$-descendants $\mathcal{R}^*(\ell)$ and $\mathcal{R}$-ancestors $\mathcal{R}^{-1*}(\ell)$ (resp. $\mathcal{R}^+(\ell)$ and $\mathcal{R}^{-1+}(\ell)$) are defined in the same way, using the reflexive and transitive closure $\to^*_{\mathcal{R}}$ (resp. the transitive closure $\to^+_{\mathcal{R}}$) instead of $\to_{\mathcal{R}}$. A TRS $\mathcal{R}$ is called *terminating*, *strongly normalising*, or *noetherian*, if there is no infinite sequence

terminating rewrite system

$$t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_n \to_{\mathcal{R}} \cdots .$$

A TRS $\mathcal{R}$ satisfies the *Church-Rosser property* if the following diagram holds:

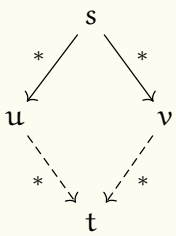
Church-Rosser property



$$\equiv \forall u, v;\ u \to^{\equiv}_{\mathcal{R}} v \Rightarrow \exists t : u \to^*_{\mathcal{R}} t \wedge v \to^*_{\mathcal{R}} t \,.$$

This is known to be equivalent to the *confluence property*:

confluence property



$$\equiv \forall s, u, v;\ s \to^*_{\mathcal{R}} u \wedge s \to^*_{\mathcal{R}} v \Rightarrow \exists t : u \to^*_{\mathcal{R}} t \wedge v \to^*_{\mathcal{R}} t \,. \quad (2.1)$$

Less general modes of rewriting can be defined as restricted TRS; for instance *semi-Thue systems*, or *word rewriting systems*, can be defined as rewrite systems on a unary alphabet $\mathbb{A}$ (i.e. $\mathbb{A} = \mathbb{A}_1$) such that all rules $l \to r$ satisfy $l, r \in \mathcal{T}(\mathbb{A}_1, \{x\})$. This corresponds of course to the unary encoding of words into terms glimpsed in the previous chapter – to be seen again in the next section – and yields rules of the form $a_1(\cdots a_n(x)\cdots) \to b_1(\cdots b_m(x)\cdots)$, which are written more simply as $a_1 \ldots a_n \to b_1 \ldots b_m$.

word rewriting systems

In this thesis, and most especially in Part II, TRS are used to encode the transitions of systems of interest, in preference to other commonly-used formalisms such as tree transducers. Recall for instance the simple token-passing protocol evoked at

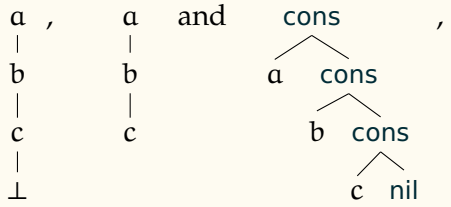
the end of the previous chapter; on binary trees, its transitions are expressed by the following rewrite system on the alphabet $\{\bullet, \circ/_2, \bullet, \circ/_0\}$:

$$\begin{aligned}
\circ(\bullet, \circ) &\to \bullet(\circ, \circ), & \circ(\bullet(x,y), \circ(x',y')) &\to \bullet(\circ(x,y), \circ(x',y')),\\
\circ(\circ, \bullet) &\to \bullet(\circ, \circ), & \circ(\circ(x,y), \bullet(x',y')) &\to \bullet(\circ(x,y), \circ(x',y')),\\
\circ(\bullet(x,y), \circ) &\to \bullet(\circ(x,y), \circ), & \circ(\bullet, \circ(x,y)) &\to \bullet(\circ, \circ(x,y)),\\
\circ(\circ(x,y), \bullet) &\to \bullet(\circ(x,y), \circ), & \circ(\circ, \bullet(x,y)) &\to \bullet(\circ, \circ(x,y)) \,.
\end{aligned}$$

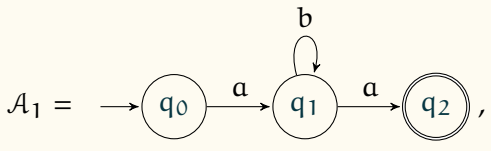
The domain of term rewriting is extremely rich and fundamental; it shares a significant part of its vocabulary, results and history with λ-calculus. The reader interested in surveys of rewriting theory is invited to consult the books [Dershowitz & Jouannaud, 1990; Kirchner & Kirchner, 1996; Baader & Nipkow, 1998].

2.4 Bottom-Up Tree Automata

As was mentioned in the previous chapter, trees generalise words; we gave the example of the word abc, and suggested $a(b(c(\bot)))$ as a possible tree encoding of it. In the newly acquired vocabulary of the previous section, it is now understood as a tree over the ranked alphabet $\{a, b, c/_1, \bot/_0\}$. There are endless varieties of other possible encodings, of course, from the asymmetric $a(b(c))$ to the classic LISP-style list encoding. To recapitulate,



respectively over $\{a, b, c/_1, \bot/_0\}$, $\{a, b/_1, c/_0\}$ and $\{a, b, c/_0, \mathsf{cons}/_2, \mathsf{nil}/_0\}$, are all perfectly valid tree encodings of the word abc. For the purposes of this discussion, let us choose the first style, and consider the word language $L = \{ ab^k a \mid k \in \mathbb{N} \} = \{aa, aba, abba, abbba, \dots\}$. This is a regular language, accepted by the finite-state automaton



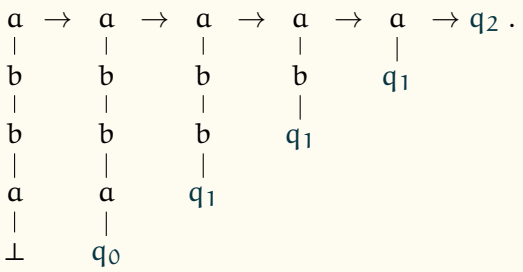
defined over the alphabet $\mathbb{A} = \{a, b, c\}$. The execution of $\mathcal{A}_1$ over a word w can be represented as a sequence of words over $\mathbb{A} \uplus Q$, where $Q = \{q_0, q_1, q_2\}$ is the set of states of $\mathcal{A}_1$. To do so, let us translate every transition (p, σ, q) of $\mathcal{A}_1$ into a rewrite rule $p\sigma \to q$. Then it suffices to start with $q_0 w$, and rewrite until a normal form is reached. With each transition, the first character of the word is "consumed" to change the state, until nothing is left but the state one ends up in. For instance, for the word $abba$, this yields

$$q_0 abba \to q_1 bba \to q_1 ba \to q_1 a \to q_2 \,.$$

A word is accepted if and only if it can be rewritten into a final state, which is clearly the case for abba. Now, we want to use the same principle to recognise the tree encoding of a word: the idea is to consume the (linear) tree to switch from state to state, thanks to rewrite rules. One could attempt to do so either from the root to the leaves (i.e. top-down) or from the leaves to the root (i.e. bottom-up). Either way works equally well, but the latter will prove slightly more convenient for us, and so it is what we shall use. The downside is that it requires the word to be encoded from the bottom up as well; quite fortuitously, the language L happens to be palindromic, which frees us from having to worry about such details in these examples. The adaptation from left-to-right word consumption to bottom-up tree consumption is then straightforward: a transition (p, σ, q) of the automaton becomes the ground rewrite rule $\sigma(p) \to q$, and the rule $\bot \to q_0$ must be added to prepare the initial state. This yields

$$a(b(b(a(\bot)))) \to a(b(b(a(q_0)))) \to a(b(b(q_1))) \to a(b(q_1)) \to a(q_1) \to q_2 \,,$$

or, in vertical tree representation:



At this point, there are two kinds of rules in play: nullary rules, of the form $\sigma \to q$, and unary rules, of the form $\sigma(p) \to q$. It seems natural to wonder what one could gain from extending such a system in the obvious way by supporting rules of the form $\sigma(q_1, \ldots, q_n) \to q$, acting on symbols of arbitrary arity. As it turns out, that query instantaneously leads to the definition of the most common variety of tree automata, which we are now going to state formally and which will be used throughout this document: non-deterministic *bottom-up tree automata* (*BUTA*), also called more generally (non-deterministic) finite tree automata (*NFTA*, *FTA*). Whenever we laconically write "tree automata" (*TA*) or even simply "automata" in the context of trees, this is what is meant.

bottom-up tree automata
BUTA

▽ Definition 2.1: Bottom-Up Tree Automaton

A bottom-up tree automaton $\mathcal{A}$ is a tuple $\langle \mathbb{A}, Q, F, \Delta \rangle$, where

- $\mathbb{A}$ is a finite ranked alphabet,
- Q is a finite set of *states*,
- F is the set of *final states*,
- Δ is the set of *transition rules*.

States are fresh nullary symbols, and final states are taken from a subset of states; in short, $Q \cap \mathbb{A} = \varnothing$ and $F \subseteq Q$. The transitions of Δ form a ground rewrite system on $\mathcal{T}(\mathbb{A} \uplus Q)$, where each rule is *normalised*, that is to say of the form

$$\sigma(q_1, \ldots, q_n) \to q, \quad \text{with } \sigma \in \mathbb{A}_n, q_1, \ldots, q_n, q \in Q \,.$$

The rewriting action of the system Δ is naturally written $\to_\Delta$, but can also be written $\to_{\mathcal{A}}$ if Δ has not been named explicitly but $\mathcal{A}$ has. Or, using the notations from section 5.4[p115], $\to_{\mathcal{A}:\Delta}$ is also an option.

The set $\mathcal{T}(\mathbb{A} \uplus Q)$ is called the set of configurations of the automaton, and captures the successive degrees of rewriting through which the input term passes under the action of the rewrite rules of Δ. One notices that, unlike finite-state automata, BUTA have no initial states. Recalling the $abba$ example just above, the first rewriting operation introduced the initial state q_0 of the original FSA into the tree. This is how BUTA behave in general: the leaves are rewritten first, and then the other rules can spring in action; rules of the form $a/_0 \to q_i$ replace initial states in a sense, and are sometimes called initial rules. The end-goal of a tree automaton is to rewrite the input term into a final state $q_f \in F$, if that is at all possible, which prompts the following definition for the accepted language $\mathcal{L}(\mathcal{A})$ of a tree automaton $\mathcal{A}$:

$\mathcal{L}(\mathcal{A})$: accepted language

$$\mathcal{L}(\mathcal{A}) = \{\, t \in \mathcal{T}(\mathbb{A}) \mid \exists q_f \in F : t \to_\Delta^* q_f \,\}\,.$$

Occasionally, there will be a need to focus on terms that rewrite into (or *evaluate* into, are *accepted* into, are *recognised* into,...) some specific state q. In those cases, we shall speak of the q-language $\mathcal{L}^q(\mathcal{A})$ of $\mathcal{A}$, and so we have

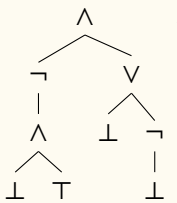
$\mathcal{L}^q(\mathcal{A})$: q-accepted language

$$\mathcal{L}^q(\mathcal{A}) = \{\, t \in \mathcal{T}(\mathbb{A}) \mid t \to_\Delta^* q \,\} \quad \text{and} \quad \mathcal{L}(\mathcal{A}) = \bigcup_{q_f \in F} \mathcal{L}^{q_f}(\mathcal{A})\,.$$

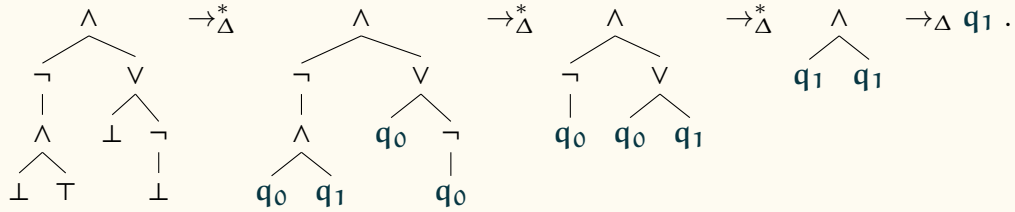
To illustrate that, let us take the very classical example of an automaton $\mathcal{A}$ accepting the tree representation of true variable-free propositional formulæ. We take the alphabet $\mathbb{A} = \{\, \wedge, \vee/_2, \neg/_1, \top, \bot/_0 \,\}$, states $Q = \{\, q_0, q_1 \,\}$, $F = \{q_1\}$, and the transitions

$$\Delta = \left\{ \begin{array}{l} \top \to q_1, \quad \bot \to q_0, \quad \neg(q_b) \to q_{\neg b} \\ \wedge(q_b, q_{b'}) \to q_{b \wedge b'}, \quad \vee(q_b, q_{b'}) \to q_{b \vee b'} \end{array} \;\middle|\; b, b' \in \{0, 1\} \right\}. \tag{2.2}$$

This expression uses an obvious short-hand, using 0 for *false* and 1 for *true*: for instance the rule $\vee(q_b, q_{b'}) \to q_{b \vee b'}$ actually expands to $\vee(q_0, q_0) \to q_0$ as $q_0 \vee q_0$ yields 0, $\vee(q_0, q_1) \to q_1$ as $q_0 \vee q_1$ yields 1, and so forth. Thus there are actually twelve rules in Δ. Considering now the term

$t =$  ,

we have the following possible reduction:



Thus $t \to_\Delta^* q_1 \in F$: it is accepted by $\mathcal{A}$. Note that the first three transformations actually result from the application of several transition rules at once; it would otherwise have been somewhat tedious to represent each and every configuration. Indeed, there are in total nine rewriting steps, or ten configurations (that is, elements of $\mathcal{T}(\mathbb{A} \cup Q)$), counting the initial configuration with the pristine term t. Note that

those steps could be performed in many different orders although that does not matter for our purposes – for instance, the left subtree could have been entirely reduced to q_1 before touching the right subtree, with the same result. Here is a breakdown of the rules which were used at each accelerated step:

(1) $\bot \to q_0, \top \to q_1 \in \Delta$

(2) $\wedge(q_0, q_1) \to q_0, \neg(q_0) \to q_1 \in \Delta$

(3) $\neg(q_0) \to q_1, \vee(q_0, q_1) \to q_1 \in \Delta$

(4) $\wedge(q_1, q_1) \to q_1 \in \Delta$

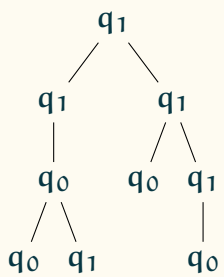
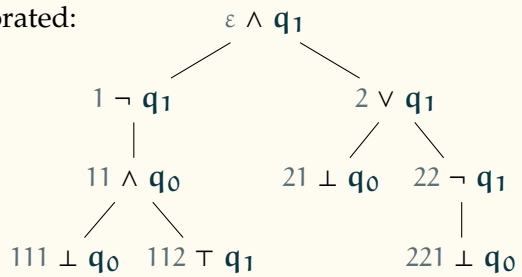
An inconvenient feature of reductions of the kind presented above is that there is no history of the intermediate steps, or record of which subtrees were accepted in which states; such knowledge often proves quite useful. Thus it is customary not to reason directly in terms of rewriting, but in terms of such a history, which is called a *run*. This generalises runs for finite-state automata, which are simply the words $q_0 q_1 \dots q_n$ of the sequences of states through which the automaton passes. Going back to our earlier example on words, the run of the FSA $\mathcal{A}_1$ on $w = abba$ was the word $\rho = q_0 q_1 q_1 q_1 q_2$. In the case of words, $\#\rho = \#w + 1$ because of the initial state, but in the case of BUTA there are no initial states, and so a run will be a tree of the exact same shape as the input term. The requirement is of course that this tree has to be decorated in accordance to the transition rules. Thus, formally, a run of a tree automaton $\mathcal{A}$ on a term $t \in \mathcal{T}(\mathbb{A})$ is a tree $\rho : \mathcal{P}(t) \to Q$ such that for all nodes $\alpha \in \mathcal{P}(t)$, or equivalently $\alpha \in \mathcal{P}(\rho)$,

run (BUTA)

$$t(\alpha)(\rho(\alpha.1), \dots, \rho(\alpha.n)) \to \rho(\alpha) \in \Delta\,. \tag{2.3}$$

A run ρ is a q-run if $\rho(\varepsilon) = q$, and it is called *accepting* (or *successful*) if it is a q_f-run, for some $q_f \in F$. This is an equivalent characterisation of the language accepted by a tree automaton, and in fact the one which is most commonly used: a term t is accepted by $\mathcal{A}$ if and only if there exists an accepting run of $\mathcal{A}$ on t. The nine-step reduction $t \to_\Delta^* q_1$ of the example above is succinctly summarised in the following accepting run:

accepting run

$\rho =$  or, decorated:  .

The second version shows the shared structure of t and ρ, as well as $t(\alpha)$ and $\rho(\alpha)$ together at each position α.

Extending the terminology for word languages, a tree language accepted by some BUTA is said to be a *regular tree language*; in other words, L is a regular tree language if and only if there exists a BUTA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$. There is another widely-used strain of tree automata accepting the same class of languages: as was hinted before, they are non-deterministic *top-down finite tree automata*. A few technicalities notwithstanding, moving from the definition of bottom-up to that of top-down automata is pretty much a matter of rebranding final states

regular tree language

top-down finite tree automata

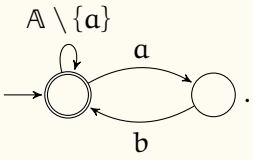
There are a few technicalities to get the definition of top-down automata. Destroying the tree from the top down is problematic, so one uses unary states and rules of the form $q(f(u_1, \ldots, u_n)) \to f(q_1(u_1), \ldots, q_n(u_n))$. Better yet is to forgo the rewriting aspect and define acceptance directly in terms of runs.

into initial states, and changing the direction of the arrows. A quick word about determinism is warranted at this point: the flavour of BUTA which was defined earlier is non-deterministic. There is indeed nothing prohibiting two transitions $a \to p$ and $a \to q$, or in general any number of rules with identical left-hand sides, from cohabiting in the same automaton. A BUTA, no two transitions of which have the same left-hand side, is said to be deterministic. In the case of BUTA, determinism does not affect expressive power; however, the top-down variant does not share that property, the deterministic version being strictly weaker. Section 2.6[p37] focuses on such questions of expressive power, closure properties, etcetera. Top-down automata are not used at all in this thesis, but they are found as often as BUTA in the literature. One can think of them both as two equivalent models, bottom-up automata corresponding intuitively to the *evaluation* of a term, and top-down automata to the *generation* of terms.

We would be remiss to close a section on regular tree languages without pausing to mention the strong ties of automata to logics. An automaton is an acceptor: is accepts or rejects inputs, be they words or trees, according to whether they meet the requirements encoded into the automaton. A formula of some logic equipped with relevant predicates can fulfil the same function. To exemplify this, let us work on words on some alphabet $\mathbb{A}$, and consider the binary predicate S such that $S(\alpha, \beta)$ holds if the position β is the immediate successor of α – to extend this to trees, one would need to have a predicate for "first son", one for "second son", and so on. To test which symbol is at what position, we consider every symbol of $\sigma \in \mathbb{A}$ as a unary predicate, such that $\sigma(\alpha)$ holds if the symbol at position α is σ. Using first-order logic over the domain of positions and those predicates, one can then write specifications φ such as this:

$$\varphi = \forall \alpha,\ a(\alpha) \implies \exists \beta : S(\alpha, \beta) \wedge b(\beta)\ .$$

The formula φ is satisfied by exactly the set of words such that every occurrence of a is immediately followed by an occurrence of b. Let us compare this with the word automaton $\mathcal{A}$:

$\mathcal{A} =$ 

.

$\mathcal{A}$ accepts exactly the set of words which are models of φ. In that respect, it makes sense to speak of the language accepted, or described, by a formula, and to write statements such as $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$. This raises the interesting question of the respective expressive powers of classes of logic formulæ and classes of automata. There is an entire field, called descriptive complexity theory, dedicated to identifying the relationships between logics, formal machines, and decision problems. Of singular interest is the well-known 1960 theorem of Büchi, showing that regular word languages are exactly described by weak monadic second-order logic with one successor (*WS1S*), with the predicates defined above. Extending this to trees, we let k be the maximal arity of symbols in a ranked alphabet $\mathbb{A}$, and define k successor relations as hinted above. The resulting logic is called weak monadic second-order logic with k successors (*WSkS*); its expressive power covers exactly the regular tree languages, as shown by Thatcher and Wright in 1968, and

WS1S

WSkS

by Doner in 1970. The reader eager to learn *much* more about this subject is advised to consult the third chapter of [Comon et al., 2008] and its bibliographic notes. The thirteenth chapter of [Hosoya, 2010] also provides a short introduction to logic on trees, with a slant towards XML logic-based queries.

As Part III focuses specifically on TAGE and their decision problems, its introductory chapter extends the present section.

2.5 Tree Automata With Global Constraints

There is an aspect which is lacking in both the branching varieties of tree automata seen in the previous section, and the tree-walking automata of Part IV of this thesis: neither can test whether two subterms are the same. For instance, the languages

$$L_= = \{ f(u,u) \mid f \in \mathbb{A}_2, u \in \mathcal{T}(\mathbb{A}) \} \quad \text{and} \tag{2.4}$$

$$L_\neq = \{ f(u,v) \mid f \in \mathbb{A}_2;\ u,v \in \mathcal{T}(\mathbb{A});\ u \neq v \} \tag{2.5}$$

are both non-regular, meaning that there is no bottom-up tree automaton capable of recognising them. Yet that kind of tests is quite worthwhile; for instance Part II illustrates the connexions of equality testing to rewriting and how this helps in tree model-checking, and Part III presents more examples in cryptography and XML processing, as well as an overview of the 30-year long history of tree automata extended with such tests.

The non-regularity of $L_=$ is easily proven with a pumping argument, similarly to $a^n b^n$ in the word case. Intuitively, both brothers at positions 1 and 2 are evaluated independently, and the automaton, having no memory, can store only a finite amount of information about each of them.

This section presents a relatively recent class, introduced in Emmanuel Filiot's PhD thesis [Filiot, 2008] and in the papers [Filiot, Talbot & Tison, 2008, 2010], called *tree automata with global equality and disequality constraints* (*TAGED*), which will sometimes be rendered as $TA^{=}_{\neq}$.

tree automata with global equality and disequality constraints
TAGED
$TA^{=}_{\neq}$

▽ Definition 2.2: TAGED

A tree automaton with global equality and disequality constraints $\mathcal{A}$ is a tuple $\langle \mathbb{A}, Q, F, \Delta, \cong, \not\cong \rangle$, where

$\langle \mathbb{A}, Q, F, \Delta \rangle$	is a bottom-up tree automaton,
$\cong \subseteq Q^2$	is the *equality relation*, or *constraints*,
$\not\cong \subseteq Q^2$	is the *disequality relation*, or *constraints*.

TAGED function almost exactly in the same way as BUTA – indeed they *are* BUTA – in that they have the same basic notion of runs. The difference is that TAGED are more restrictive: in order for a run ρ of the *underlying BUTA* $\mathrm{ta}(\mathcal{A}) = \langle \mathbb{A}, Q, F, \Delta \rangle$ to be a run of $\mathcal{A}$ as well, it needs to be compatible with the constraints. A run ρ is *compatible* with the equality constraints of $\cong$ if, whenever two positions α and β are evaluated into states p and q such that $p \cong q$, then the subterms under those positions are equal. Equality is of course meant extensionally, that is to say, $u = v$ if $\mathcal{P}(u) = \mathcal{P}(v)$ and $\forall \alpha \in \mathcal{P}(u),\ u(\alpha) = v(\alpha)$. Thus, compatibility with the equality constraints is expressed as

$\mathrm{ta}(\mathcal{A})$: *underlying automaton*

compatibility with $\cong, \not\cong$

$$\forall \alpha, \beta \in \mathcal{P}(t) : \rho(\alpha) \cong \rho(\beta) \implies t|_\alpha = t|_\beta \ . \tag{2.6}$$

In the same way, ρ is compatible with the disequality (or difference) constraints if

$$\forall \alpha, \beta \in \mathcal{P}(t) : \rho(\alpha) \not\cong \rho(\beta) \implies t|_\alpha \neq t|_\beta \ . \tag{2.7}$$

If $\not\cong$ is not assumed to be irreflexive, this last definition can be extended into

$$\forall \alpha, \beta \in \mathcal{P}(t) : \alpha \neq \beta \wedge \rho(\alpha) \not\cong \rho(\beta) \implies t|_\alpha \neq t|_\beta \ . \tag{2.8}$$

Furthermore, a run ρ of the TAGED $\mathcal{A}$ is accepting for $\mathcal{A}$ if it is accepting for $\mathrm{ta}(\mathcal{A})$, which is to say, if $\rho(\varepsilon) \in F$. If $\not\cong$ is empty, $\mathcal{A}$ is said to be *positive*, or a tree automaton with global equality constraints (*TAGE*, $TA^{=}$). If $\cong$ is empty, then it is said to be *negative*, or a tree automaton with global disequality constraints (*TAGD*, $TA^{\neq}$). In this thesis, we are exclusively interested in equality constraints, and thus focus on the positive sub-class.

TAGE
$TA^{=}$
TAGD
$TA^{\neq}$

TAGED are closed by union and intersection, and we take notations for the corresponding constructions. Letting $\mathcal{A}$ and $\mathcal{B}$ be two TAGED, $\mathcal{A} \uplus \mathcal{B}$ is another TAGED such that $\mathcal{L}(\mathcal{A} \uplus \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$, which we call the *disjoint union* of $\mathcal{A}$ and $\mathcal{B}$. The construction simply consists in renaming states so that the sets of states of $\mathcal{A}$ and $\mathcal{B}$ are made disjoint, and trivially merging the automata – constraints included. Likewise, $\mathcal{A} \times \mathcal{B}$ is the TAGED such that $\mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$, obtained through the usual product construction, with additional provisions for the constraints.

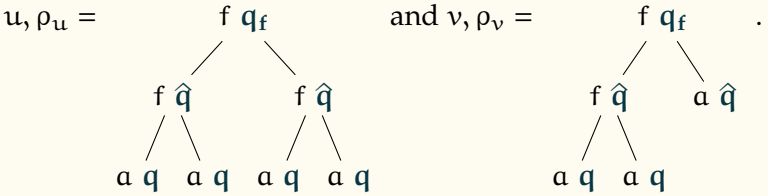
$\mathcal{A} \uplus \mathcal{B}$: *disjoint automata union*

$\mathcal{A} \times \mathcal{B}$: *product automaton*

It is clear that TAGED are at least as expressive as BUTA, as those classes coincide exactly when $\cong$ and $\not\cong$ are both empty. They are in fact strictly more expressive, since they can recognise languages which BUTA cannot, such as the aforementioned $L_{=}$ and $L_{\neq}$. As a first example, consider the following TAGE $\mathcal{A}$, with $\mathbb{A} = \{ a/_0, f/_2 \}$, $Q = \{ q, \hat{q}, q_f \}$, $F = \{ q_f \}$, $\hat{q} \cong \hat{q}$, $\hat{q} \not\cong q_f$, and

$$\Delta = \{ f(\hat{q}, \hat{q}) \to q_f,\ f(q, q) \to q,\ f(q, q) \to \hat{q},\ a \to q,\ a \to \hat{q} \} .$$

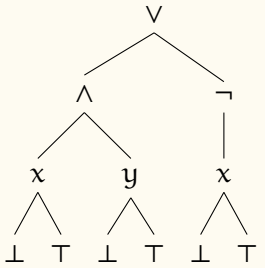
Below are two terms $u, v \in \mathcal{T}(\mathbb{A})$, and corresponding runs of the underlying BUTA. For clarity, the terms and their runs are superimposed:



Both those runs are accepting for the underlying BUTA, since q_f is a final state, and both of these sport two occurrences of $\hat{q}$, at positions 1 and 2. In the first case we have $u|_1 = u|_2 = f(a, a)$, and therefore ρ_u is compatible with the equality constraint $\hat{q} \cong \hat{q}$, and u is accepted by the TAGE $\mathcal{A}$. Contrast this with the second term, whose run has the same two instances of $\hat{q}$, but one is over $v|_1 = f(a, a)$ while the other is over $v|_2 = a$. Obviously $v|_1 \neq v|_2$, which violates the equality constraint $\hat{q} \cong \hat{q}$, and v is rejected by $\mathcal{A}$. Thus it is clear that $\mathcal{A}$ accepts terms if and only if their left subtree is equal to their right, that is to say, $\mathcal{L}(\mathcal{A}) = L_{=}$. The example suffices to show the expressive power of TAGE to be strictly greater than that of BUTA.

The ur-example of the expressive power of TAGED is given by the extension of the BUTA accepting true boolean expressions, seen earlier, to full propositional logic; the equality constraints provide everything needed to encode propositional

variables. The propositional formulæ are again represented as trees, with a little technicality when it comes to variables, taken from a set $\mathbb{X}$. A variable x is represented by the tree $x(\top, \bot)$, the idea being to evaluate each such tree in such a way that the corresponding state ν_x, constrained by $\nu_x \cong \nu_x$, appears on either $\top$ or $\bot$, thereby imposing a valuation over the formula. For instance, the propositional formula $(x \wedge y) \vee \neg x$ is represented by the tree



. (2.9)

The alphabet is the same as before, with the addition of the free variables of the formula, taken as binary symbols: $\mathbb{A} = \{\wedge, \vee/_2, \neg/_1, \top, \bot/_0\} \uplus \mathbb{X}$. The states are unchanged as well, with the addition of one fresh state per variable: $Q = \{q_0, q_1\} \uplus \{\nu_x \mid x \in \mathbb{X}\}$ and $F = \{q_1\}$. All the existing transitions of (2.2)$_{[p32]}$ are kept, and the following are added for each $x \in \mathbb{X}$:

$$\top \to \nu_x,\ \bot \to \nu_x,\ x(q_0, \nu_x) \to q_1,\ x(\nu_x, q_1) \to q_0\ .$$

Lastly, as said above, we add the constraint $\nu_x \cong \nu_x$. The resulting automaton accepts the representation of a propositional formula φ if and only if φ is satisfiable.

As we shall see in the next section – and again in much more detail in part III – such expressive power comes at the cost of a considerable increase in algorithmic complexity for most decision problems, up to and including the loss of decidability for some. Taking advantage of automata with constraints in practical contexts therefore requires fine-tuned algorithms and heuristics, and resorting to semi-algorithms or approximated procedures is sometimes inevitable.

2.6 Decision Problems and Complexities

Throughout this thesis, we keep referring to various kinds of automata, their decision problems, and the algorithmic complexity or decidability of the latter. This section purports to serve as a convenient reference sheet on those matters, to refresh memories and make it easier to compare the merits and pitfalls of different formalisms.

The nub of the matter is summarised in Figure 2.2, where for each kind of automata of particular interest to this thesis, and for each boolean closure property, for the determinisation property, and for each decision problem in Figure 2.3, we have written down the corresponding result or complexity class, as we could find them in the literature. In order to fit what amounts to over one hundred complexity results in such a small space, the figure employs systematic abbreviations for the complexity classes involved, which are decrypted below. Complexity classes Γ appear in the columns corresponding to decision problems, and are defined by the grammar

Uniform Membership

Note that the fifth column of Fig. 2.2 deals with *uniform* membership, in the sense of Fig. 2.3. In general, in this thesis, we say simply "membership" to mean "uniform membership", except where the difference matters, in which case the context will make it clear.

	$\cup$-closed?	$\cap$-closed?	$\neg$-closed?	determinisable?	$t \in \mathcal{L}(\mathcal{A})$?	$\mathcal{L}(\mathcal{A}) = \emptyset$?	$\lvert\mathcal{L}(\mathcal{A})\rvert = 1$?	$\lvert\mathcal{L}(\mathcal{A})\rvert \in \mathbb{N}$?	$\mathcal{L}(\mathcal{A}) = \mathcal{T}(\mathcal{A})$?	$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$?	$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$?	$\mathcal{L}(\bigcap_i \mathcal{A}_i) = \emptyset$?
NFA	P^1	P^2	X	X	P^1	P^1	P	P^1	$\overline{P_s}$	$\overline{P_s}$	$\overline{P_s}$	$\overline{P_s}$
DFA	P^1	P^2	P^1	C	P^1	P^1	P^1	P^1	P^2	P^2	P^2	$\overline{P_s}$
BUTA	P^1	P^2	X	X	P^1	P^1	P	P^2	$\overline{X}$	$\overline{X}$	$\overline{X}$	$\overline{X}$
DBUTA	P^1	P^2	P^1	C	P^1	P^1	P	P^2	P^2	P^2	P^2	$\overline{X}$
TAGED	P^1	P^2	–	–	$\overline{P!}$	R		R	E	E	E	R
TAGE	P^1	P^2	–	–	$\overline{P!}$	$\overline{X}$		$\overline{X}$	E	E	E	2X
RTA	P^1	P^2	–	–	$\overline{P!}$	P^1		P^3	E	E	E	2X
TAGD	P^1	P^2	–	–	$\overline{P!}$	X!						
TWA	P^1	P^1	–	–	P	$\overline{X}$				$\overline{X}$	$\overline{X}$	
DTWA	+	P^1	+	C	P^1	$\overline{X}$						

Figure 2.2: Automata, their closure properties and decision complexities.

$\Gamma :=$ γ deterministic time complexity γ
$\gamma!$ non-deterministic time complexity γ
γ_s space complexity γ
$\overline{\Gamma}$ Γ-complete
R recursive, decidable, nothing more specific known
E co-recursively enumerable, co–semi-decidable, but undecidable
unknown – at least to us, and at the time of writing;

and, n denoting the size of the input and p some polynomial function:

$\gamma :=$			
	C	constant	$O(1)$
	P^k	polynomial of degree at most k	$O(n^k)$
	P	polynomial, unspecified degree	$O(n^{O(1)})$
	X	exponential	$O(2^{p(n)})$
	2X	doubly exponential	$O(2^{2^{p(n)}})$.

For instance, P designates the class PTIME, P! means non-deterministic polynomial time, that is to say the class NP, $\overline{P!}$ is therefore the class of NP-complete problems, $\overline{P_s}$ is PSPACE-complete, P^2 means "solvable in quadratic time", X! is NEXPTIME, 2X is 2-EXPTIME, and so forth. In the context of closure and determinisation properties – that is to say, the first four columns of the figure – a symbol γ' appears, with the following possibilities:

$\gamma' :=$		
	γ	closed, construction of size γ, done in time γ
	+	closed, time and size unspecified
	–	not closed.

There remains to specify the size of the inputs, which are automata and terms. The size of a tree has already been defined, in section 2.2, as how many nodes it has. The size of a TAGED is defined roughly, following [Comon et al., 2008], as the number of symbols required to encode it:

$\|\mathcal{A}\|$: *size of an automaton* $\mathcal{A}$

$$\|\langle \mathbb{A}, Q, F, \Delta, \cong, \not\cong \rangle\| = |Q| + 2 \cdot \left(|\cong| + |\not\cong|\right) + \sum_{\sigma(p_1,\ldots,p_n)\to q \in \Delta} (n+2) \ .$$

Membership:	*in:* t	*out:* $t \in \mathcal{L}(\mathcal{A})$?
Uniform Membership:	*in:* $\mathcal{A}, t$	*out:* $t \in \mathcal{L}(\mathcal{A})$?
Emptiness:	*in:* $\mathcal{A}$	*out:* $\mathcal{L}(\mathcal{A}) = \varnothing$?
Singleton Set Property:	*in:* $\mathcal{A}$	*out:* $\lvert\mathcal{L}(\mathcal{A})\rvert = 1$?
Finiteness:	*in:* $\mathcal{A}$	*out:* $\lvert\mathcal{L}(\mathcal{A})\rvert \in \mathbb{N}$?
Universality:	*in:* $\mathcal{A}$	*out:* $\mathcal{L}(\mathcal{A}) = \mathcal{T}(\mathbb{A})$?
Containment:	*in:* $\mathcal{A}, \mathcal{B}$	*out:* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$?
Equivalence:	*in:* $\mathcal{A}, \mathcal{B}$	*out:* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$?
Intersection Emptiness:	*in:* $\mathcal{A}_1, \ldots, \mathcal{A}_n$	*out:* $\mathcal{L}(\bigcap_i \mathcal{A}_i) = \varnothing$?

Figure 2.3: Decision problems: inputs and outputs.

Note that the alphabet is not included in the size. Furthermore, this definition also applies to TAGE, TAGD, RTA (cf. Sec. 5.2.2), and even BUTA, by seeing them as TAGED $\langle \mathbb{A}, Q, F, \Delta, \varnothing, \varnothing \rangle$; by extension, it carries over to FSA. In the case of TWA (Sec. 8.1[p144]), we have simply

$$\|\langle \mathbb{A}, Q, I, F, \Delta \rangle\| = |Q| + 5 \cdot |\Delta| \,.$$

Here end the preliminaries. As summarised in section 1.4[p19], each of the next parts of the thesis deals with a main domain of our contributions, and opens with a survey of that domain.

— Part II —

Approximating Linear Temporal Logic Safety Properties over Rewrite-Rules Sequences

Chapter 3
Term Rewriting for Model-Checking

Contents

—Where there is no epigraph.

Model-checking techniques are in no way limited to finite state spaces – a fact that section 1.2[p13] has already touched upon. The method which we develop in the next chapter relies on rewriting as its central paradigm, with close ties to tree regular model-checking, reachability analysis, and rewriting logic. The present introductory chapter summarises the problem at hand, offers some elements of context about related work in those fields, and stresses some results and concepts of notable bearing on what follows.

The goal is to check temporal properties of a system – be it a program, a circuit, or a cash machine – whose states are represented by trees and whose behaviour is encoded into a term rewriting system $\mathcal{R}$. The properties do not deal with the evolutions of the state of the system, but with the succession of its actions. It is assumed in this context that the rewrite rules of $\mathcal{R}$ correspond to pertinent events of the system. Those sequences of rewrite rules that capture executions of the system, which are defined formally and called *maximal rewrite words* in the next chapter, therefore provide the basis upon which the temporal semantics are constructed.

• versus ○

In $\Box(X \Rightarrow \bullet Y)$, $\bullet$ is the "next" operator of temporal logic. In that case, it is actually a *strong* next operator, as opposed to the *weak* next, which is written $\circ$. This is explained in detail in section 4.1.2[p57], but it is of no importance for the moment.

Consider for instance an initial language $\Pi \subseteq \mathcal{T}(\mathbb{A})$, a rewrite system $\mathcal{R}$ and the LTL property $\Box(X \Rightarrow \bullet Y)$, where $X, Y \subseteq \mathcal{R}$; that is to say, X and Y are sets of rewrite rules, or actions, of the system under consideration. This property signifies that whenever an accessible term is transformed by some rewrite rule in X, the resulting tree can in turn be rewritten by some rule in Y, and not by any rule not in Y. This is illustrated by Figure 3.1. More concretely, if $\mathcal{R}$ models a cash machine, and $X = \{\mathsf{ask\_PIN}\}$ and $Y = \{\mathsf{auth}_1, \mathsf{auth}_2, \mathsf{cancel}\}$, then this property can be read as "whenever the user enters his or her PIN, then something happens immediately after, and that can only be either the authentication of the user – through either of the two available methods – or the cancellation of the transaction; this excludes other possible but undesirable actions, such as sending the PIN over the network."

Figure 3.1 is borrowed, with slight modifications, from [Courbis et al., 2009].

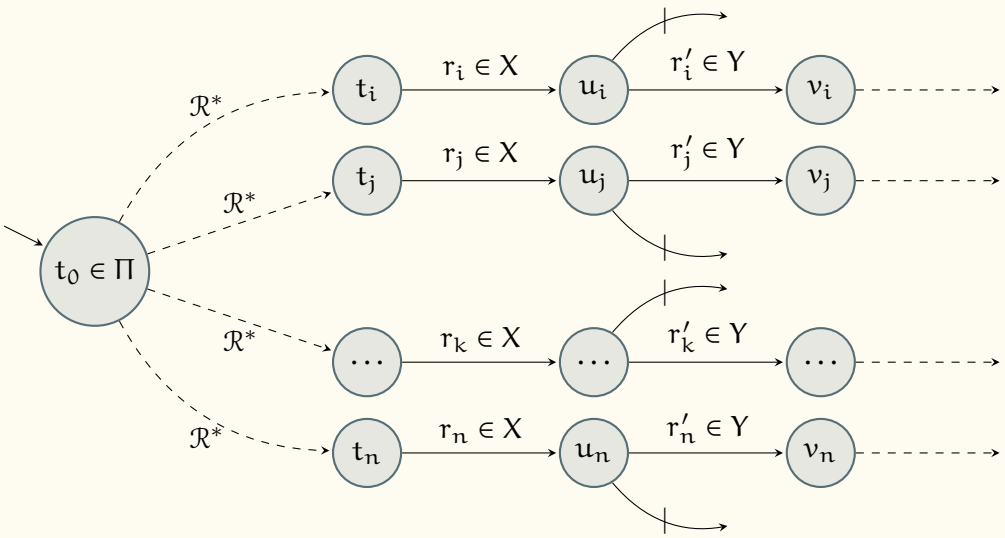


Figure 3.1: Executions of a rewrite system satisfying $\Box(X \Rightarrow \bullet Y)$.

Note that `ask_PIN` etcetera are, in this context, rewrite rules on trees representing the state of the machine.

As we shall see in the next chapter, the method which we study in order to answer such verification problems relies on the computation of automata corresponding to the tree languages reached after certain numbers of rewriting steps. In that respect, it is closely related to, and in a way generalises, the methodology of reachability analysis over term rewriting systems. Where reachability analysis boils down to an equation of the form $\mathcal{R}^*(\Pi) \cap \mathcal{B} = \emptyset$, the verification of temporal properties requires the decision – or at least approximation – of larger language equations – called *rewrite propositions* in the next chapter – such as, for the example of $\Box(X \Rightarrow \bullet Y)$,

$$[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \emptyset \ \wedge \ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \ . \tag{3.1}$$

A large part of our work consists in mechanically generating such equations on the basis of the temporal property under consideration, extending previous work in [Courbis et al., 2009]. However, for the purposes of this introduction, let us keep the question of the provenance of (3.1) in temporary stasis, and focus instead on how and to what extent the existing techniques of reachability analysis can be brought to bear on such equations, once they are obtained.

3.1 On the Usefulness of Rewriting for Verification

Before we begin, it is worth saying a few general words concerning the pertinence of choosing term rewriting systems as the central formalism in which to model the system to verify. It is clear that TRS are very expressive: they are indeed a Turing-complete model of computation, borne of the traditions of λ-calculus. Beyond the raw expressive power, they often allow simple, readable, clean and compact models of complex systems and programs. For instance [Jones, 1987; Jones & Andersen, 2007] show how even bare TRS can easily encode higher-order functions and other bells and whistles of modern functional languages of the ML

family; see also the corresponding examples of [Genet, 2009, Eg. 53, 54], which illustrate how terse and straightforward the TRS encodings are, even with regards to the very expressive original ML-style syntax.

Term rewriting systems have been used intensively in automated deduction for about four decades, and can model parallel as well as sequential systems: rewriting can naturally be interpreted as transformations occurring in parallel [Meseguer, 1992]. Used to encode high-level specifications of cryptographic protocols, they have been put to work [Genet & Klay, 2000; Armando, Basin, Boichut, Chevalier, Compagna, Cuéllar, Drielsma, Héam, Kouchnarenko, Mantovani, Mödersheim, von Oheimb, Rusinowitch, Santiago, Turuani, Viganò & Vigneron, 2005], with considerable success, on proving their security, insecurity, or the necessity of specific countermeasures. Indeed, the techniques allow both to produce proofs of correctness and to exhibit examples of attacks, be they new non-trivial attacks on well-known, classical protocols of the literature [Chevalier & Vigneron, 2002], or semi-expected attacks against freshly developed industrial protocols with relaxed countermeasures [Heen, Genet, Geller & Prigent, 2008], thereby establishing the critical status of the countermeasures in question.

At the other end of the spectrum of abstraction, TRS have been used to provide models for much lower-level semantics, for instance in the case of Java Bytecode programs [Boichut, Genet, Jensen & Roux, 2007; Barré, Hubert, Roux & Genet, 2009], for which safety and security properties are then proven through reachability analysis – as in most of the works above. Back at higher levels of abstraction, let us also mention similar verifications to the calculus of communicating systems [Courbis, 2011], and more generally the rich ecosystem surrounding TRS as a central formal model amenable to both execution and verification through classical model-checking, abstract interpretation, static analysis, interactive proofs etcetera [Eker, Meseguer & Sridharanarayanan, 2003; Feuillade, Genet & Tong, 2004; Genet, 1998; Takai, 2004; Clavel, Palomino & Riesco, 2006].

rewriting logic

Another thriving approach using TRS as the central tool is *rewriting logic* [Meseguer, 1992; Martí-Oliet & Meseguer, 1996, 2002], intended as a unifying logical framework in which other logics can be implemented, and a natural model of concurrent systems. In recent years, new results in that field have deeply extended the spectrum of its applications to verification [Escobar & Meseguer, 2007; Serbanuta, Rosu & Meseguer, 2009; Boronat, Heckel & Meseguer, 2009; Ölveczky, 2010], especially in relation with temporal logic for rewriting [Meseguer, 2008; Eker et al., 2003; Bae & Meseguer, 2010].

To further extend the versatility of the rewrite-based techniques, reachability analysis can be guided by temporal properties, expressed for instance in LTL, as seen in works such as [Boyer & Genet, 2009]'s Regular LTL, where the rewrite relation is abstracted into a finite Kripke structure susceptible to standard model-checking approaches, and [Courbis et al., 2009], which our work generalises. Yet while both endeavours yield, in fine, a positive approximated procedure over LTL, there is a major difference of viewpoint between the two: [Boyer & Genet, 2009] expresses properties on the states, while we and [Courbis et al., 2009] focus on properties of actions, following more closely the philosophy presented in [Meseguer, 1992].

Furthermore, unlike [Bae & Meseguer, 2010], where LTL model-checking is performed over finite structures, the approach exposed in the next chapter handles temporal formulæ over infinite state systems. In this sense, it is close to [Escobar & Meseguer, 2007]. However, in spite of its simplicity for practical applications, our approach does not permit – in its current state, at least – to consider equational theories.

3.2 Reachability Analysis for Term Rewriting

We have already touched upon the general gist of reachability analysis in section 1.2[p13], though the discourse took place in the context of transducers. We are now placed in the more general context of term rewriting systems; after all, a tree transducer can be seen as a special kind of rewriting system, where the rewriting is done bottom-up, or top-down. The general *reachability problem* is whether a term t can be rewritten into another term u – or whether u is reachable from t – by means of a rewrite system $\mathcal{R}$, which is written $t \to_{\mathcal{R}}^{*} u$. It is plain to see that this problem is decidable if $\mathcal{R}$ is noetherian. To drive that point home, consider the rewrite tree rooted in t: any rewriting step creates finitely many branches – there are finitely many rewrite rules applying on a finite tree – and no path may be infinite. Therefore the rewrite tree is finite, and can be explored in finite time; contrariwise, a non-terminating TRS would produce a tree with at least some infinite paths.

reachability problem

Of course, on top of being very inefficient for any non-trivial noetherian TRS – "finite" does not imply "tractable" – this approach breaks down if there is an infinite number of initial terms t for which such a check must be made, as is often the case. For instance, in program verification, good behaviour must be enforced for all possible inputs, of which there are typically infinitely many: our initial language Π is not and should not be required to be finite. Thus even in the – fairly restrictive – case where $\mathcal{R}$ is noetherian, an exhaustive check of the rewrite tree is either impossible or impractical. Fortunately, there are several other ways in which the problem may be approached, and sometimes decided even if Π is infinite and $\mathcal{R}$ non-terminating; three of those were mentioned in section 1.2, namely (1) the characterisation of classes for which the set of reachable terms $\mathcal{R}^*(\Pi)$ is regular, and therefore can be represented exactly by an automaton, (2) acceleration techniques, and (3) approximations and approximated procedures – over-approximations and positive procedures, mainly.

While our interest lies in this last approach, our approximated methods do involve languages of the form $\mathcal{R}^*(\Pi)$, as can be seen to appear in (3.1), and thus the quality of exact methods becomes tributary to that of our own. An important negative result in that respect, which was shown in [Gilleron & Tison, 1995], is that it is not decidable, given a regular language Π and a rewrite system $\mathcal{R}$, whether $\mathcal{R}^*(\Pi)$ is regular. This remains undecidable even in the case of noetherian and confluent (2.1)[p29] linear rewrite systems. On the other hand, the literature is rife with constructive, positive results concerning the preservation of regularity under forward closure, that is to say, "under which conditions on the rewrite system $\mathcal{R}$ is $\mathcal{R}^*(\Pi)$ still regular?"

3.2.1 Preservation of Regularity Through Forward Closure

Preservation and Alphabet

Interestingly, [Gyenizse & Vágvölgyi, 1998] points out that preservation of regularity through forward closure depends on both $\mathcal{R}$ and the underlying alphabet $\mathbb{A}$. Indeed, the paper exhibits a TRS $\mathcal{R}$ and alphabet $\mathbb{A}$ such that $\mathcal{R}$ preserves regularity for languages of $\wp(\mathcal{T}(\mathbb{A}))$, but not for those of $\wp(\mathcal{T}(\mathbb{A}'))$, where $\mathbb{A}' = \mathbb{A} \uplus \{ f/_1 \}$. See also [Otto, 1998]. We always consider preservation with the alphabet $\mathbb{A}$ fixed in this thesis.

ground rewrite systems

A survey of such results appears in the research report [Feuillade et al., 2004], and a more recent one can be found in the habilitation thesis [Genet, 2009, Sec. 2.1.1], which is our main source for the next paragraphs. Let us just mention the best-known classes of forward-closure regularity-preserving TRS:

(1) **ground rewrite systems**, that is to say TRS making no use of variables. The preservation of regularity was first shown in [Brainerd, 1969], where such TRS were simply called regular systems, and proven again in a more general context in [Dauchet & Tison, 1990].

(2) **right-linear and monadic systems**, where *monadic* means that the left-hand sides of the rules are not variables, and the right-hand sides are either some variable $x \in \mathbb{X}$ or of the form $\sigma(x_1, \ldots, x_n)$, with $\sigma \in \mathbb{A}_n; x_1, \ldots, x_n \in \mathbb{X}$ [Salomaa, 1988].

(3) **linear and semi-monadic systems**, where *semi-monadic* means that right-hand sides of the rules are of the form $\sigma(u_1, \ldots, u_n)$, where $\forall k \in [\![1, n]\!]$, $u_k \in \mathbb{X} \cup \mathcal{T}(\mathbb{A})$ [Coquidé, Dauchet, Gilleron & Vágvölgyi, 1991].

(4) **linear decreasing systems**, where *decreasing* means that for each rule $l \to r$, the variables common to left- and right-hand sides occur only at depth one in the right-hand side; that is to say, $\forall l \to r \in \mathcal{R}, \alpha \in \mathcal{P}(r);\ r(\alpha) \in \mathcal{V}(l) \cap \mathcal{V}(r) \Rightarrow \#\alpha = 1$ [Jacquemard, 1996].

(5) **right-linear decreasing systems**, defined as (4), but for which only right-linearity is required [Nagaya & Toyama, 1999, 2002]. Note that this class is more general than all the other enumerated so far.

(n) Many other such classes, some even more general, have been isolated in the late 90s and early 2000s, although in many cases, they are not characterised by simple syntactic restrictions, as are the classes above. Let us mention, without definition, the classes of systems that are:

- **a.** **linear generalised semi-monadic** [Gyenizse & Vágvölgyi, 1998],
- **b.** **constructor-based** [Réty, 1999],
- **c.** **linear finite-path overlapping** [Takai, Kaji & Seki, 2000],
- **d.** **right-linear finite-path overlapping** [Takai et al., 2000],
- **e.** **linear I/O separated layered transducing** [Seki et al., 2002],
- **f.** **well-oriented** [Bouajjani & Touili, 2002],
- **g.** **linear generalised finite-path overlapping** [Takai, 2004].

It should be noted that the constructor-based class distinguishes itself from the others by imposing restrictions on the language Π – which none of the other classes do – in order to weaken the necessary restrictions on the system $\mathcal{R}$. Of course, this results in it being incomparable to everything else. Also of note is that linear I/O separated layered transducing rewrite systems correspond exactly to linear bottom-up tree transducers.

Figure 3.2 shows the relationship between all those classes; the most expressive classes are at the top, and the least expressive at the bottom of the graph; the styles of the nodes reflect how they integrate with the tree automata completion algorithms which we sketch below – the legend is given by $(3.2)_{[p47]}$. It is worth

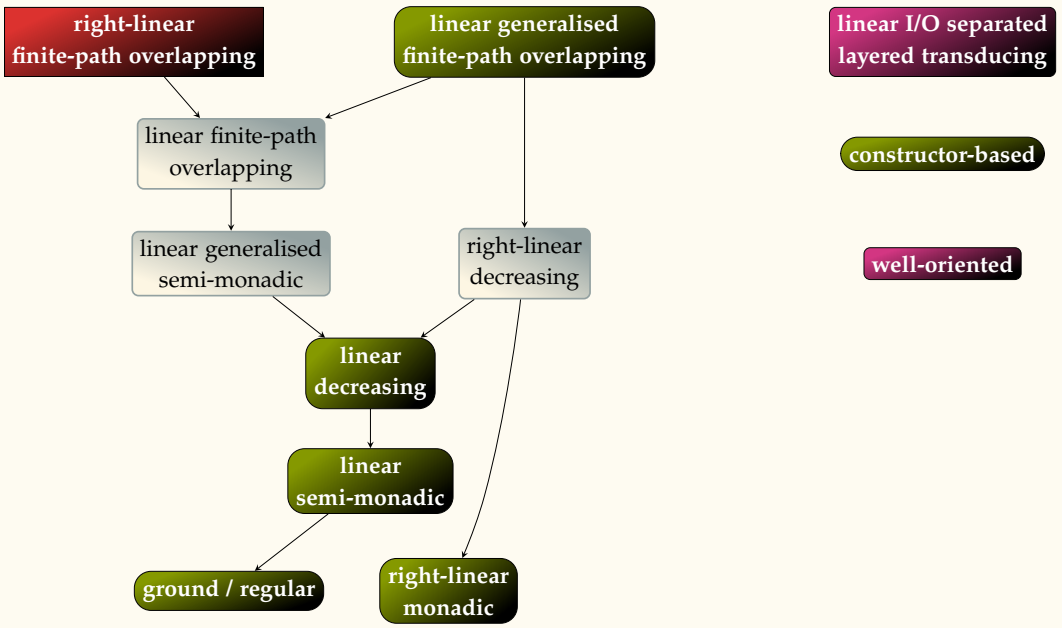


Figure 3.2: Forward-closure regularity-preserving classes of TRS.

noting that those forward-closure regularity-preserving classes contain no shortage of non-terminating, non-confluent rewrite systems. Take for instance the trivial ground rewrite system

$$\{ a \to f(a),\ a \to g(a) \},$$

which is neither terminating nor confluent, and yet is regularity-preserving by dint of being ground. This emphasises the point that decidability of reachability, and termination and confluence, are orthogonal issues. This is fortunate, as the programs and protocols which we mean to verify have no a priori reason to be terminating.

3.2.2 Tree Automata Completion Algorithm

tree automata completion

Unfortunately, those classes are still fairly restrictive, and so strict over-approximations remain occasionally unavoidable. The method through which such approximations are computed is inspired by the classical Knuth & Bendix completion algorithm [Knuth & Bendix, 1970]. It is referred to as *tree automata completion*, and was first introduced in [Genet, 1998]; it has been progressively refined over the last decade, for instance by removing left-linearity preconditions for sound over-approximations [Boichut, Héam & Kouchnarenko, 2008], and extending to equational completion [Genet & Rusu, 2010]. It is also implemented in tools such as Timbuk, as documented in [Feuillade et al., 2004; Genet, 2009]. The general idea of the completion algorithm is, starting with a BUTA $\mathcal{A}_0$ such that $\mathcal{L}(\mathcal{A}) = \Pi$, to compute successively $\mathcal{A}_1, \ldots, \mathcal{A}_{k+1}$, where $\mathcal{R}^i(\Pi) \subseteq \mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{A}_{i+1})$, for all $i \in [\![1, k]\!]$. The process stops when a fixpoint is reached, that is to say when $\mathcal{L}(A_k) = \mathcal{L}(\mathcal{A}_{k+1})$, or in practice when $\mathcal{A}_k = \mathcal{A}_{k+1}$; then $\mathcal{R}^{(*)}(\Pi) = \mathcal{L}(A_k) \supseteq \mathcal{R}^*(\Pi)$. Each $\mathcal{A}_{i+1}$ is obtained from $\mathcal{A}_i$ through a *completion step*, which rests on the joining of critical pairs between the rewriting systems $\mathcal{R}$ and $\mathcal{A}_i{:}\Delta$. In this context, a *critical pair* is a couple $(\sigma l, \sigma r)$, where $\sigma : \mathbb{X} \to \mathcal{A}_0{:}Q$ is a substitution and $l \to r \in \mathcal{R}$ is a rewrite rule, such that there is a state q with $\sigma l \to^*_{\mathcal{A}_i} q$ and $\sigma r \not\to^*_{\mathcal{A}_i} q$. For $\mathcal{A}_i$ to support

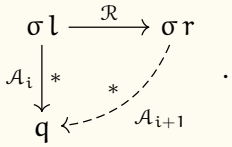
What we present here assumes left-linearity; in practice, this condition can be relaxed, or even done away with [Boichut et al., 2008].

completion step

See section 5.4[p115] for the notation $\mathcal{A}_i{:}\Delta$, in case it is not immediately obvious.

critical pair

the rewriting of $\sigma\, l$ into $\sigma\, r$, the critical pair must be joined in $\mathcal{A}_{i+1}$, following the diagram



This would be most simply accomplished by adding the "transition" $\sigma\, r \to q$ to $\mathcal{A}_i{:}\Delta$ to get $\mathcal{A}_{i+1}{:}\Delta$. However, that is complicated by the fact that $\sigma\, r$ may not be of the usual form $f(p_1, \ldots, p_n)$, and thus the new transition needs to be normalised beforehand. This is accomplished by introducing an equivalent set of BUTA transitions, inductively defined to accept the subterms of $\sigma\, r$ into fresh states $p_1, \ldots, p_n$. To guarantee the termination of the completion algorithm, appropriate abstractions may be introduced at that point, which consist in merging states in the normalised transitions, thereby causing the normalised version of the transition to accept a superset of the terms recognised by the original.

3.2.3 Exact Behaviours of Completion

The completion algorithms have some very interesting features: one of these is that, under certain assumptions on the abstraction function, if the completion terminates, then the result is exactly $\mathcal{R}^*(\Pi) = \mathcal{L}(A_k)$. This provides a new – and often more simple – way to prove that $\mathcal{R}^*(\Pi)$ is regular: it suffices to prove termination of the completion, under those assumptions [Genet, 2009, Sec. 3.3.1]. The node styles in Fig. 3.2[p46] reflect how the aforementioned regularity-preserving classes behave in that respect:

Proof Inherits Unknown Not Suitable . (3.2)

The four styles correspond respectively to: **(1)** classes for which a direct proof is provided in [Genet, 2009], **(2)** classes which inherit a proof from a more general one, **(3)** classes whose status is unknown, and **(4)** classes which are not amenable to this methodology. Besides the proofs, one directly obtains an exact result through the completion algorithm for the classes (1), provided that one does not use approximations. Furthermore, there exists an exact normalisation strategy to compute the approximation on the fly, such that the classes (1) yield an exact result without any human input. The exact normalisation strategy can also yield exact results for TRS that fall outside the known classes (1). Thus the same algorithm can yield exact results when possible, and over-approximations in the other cases. This even carries over to the case of equational completion.

3.2.4 One-Step Rewriting, and Completion

Given the nature of the type of properties we are interested in, and as can be seen on the example formula (3.1)[p42], our interest is not solely focused on the set of descendants $\mathcal{R}^*(\Pi)$, but also on expressions of the form $\mathcal{R}(\Pi)$, that is to say, one-step application of a rewrite system. Unfortunately, and quite surprisingly, this subject has not been studied extensively in the literature. Actually, we could

not find any specific paper focusing specifically on the issue, and the scant, brief mentions we could find are usually negative results, e.g. [Genet, 2009, Eg. 116] shows that doing one completion step does not actually work for that purpose. We discuss this example below.

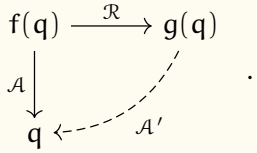
What is clear is that, in general, regularity is not preserved through one-step rewriting. By way of a counter-example, consider non-linear rewrite rules, especially non–right-linear ones such as $g(x) \to f(x,x)$. With the reminder that the language of ground terms of $f(x,x)$ is denoted by $L_=$ (2.4)$_{[p35]}$, it is immediate that

$$\{\, g(x) \to f(x,x) \,\}(\mathcal{T}(\mathbb{A})) = L_= \,,$$

and while $\mathcal{T}(\mathbb{A})$ is trivially regular, $L_=$ is, as we have already seen, known to be non-regular [Comon et al., 2008]. On the other hand, if the TRS is linear, it is intuitively apparent that regularity will be preserved through one-step rewriting; however even linearity is not sufficient to make one step of completion yield the expected result. Consider Genet's example 116, of the very simple linear TRS $\mathcal{R} = \{\, f(x) \to g(x) \,\}$ and of the BUTA $\mathcal{A}$ such that

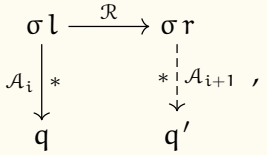
$$F = \{q\} \quad \text{and} \quad \Delta = \{\, a \to q, f(q) \to q \,\},$$

trivially recognising the regular language $\{\, a, f(a), f(f(a)), \dots \}$ which we abbreviate using the intuitively clear regular expression $f^*(a)$. We have $\mathcal{R}(f^*(a)) = f^*(g(f^*(a)))$, which is regular, and $\mathcal{R}^*(f^*(a)) = \{\, f, g \,\}^*(a)$, which is also regular – this is not surprising, as $\mathcal{R}$ happens to be right-linear and monadic. Let us apply one step of the completion algorithm on $\mathcal{A}$, yielding $\mathcal{A}'$: we have only one transition, one substitution $\{\, x \mapsto q \,\}$, and thus one critical pair, thus joined in $\mathcal{A}'$:



We obtain $A':\Delta = \{\, a \to q, f(q) \to q, g(q) \to q \,\}$, and then it is plain that our target has been largely overshot, as $\mathcal{L}(\mathcal{A}') = \mathcal{R}^*(f^*(a)) \supset \mathcal{R}(f^*(a))$. This being said, it can certainly be envisaged to modify the completion algorithm slightly in order to achieve our objectives; although this does not seem to have actually been done, the notion that it *can* be done fairly easily appears to be part of the folklore.

One-Step Completion. In order to accommodate one-step rewriting, we suggest to alter the joining operation along the lines of the following diagram:



The introduction of a new state is similar to [Genet & Rusu, 2010], for instance, although it serves another purpose here.

where q' is a fresh state. On top of that, for each pair (q, q'), A_{i+1} should also receive a copy of the transitions with q in the left-hand side, using the new state q' in its stead: that is to say we add the rules

$$\{\, q \mapsto q' \,\}\{\, f(p_1, \dots, p_n) \to p \in \mathcal{A}_i{:}\Delta \mid \exists i : p_i = q \,\}\,.$$

Furthermore, if q was final, it is replaced in that role by q': we have $\mathcal{A}_{i+1}{:}F = \{\, q \mapsto q' \,\}(\mathcal{A}_i{:}F)$. Let us try Genet's counter-example again, with this new algorithm; we obtain the transitions

$$\mathcal{A}'{:}\Delta = \left\{\, a \to q, f(q) \to q, g(q) \to q', f(q') \to q' \,\right\},$$

and this time $\mathcal{L}(\mathcal{A}') = f^*(g(f^*(a))) = \mathcal{R}(f^*(a))$. The idea is to proceed as before with the q-rules until the subterm where the rewriting applies is reached in q, then to apply the rewriting, reaching q', after which the construction goes on as before, with q' instead of q. But because of the asymmetry of rules such as $f(q) \to q'$, going from q to q', the rewriting can only be applied at most once in any term, and it has to be applied at least once, given that q' is the new final state. This can be iterated: a second step of completion would then yield the critical pairs

$$\begin{array}{ccc} f(q) & \xrightarrow{\mathcal{R}} & g(q) \\ {\scriptstyle \mathcal{A}'}\downarrow & & \vdots\,{\scriptstyle \mathcal{A}''} \\ q & & q'_2 \end{array} \quad \text{and} \quad \begin{array}{ccc} f(q') & \xrightarrow{\mathcal{R}} & g(q') \\ {\scriptstyle \mathcal{A}'}\downarrow & & \vdots\,{\scriptstyle \mathcal{A}''} \\ q' & & q'' \end{array} .$$

The first pair generates a dead duplicate of the rules of the first step; since q'_2 is not co-accessible, it can be ignored safely. The second critical pair yields the new rules $g(q') \to q''$ and $f(q'') \to q''$, thence the recognised language

$$\mathcal{L}\left(\mathcal{A}''\right) = \underbrace{f^*(g(\overbrace{f^*(g(\underbrace{f^*(a)}_{\mathcal{L}^q(\mathcal{A}'')}))}^{\mathcal{L}^{q'}(\mathcal{A}'')}))}_{\mathcal{L}^{q''}(\mathcal{A}'')} .$$

One needs to take more precautions to deal with less trivial transitions; consider for instance $h(q,q) \to q$, instead of $f(q) \to q$, and any rewrite rule whose left-hand side can match $h(q,q)$. Then the method outlined above would yield an additional rule $h(q',q') \to q'$, which would allow two separate applications of the rewrite rule. To prevent that, one has to generate instead $h(q',q) \to q'$ and $h(q,q') \to q$.

This kind of technique should suffice for exact computations, at least in the linear case. Of course, the above is only a sketch, more work is required to fully adapt the construction. The point of this exercise is that there is no reason to believe that one-step rewriting completion is fundamentally harder to achieve than forward-closure completion, nor that the techniques that contribute to the latter have no bearing on the former. Hence the dearth of literature on one-step rewriting should not dissuade us from building a model-checking framework which relies on the computation of the languages involved in such equations as $(3.1)_{[p42]}$, which do involve single-step rewriting.

3.2.5 The Importance of Being Left-Linear

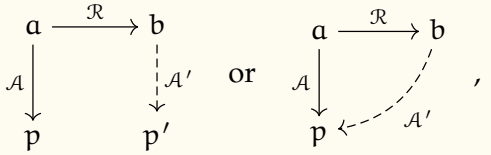
Linear systems should be covered by our adaptation of the completion, and yield exact results, as there is no need to resort to approximations to ensure termination – in that respect things are simpler for one-step than for forward-closure.

What about right-linear systems which are not also left-linear? The reason that left-linearity is a requirement for completion – whether one-step or not – is illustrated

by a transition $f(p, q) \to q'$ and a rewrite rule $f(x, x) \to g(x)$; there is simply no suitable substitution if $p \neq q$, and so the abstraction of subterms by states which underpins the completion breaks down. The usual solutions are discussed in [Genet, 2009, Sec. 4.4.1]; in a nutshell, the two main approaches are (1) the computation of intersections [Boichut, Héam & Kouchnarenko, 2006; Boichut, Courbis, Héam & Kouchnarenko, 2009], and (2) determinisation.

Indeed, rewriting occurs on terms of the form $f(u, u)$, with $u \in \mathcal{L}^p(\mathcal{A}) \cap \mathcal{L}^q(\mathcal{A})$. New rules can be computed and added, culminating in a fresh state $\hat{q}$ such that $\mathcal{L}^{\hat{q}}(\mathcal{A}) = \mathcal{L}^p(\mathcal{A}) \cap \mathcal{L}^q(\mathcal{A})$. Then $f(p, q) \to q'$ can be replaced with $f(\hat{q}, \hat{q}) \to q'$ for the purpose of determining substitutions and joining critical pairs, yielding the new rule $g(\hat{q}) \to q'$. This can be very expensive; since the joining rests on the computation of a product of automata of size $O(\|\mathcal{A}\|)$ each, the application of a non–left-linear rule results in a polynomial blowup, of degree bounded by the highest arity.

Another way to solve – or actually to *remove* – the problem is to determinise $\mathcal{A}$; then for any two states $p \neq q$, $\mathcal{L}^p(\mathcal{A}) \cap \mathcal{L}^q(\mathcal{A}) = \varnothing$, which obviates the need to deal with such configurations at all. This is also very expensive, because of the unavoidable exponential blowup associated with determinisation in the worst case [Comon et al., 2008]. Furthermore, the explosion compounds itself when several consecutive steps are needed: consider $\mathcal{A}$ with $F = \{p, q\}$, $\Delta = \{a \to p, b \to q\}$ and $\mathcal{R} = \{a \to b\}$. For one-step or forward-closure completion, we have the critical pair



yielding for one-step $\mathcal{A}'{:}\Delta = \{a \to p, b \to q, b \to p'\}$ and $\mathcal{A}'{:}F = \{p', q\}$, and $\mathcal{A}'{:}\Delta = \{a \to p, b \to q, b \to p\}$ for forward closure; but even though $\mathcal{A}$ is deterministic, in both cases $\mathcal{A}'$ no longer is. Thus iteration of the determinisation method leads to a blowup of the order of a tower of exponentials. A compromise is to maintain a weaker, local form of determinism, concerning only the specific states involved instead of the whole automata. Genet experimented with this method in the Timbuk tool, but found that the benefit was probably not worth the overhead incurred.

A particular case of non–left-linearity is that of rewrite rules of the form $f(x, x, y) \to g(y)$, or more generally $l \to r$ such that $\mathcal{V}(l) = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and $\mathcal{V}(r) \subseteq \{y_1, \ldots, y_m\}$, whose left-linearity is broken only by $x_1, \ldots, x_n$. In that case, it suffices to compute the intersection once to check that the rule can apply, i.e. the intersection is non-empty, and then those transitions can be discarded; $\sigma r \to q$ is then added as normal. It so happens that this case occurs fairly often; Genet mentions the general knowledge deduction rule of an intruder, in the context of a cryptographic protocol, and the XOR rule:

$$\mathit{decrypt}(\mathit{encrypt}(k, m), k) \to m, \qquad x \oplus x \to 0\,. \tag{3.3}$$

Both satisfy this pattern, and can therefore be employed efficiently. Of course, if all else fails, one can always resort to over-approximations, which can soundly be provided for any TRS in the case of forward-closure [Boichut et al., 2008], and this method could certainly be adapted for one-step rewriting.

3.2.6 One-Step Rewriting, and Constraints

However, the exact computation of one-step rewriting is always possible, provided that one is willing to go beyond regular languages, and employ a more powerful class of automata. Indeed, the classes of automata with constraints were introduced specifically to deal with the non-linearity problems evoked above – see Chapter 5[p107] for a survey of such classes.

Specifically, we shall focus in this thesis on tree automata with global equality constraints, which were already defined in section 2.5[p35]. Indeed, they sport a number of properties which make them suitable for our needs, as shown in [Courbis et al., 2009, Prp. 5, 7 & 6]. Namely, for any rewrite system $\mathcal{R}$ and regular tree language Π, and TAGE-definable language $\Pi^=$,

(1) $\mathcal{R}^{-1}(\mathcal{T})$ is recognised by an RTA – a TA if $\mathcal{R}$ is left-linear,

(2) $\mathcal{R}(\Pi)$ is recognised by a TAGE, and

(3) whether $\mathcal{R}(\Pi^=) = \varnothing$ is testable in ExpTime.

Note that **(1)** $\Rightarrow$ **(3)**: $\mathcal{R}(\Pi^=) = \varnothing \Leftrightarrow \Pi^= \cap \mathcal{R}^{-1}(\mathcal{T}) = \varnothing$.

Thus we can compute one step of rewriting exactly, even if the rewrite system satisfies none of the required linearity conditions, and if the ultimate purpose is an emptiness test – as it often is – this is brought to two exact steps: $X(Y(\Pi)) = \varnothing$ is decidable in exponential time, whatever the properties of $X, Y \subseteq \mathcal{R}$. Further steps can be dealt with under the restrictions outlined above.

We shall come back to those considerations at the end of the next chapter; meanwhile, most of the discussion focusses on obtaining the language equations which translate the desired temporal properties.

Chapter 4
Approximating LTL on Rewrite Sequences

Contents

—Where things are translated into other things and something else is checked.

From a temporal property to a rewrite proposition, and therefrom to a positive approximated procedure. Such is the progression that undergirds the model-checking framework which has been sketched at the beginning of the last chapter (3[p41]), and that we flesh out in this one. Schematically, one starts with three inputs: the term rewriting system $\mathcal{R}$, the initial tree language Π, which we assume to be regular, and the temporal property φ that must be checked. In the first step, correctness of the system with respect to the specification φ is translated into a rewrite proposition π which is, in the second step, translated into a positive approximated procedure δ based upon tree automata with and without constraints – or potentially several such procedures $\delta_1, \ldots, \delta_n$, as there may be different, incomparable ways of performing the required approximations.

Rewrite Proposition

This is what we call the "language equations" such as (3.1)[p42], because they are not really equations, but formulæ of propositional logic whose atoms are comparisons between languages obtained through rewriting. A precise definition appears in the next section.

This approach, inspired by [Genet & Klay, 2000]'s method for the analysis of cryptographic protocols, was first proposed in [Courbis et al., 2009], where both

translation steps were performed and proven manually on three specific formulæ of linear temporal logic, chosen for their relevance to model-checking, in particular with respect to the security of Java MIDLets and in the context of the French ANR RAVAJ project. Our objective is to generalise that work to a fragment of LTL; that is to say, both steps of the translation must be mechanised in order to obtain a working, automatic verification framework. The main result of [Courbis et al., 2009] is the following trio of translations into rewrite propositions:

ANR RAVAJ Project

http://www.irisa.fr/celtique/genet/RAVAJ

$$\mathcal{R}, \Pi \models \Box(X \Rightarrow \bullet Y) \\ \Leftrightarrow [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \ \wedge \ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \,, \tag{4.1}$$

$$\mathcal{R}, \Pi \models \neg Y \ \wedge \ \Box(\bullet Y \Rightarrow X) \\ \Leftrightarrow Y(\Pi) = \varnothing \ \wedge \ Y([\mathcal{R} \setminus X](\mathcal{R}^*(\Pi))) = \varnothing \,, \tag{4.2}$$

$$\mathcal{R}, \Pi \models \Box(X \Rightarrow \circ \Box \neg Y) \\ \Leftrightarrow Y(\mathcal{R}^*(X(\mathcal{R}^*(\Pi)))) = \varnothing \,. \tag{4.3}$$

The paper also provides the general form of three corresponding positive approximated procedures; for instance, (4.1) is shown to be positively approximated by the conjunction of the procedures

$$\texttt{IsEmpty}(\texttt{OneStep}(\mathcal{R} \setminus Y, \texttt{Approx}(\mathcal{A}, \mathcal{R})), X)$$

and

$$\texttt{Subset}(\texttt{OneStep}(X, \texttt{Approx}(\mathcal{A}, \mathcal{R})), \texttt{Backward}(Y)) \,,$$

where $\mathcal{L}(\mathcal{A}) = \Pi$, $\texttt{Approx}(\mathcal{A}, \mathcal{R})$ is the completion algorithm, yielding another tree automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, `OneStep` and `Backward` perform exact one-step rewriting and backwards rewriting, yielding TAGE, as seen in section 3.2.6[p51], `IsEmpty`$(\mathcal{A}, X)$ is the emptiness test for $X(\mathcal{L}(\mathcal{A}))$, where $\mathcal{A}$ is a TAGE, `SubSet` is any containment test for TA, which suffices here because of the additional precondition that Y must be left-linear. Note that, syntax notwithstanding, this is almost a straightforward reformulation of the original rewrite proposition – although there are a few subtleties, which we leave aside for the moment.

Contrast this to the first step (4.1), of translation into a rewrite proposition. Looking at the three examples, the general shape of the temporal formulæ is obviously not preserved by this transformation. Nor should one expect it to be; it is not the syntax of the temporal formula that is being translated, but the semantics of the fact that the system abides by the temporal property expressed by said formula. Therefore, in order to effect or even discuss such a translation, we need to start by clearly defining the semantics of our brand of LTL. If we were working on infinite words, there would be no question about that – there is just one generally agreed-upon way to define LTL semantics on infinite words, – but this is not the case: the system $\mathcal{R}$ is not required to be terminating; indeed this is a valuable characteristic, as we may well endeavour to check reactive systems, which are required to have non-terminating behaviours. Hence the need to accommodate both terminating and non-terminating executions in the semantics. There are several ways to go about that; the next section presents out choice in this matter.

Let us come back to the second step, translation from rewrite proposition to positive approximated procedures. The reason why [Courbis et al., 2009] proposes

a linearity condition is to make the inclusion test $X(\mathcal{R}^{(*)}(\Pi)) \subseteq Y^{-1}(\mathcal{T})$ decidable. With a left-linear Y, $Y^{-1}(\mathcal{T})$ is regular, and the test can be rephrased as $X(\mathcal{R}^{(*)}(\Pi)) \cap (\mathcal{T} \setminus Y^{-1}(\mathcal{T})) = \emptyset$. Concretely, the TA $\texttt{Backward}(Y)$ is complemented, and its product with the TAGE $\texttt{OneStep}(X, \texttt{Approx}(\mathcal{A}, \mathcal{R}))$ representing the over-approximation $X(\mathcal{R}^{(*)}(\Pi))$ of $X(\mathcal{R}^{*}(\Pi))$ is computed, and tested for emptiness. If Y is not left-linear, then $\texttt{Backward}(Y)$ is a TAGE, and containment becomes undecidable – note that TAGE cannot be complemented; thus one then needs an extra layer of approximation. A third way to go about that would be to compute a regular under-approximation of $Y^{-1}(\mathcal{T})$. There are also different ways in which the left-hand side expression can be handled: instead of computing an exact TAGE, one could over-approximate the rewriting by X in $X(\mathcal{R}^{(*)}(\Pi))$, so as to get a standard tree automaton – although doing so is useless in this particular case.

As in section 1.2[p13], $\mathcal{R}^{(*)}(\Pi)$ represents a regular over-approximation of $\mathcal{R}^{*}(\Pi)$; we shall eventually use this notation in preference to the pseudo-code of the current discussion.

This goes to show that, even under the simplifying assumption that there is only one way to perform an over- or under-approximation, there are still several valid positive approximated procedures for all but the most trivial expressions. Some of those are blatantly worse than others – for instance, any procedure performing the double approximation of $X(\mathcal{R}^{(*)}(\Pi))$ will necessarily be coarser than a procedure performing a single approximation but otherwise identical. The double approximation does not improve decidability at all, as the right-hand side is the limiting factor here. Later in this chapter, we shall dismiss such dominated procedures, however let us state here that in practice, it may drastically improve tractability by diminishing the number of constraints in the product automaton; see Chapter 6[p117] on that subject.

Organisation of the chapter.

- Section 4.1 presents the notions and notations in use throughout this chapter, including the choice of temporal semantics, the definition of rewrite propositions, and precise statements of the problems at hand.
- Section 4.2[p59] presents an intuition of both the manner in which and the extent to which the translation into a rewrite proposition may be effected. This intuition provides the building blocks of our framework, which need to be formalised. The first such block, developed in section 4.2.3, is the notion of *signatures*, which we use to "flatten" a certain fragment of LTL formulæ upon the time-line; they are an integral part of our method, as they enable us to keep track of the languages reached at different points in time, and of the existence or non-existence of certain transitions.
- Section 4.3[p73] relies on signatures to provide a set of translation rules that perform the translation into rewrite propositions for a fragment of LTL. The process yields a derivation tree that shows correctness of the translation. This takes care of the first step.
- Section 4.4[p87] focuses on the second part, namely the generation of positive approximated procedures. This is addressed by means of procedure generation rules producing all possible theorems. Although the discussion remains more abstract than an implementation would be, those rules form the general skeleton that an implementation should flesh out. In light of this, possibilities for optimising the generated rewrite propositions are discussed in section 4.4.2[p95].

◇ Section 4.5[p97] shows full derivations of both steps on the three usual example formulæ, and examines whether and how well the developed methods apply to common temporal properties and existing TRS models in various domains. In particular, we use surveys of popular temporal patterns and existing TRS encodings as touchstones of the potential usefulness of our methods.

4.1 Preliminaries & Problem Statement

In order to offer a precise definition of the semantics of our temporal logic, we first need to establish the kind of words upon which it is based. As mentioned before, there is a need to accommodate both terminating and non-terminating behaviours; furthermore, it is not an aspect that needs to be hidden or abstracted away. We want to be able to express properties regarding the presence or absence of a next transition succinctly and naturally. For this reason, we work directly on words which may be infinite or finite, including the empty word, as the system may well do nothing at all.

$\mathbb{A}$ is the only alphabet in this entire chapter, and so it is kept implicit; for instance we write the set of terms $\mathcal{T}$ instead of $\mathcal{T}(\mathbb{A})$, etcetera. The same applies to constructs depending on $\mathcal{R}$, as it is also unequivocal throughout the chapter.

4.1.1 Rewrite Words & Maximal Rewrite Words

$\mathcal{W}$: *words on $\mathcal{R}$, finite or infinite*

Let $\mathbb{A}$ be a ranked alphabet, $\mathcal{R}$ a finite rewrite system, and $\Pi \subseteq \mathcal{T}$ any set of terms. A *finite or infinite word on* $\mathcal{R}$ is an element of

$$\mathcal{W} = \bigcup_{n \in \overline{\mathbb{N}}} \left([\![1, n]\!] \to \mathcal{R} \right) .$$

$\#w$: *length of (in)finite word*

The length $\#w \in \overline{\mathbb{N}}$ of a word w is defined as $\mathrm{Card}(\mathrm{dom}\, w)$. Note that the empty function – of graph $\varnothing \times \mathcal{R} = \varnothing$ – is a word, which we call the *empty word*, denoted by λ. Let $w \in \mathcal{W}$ be a word of domain $[\![1, n]\!]$, for $n \in \overline{\mathbb{N}}$, and let $m \in \mathbb{N}_1$; then the m-*suffix of* w is the word denoted by w^m, such that

w^m: *suffix of w, of rank m*

$$w^m = \left| \begin{array}{ccc} [\![1, n-m+1]\!] & \longrightarrow & \mathcal{R} \\ k & \longmapsto & w(k+m-1) \end{array} \right. .$$

Suffix Cheat Sheet

$$\begin{aligned} w^{1+m}(k) &= w(k+m) , \\ &= w^{1+k}(m) , \\ w^{\#w} &= w(\#w) , \\ w^{k+\#w} &= \lambda, \ \forall k > 0 , \\ \#w^{1+m} &= |\#w - m|_0 , \\ (w^i)^{1+m} &= w^{i+m} . \end{aligned}$$

Note that $w^1 = w$, for any word w. The intuitive meaning that we attach to a word w is a sequence of rewrite rules of $\mathcal{R}$, called in succession – in other words, it represents a "run" of the TRS $\mathcal{R}$. Of course, there is nothing in the above definition of words that guarantees that such a sequence is in any way feasible, and such a notion only makes sense with respect to initial terms to be rewritten. Thus we now define the *maximal rewrite words of $\mathcal{R}$, originating in* Π:

$(\!|\Pi|\!)$: *maximal rewrite words of $\mathcal{R}$, originating in Π*

$$(\!|\Pi|\!) = \left\{ w \in \mathcal{W} \;\middle|\; \begin{array}{l} \exists u_0 \in \Pi : \exists u_1, \ldots, u_{\#w} \in \mathcal{T} : \forall k \in \mathrm{dom}\, w, \\ u_{k-1} \xrightarrow{w(k)} u_k \ \wedge \ \#w \in \mathbb{N} \Rightarrow \mathcal{R}(\{u_{\#w}\}) = \varnothing \end{array} \right\} .$$

Note the potential presence of the empty word in that set. Informally, a word w is in $(\!|\Pi|\!)$ if and only if the rewrite rules $w(1), \ldots, w(n), \ldots$ can be activated in succession, starting from a term $u_0 \in \Pi$, and the word w is "maximal" in the sense that it cannot be extended. That is to say, w ends only when no further rewrite rule can be activated. Thus $(\!|\Pi|\!)$ captures the behaviours (or runs) of $\mathcal{R}$, starting from Π;

this notion is equivalent the full paths of the rewrite graph described in [Courbis et al., 2009], and corresponds to the usual maximal trace semantics [Cousot, 2002], with a focus on transitions instead of states.

4.1.2 Defining Temporal Semantics on Rewrite Words

Choice of LTL & Syntax. Before starting to think about translating temporal logic formulæ on rewrite words, we need to define precisely the kind of temporal formulæ under consideration, and their semantics. Given that prior work in [Courbis et al., 2009] was done on LTL, and that our aim is to generalise this work, LTL – with subsets of $\mathcal{R}$ as atomic propositions – seems a reasonable choice. In practice we shall use a slight variant with generalised weak and strong next operators; the reasons for this choice will be discussed when the semantics are examined. A formula $\varphi \in \text{LTL}$ is generated by the following grammar:

LTL on finite and infinite words

$$\begin{aligned} \varphi &:= X \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bullet^m \varphi \mid \circ^m \varphi \mid \varphi \,\mathbf{U}\, \varphi && X \in \wp(\mathcal{R}) \\ &\quad \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \Diamond\, \varphi \mid \Box\, \varphi && m \in \mathbb{N}\,. \end{aligned}$$

Note that the operators which appear on the first line are functionally complete; the remaining operators are defined syntactically as: $\top = \mathcal{R} \vee \neg\mathcal{R}$, $\bot = \neg\top$, $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, $\Diamond\, \varphi = \top \,\mathbf{U}\, \varphi$ and $\Box\, \varphi = \neg \Diamond \neg\varphi$.

Choice of Semantics. In the literature, the semantics of LTL are defined and well-understood for ω-words; however the words of $(\!|\Pi|\!)$ may be finite – even empty – or infinite, which corresponds to the fact that, depending on its input, a rewrite system may either not terminate, terminate after some rewrite operations, or terminate immediately. Therefore we need semantics capable of accommodating both ω-words and finite words, including the edge-case of the empty word. In contrast to the classical case of ω-words, there are several ways to define (two-valued) semantics for LTL on finite, maximal words. One such way found in the literature is *Finite-LTL* [F-LTL, Manna & Pnueli, 1995], which complements the long-standing use of a *strong* next operator introduced in [Kamp, 1968] by coining a *weak next* variant. Figure 4.1[p58] presents our choice of semantics for this chapter, which is essentially F-LTL with generalised next operators and the added twist that words may be infinite or empty. Note that $\bullet^1$ and $\circ^1$ correspond exactly to the classical strong and weak next operators, and that for $m \geqslant 1$, $\bullet^m$ (resp. $\circ^m$) can trivially be obtained by repeating $\bullet^1$ (resp. $\circ^1$) m times. So the only non-trivial difference here is the existence of $\bullet^0$ and $\circ^0$; this will prove quite convenient when we deal with the translation of $\Box$, using the following lemma.

Finite-LTL

$\bullet^m$: strong next operator

$\circ^m$: weak next operator

∀ Lemma 4.1: Weak-Next & Always

Let $\varphi \in \text{LTL}$, $w \in \mathcal{W}$, $k \in \mathbb{N}$ and $i \in \mathbb{N}_1$; it holds that **(1)** $(w, i) \models \Box\, \varphi$ iff $(w, i) \models \bigwedge_{m=0}^{\infty} \circ^m \varphi$ and **(2)** $(w, i) \models \Box\, \varphi$ iff $(w, i) \models \bigwedge_{m=0}^{k-1} (\circ^m \varphi) \wedge \circ^k \Box\, \varphi$.

Short Proof. **(1)** $(w, i) \models \bigwedge_{m=0}^{\infty} \circ^m \varphi \iff \bigwedge_{m=0}^{\infty} (w, i) \models \circ^m \varphi \iff \forall j \in \operatorname{dom} w,\ j \geqslant i \Rightarrow (w, j) \models \varphi \iff (w, i) \models \Box\, \varphi$. **(2)** $(w, i) \models \bigwedge_{m=0}^{\infty} \circ^m \varphi \iff (w, i) \models \bigwedge_{m=0}^{k-1} (\circ^m \varphi) \wedge \bigwedge_{m=k}^{\infty} (\circ^m \varphi)(w, i + k) \models \Box\, \varphi \iff (w, i) \models \bigwedge_{m=0}^{k-1} (\circ^m \varphi) \wedge \circ^k \Box\, \varphi$. ∎

$$
\begin{array}{ll}
(w,i) \models X & \text{iff} \quad i \in \operatorname{dom} w \text{ and } w(i) \in X \\
(w,i) \models \neg\varphi & \text{iff} \quad (w,i) \not\models \varphi \\
(w,i) \models (\varphi \wedge \psi) & \text{iff} \quad (w,i) \models \varphi \text{ and } (w,i) \models \psi \\
(w,i) \models \bullet^m \varphi & \text{iff} \quad i+m \in \operatorname{dom} w \text{ and } (w,i+m) \models \varphi \\
(w,i) \models \circ^m \varphi & \text{iff} \quad i+m \notin \operatorname{dom} w \text{ or } (w,i+m) \models \varphi \\
(w,i) \models \varphi \,\mathbf{U}\, \psi & \text{iff} \quad \exists j \in \operatorname{dom} w : j \geqslant i \wedge \begin{cases} (w,j) \models \psi \quad \wedge \\ \forall k \in [\![i, j-1]\!],\ (w,k) \models \varphi \end{cases} \\
(w,i) \models \top & (w,i) \not\models \bot \\
(w,i) \models \neg X & \text{iff} \quad i \notin \operatorname{dom} w \text{ or } w(i) \notin X \\
(w,i) \models (\varphi \vee \psi) & \text{iff} \quad (w,i) \models \varphi \text{ or } (w,i) \models \psi \\
(w,i) \models (\varphi \Rightarrow \psi) & \text{iff} \quad (w,i) \models \varphi \Rightarrow (w,i) \models \psi \\
(w,i) \models \Diamond\, \varphi & \text{iff} \quad \exists j \in \operatorname{dom} w : j \geqslant i \wedge (w,j) \models \varphi \\
(w,i) \models \Box\, \varphi & \text{iff} \quad \forall j \in \operatorname{dom} w,\ j \geqslant i \Rightarrow (w,j) \models \varphi
\end{array}
$$

For any $w \in \mathcal{W}$, $i \in \mathbb{N}_1$, $m \in \mathbb{N}$ *and* $X \in \wp(\mathcal{R})$.

Figure 4.1: LTL semantics on maximal rewrite words.

Before moving on, let us stress that the choice of semantics, or even the choice of LTL for that matter, should by no means be considered as etched in stone; it is very much a variable of the general problem. However it will henceforth be considered as data for the purposes of this chapter.

TRS & LTL. Let φ be an LTL formula. It is said that a word w satisfies/is a model of φ (denoted by $w \models \varphi$) iff $(w,1) \models \varphi$. Alternatively, we have $(w,i) \models \varphi$ iff $w^i \models \varphi$. We say that the rewrite system $\mathcal{R}$, with initial language Π, satisfies/is a model of φ (denoted by $\mathcal{R}, \Pi \models \varphi$) iff $\forall w \in (\!|\Pi|\!),\ w \models \varphi$.

$\mathcal{R}, \Pi \models \varphi$: *satisfaction of the specification* φ

4.1.3 Rewrite Propositions & Problem Statement

rewrite proposition

A *rewrite proposition on* $\mathcal{R}$, *from* Π is a formula of propositional logic whose atoms are language or rewrite systems comparisons. More specifically, a rewrite proposition π is generated by the following grammar:

$$
\begin{array}{lll}
\pi := \gamma \mid \gamma \wedge \gamma \mid \gamma \vee \gamma & \gamma := \ell = \varnothing \mid \ell \subseteq \ell & X \in \wp(\mathcal{R})\,. \\
& \ell := \Pi \mid \mathcal{T} \mid X(\ell) \mid X^{-1}(\ell) \mid X^*(\ell) &
\end{array}
$$

Since the comparisons γ have obvious truth values, the interpretation of rewrite propositions is trivial; thus we shall not introduce any notation for it, and automatically confuse π with its truth value in the remainder of this chapter. Note that while other operators for propositional logic could be added, conjunction and disjunction will be enough for our purposes.

Problem Statements. The overarching goal is a systematic method to decide or approximate whether $\mathcal{R}, \Pi \models \varphi$, given a rewrite system $\mathcal{R}$, a temporal formula φ in LTL – or some fragment of LTL– and an initial language $\Pi \subseteq \mathcal{T}$. This goal is broken down into two distinct sub-problems:

(1) Finding an algorithmic method for building, from φ, a rewrite proposition π such that $\mathcal{R}, \Pi \models \varphi$ if and only if π holds. We call such a method, as well as its result, an *exact translation* of φ, and say that π translates φ.

exact translation

(2) Finding an algorithm for generating, from π, a positive approximated procedure δ, that answers positively only if π holds, or a full decision procedure, whenever possible.

By solving both sub-problems, one has $\delta \implies \pi$ and $\pi \iff \mathcal{R}, \Pi \models \varphi$, and therefore $\delta \implies \mathcal{R}, \Pi \models \varphi$, which achieves the overall goal.

One notices that the full equivalence is not needed in $\pi \Leftrightarrow \mathcal{R}, \Pi \models \varphi$; if π is only a sufficient (resp. necessary) condition, then it is an *under-approximated* (resp. *over-approximated*) translation. Of course, only under-approximated translations hold any practical interest for our purposes. Although approximated translations are briefly discussed in a few places for the sake of completeness, regarding the first problem, we are interested only in exact translations. The reason for this are twofold: where exact translation is not achievable, we have not come across any interesting ways in which fine approximations may be introduced at this stage; and secondly, having several successive layers of approximations is likely a recipe for very coarse approximated procedure at the end of the day. Thus it seems advisable to handle approximations in the second step exclusively, and to keep the translation into rewrite propositions exact.

under-approximated translation
over-approximated translation

4.2 Technical Groundwork: Antecedent Signatures

The first problem is tackled by two complementary tools: *signatures*, which are developed in this section, and *translation rules*, which rely heavily on signatures and are the object of Sec. 4.3[p73]. The beginning of the present section also serves as an intuitive introduction to the first problem, and as an a priori discussion of the scope of the translation.

4.2.1 Overview & Intuitions

The Base Cases. Counterintuitively, $\varphi = \neg X$ is actually a simpler case than $\varphi = X$ as far as the translation is concerned, so it will be considered first.

Case 1: Negative Literal. Suppose $\mathcal{R}, \Pi \models \neg X$. Recalling the semantics in Fig. 4.1[p58], this means that no term of Π can be rewritten by a rule in X. They may or may not be rewritable by rules not in X, though. Consider now

$$\pi_1 \equiv X(\Pi) = \varnothing \; ; \tag{π_1}$$

it is easy to become convinced that this is an exact translation.

Case 2: Positive Literal. Let $\varphi = X$. A first intuition would be that this is roughly the same case as before, but with the complement of X wrt. $\mathcal{R}$. So we write $\pi_2 \equiv [\mathcal{R} \setminus X](\Pi) = \varnothing$. This, however, is not strong enough. It translates the fact that only rules of X can rewrite Π. But again, while X may in fact rewrite Π, there is nothing in π_2 to enforce that. Looking at the semantics, all possible words of $(\!|\Pi|\!)$ must have at least one move (i.e. $1 \in \operatorname{dom} w$); this condition must be translated. It is equivalent to saying that all terms of Π are rewritable, which is expressed by $\Pi \subseteq \mathcal{R}^{-1}(\mathcal{T})$. More specifically, since we already impose that they are not rewritable

by $\mathcal{R} \setminus X$, we can even write directly that they are rewritable by X, i.e. $\Pi \subseteq X^{-1}(\mathcal{T})$. Putting those two conditions together, we obtain

$$\pi_2' \equiv [\mathcal{R} \setminus X](\Pi) = \varnothing \ \wedge \ \Pi \subseteq X^{-1}(\mathcal{T}) \ , \tag{π_2'}$$

and this is an exact translation.

Of Strength & Weakness. Let us reflect on the previous cases for a minute; the immediate intuition is that X is *stronger* than $\neg X$, in the sense that whenever we see X, we must write an additional clause – enforcing rewritability – compared to $\neg X$. This actually depends on the context, as the next example will show.

CASE 3: ALWAYS NEGATIVE. Let $\varphi = \Box \neg X$. This means that neither the terms of Π nor their successors can be rewritten by X; in other words $\pi_3 \equiv X(\mathcal{R}^*(\Pi)) = \varnothing$. The translation is almost the same as for $\neg X$, the only difference being the use of $\mathcal{R}^*(\Pi)$ (Π and successors) instead of just Π as in π_1. More formally,

$$\pi_3 \equiv X(\mathcal{R}^*(\Pi)) = \varnothing \equiv \pi_1[\mathcal{R}^*(\Pi)/\Pi] \ . \tag{π_3}$$

CASE 4: ALWAYS POSITIVE. Seeing this, one is tempted to infer that the same relationship that exists between the translations of $\neg X$ and $\Box \neg X$ exists as well between those of X and $\Box X$. In the case $\varphi = \Box X$, this would yield $\pi_4 \equiv \pi_2'[\mathcal{R}^*(\Pi)/\Pi] \equiv [\mathcal{R} \setminus X](\mathcal{R}^*(\Pi)) = \varnothing \wedge \mathcal{R}^*(\Pi) \subseteq X^{-1}(\mathcal{T})$. But clearly this translation is much too strong as its second part implies that every term of Π can be rewritten by X, and so can all of the successors; consequently, $(\!|\Pi|\!)$ must form an ω-language. Yet we have for instance $\lambda \models \Box X$ —note incidentally that $\lambda \models \Box \psi$ holds vacuously for any ψ. In general, under the semantics for $\Box$, words of any length, infinite, finite or nought, may satisfy $\Box X$. Thus the correct translation was simply

$$\pi_4' \equiv [\mathcal{R} \setminus X](\mathcal{R}^*(\Pi)) = \varnothing \ . \tag{π_4'}$$

So, unlike Cases 1 and 2, X is not in any sense stronger than $\neg X$ when behind a $\Box$. This is an important point which we shall need to keep track of during the translation; that necessary bookkeeping will be done by means of the *signatures* introduced in Sec. 4.2.3[p62].

Conjunction, Disjunction & Negation. CASE 5: AND & OR. It is pretty clear that if π_5 translates φ and π_5' translates ψ, then $\pi_5 \wedge \pi_5'$ translates $\varphi \wedge \psi$. This holds thanks to the implicit universal quantifier, as we have $(\mathcal{R}, \Pi \models \varphi \wedge \psi) \iff (\mathcal{R}, \Pi \models \varphi) \wedge (\mathcal{R}, \Pi \models \psi)$. Contrariwise, the same does not hold for the disjunction, and we have no general solution[(a)] to handle it. Given that one of the implications still holds, namely $(\mathcal{R}, \Pi \models \varphi \vee \psi) \impliedby (\mathcal{R}, \Pi \models \varphi) \vee (\mathcal{R}, \Pi \models \psi)$, a crude under-approximation can still be given if all else fails:

$$\pi_5 \vee \pi_5' \implies \mathcal{R}, \Pi \models \varphi \vee \psi \ . \tag{π_5''}$$

CASE 6: NEGATION. Although we have seen in Case 1 that a negative literal can easily be translated, negation cannot be handled in all generality by our method. Note that, because of the universal quantification, $\mathcal{R}, \Pi \not\models \varphi \neq \mathcal{R}, \Pi \models \neg\varphi$; thus the fact that π_6 translates φ does not a priori imply that $\neg\pi_6$ translates $\neg\varphi$. This is why we shall assume in practice that input formulæ are provided in a sanitised form,

[(a)] There are however special cases where disjunction can be translated exactly; see rules $(\vee^{\Rightarrow}_{\wedge})$[p75] and $(\vee^{\neg}_{\Rightarrow})$.

where negations only appear on literals. The presence of both weak and strong next operators facilitates this, as $\neg \circ^m \varphi \Leftrightarrow \bullet^m \neg\varphi$. Note that this is not exactly the same as requiring a Negative Normal Form (NNF), as implications remain allowed as well as disjunctions, and strictly speaking there is no convergence towards a normal form. More details are given in Sec. 4.2.2[p61].

Handling Material Implication. Case 7: Implication. We have just seen in Cases 5 and 6 that we can provide exact translations for neither negation nor disjunction. Inasmuch as $\varphi \Rightarrow \psi$ is defined as $\neg\varphi \vee \psi$, must material implication be forgone as well? An example involving an implication has been given in the introduction – page 53, – so it would seem that a translation can be provided in at least some cases. Let us take the simple example $X \Rightarrow \bullet Y$. Assuming that any term $u \in \Pi$ is rewritten into some u' by a rule in X, then u' must be rewritable by Y, and only by Y. The set of X-successors of Π being $X(\Pi)$, those conditions yield the translation

$$\pi_7 \equiv [\mathcal{R} \setminus Y](X(\Pi)) = \varnothing \ \wedge \ X(\Pi) \subseteq Y^{-1}(\mathcal{T}) \ . \tag{π_7}$$

Note that the way in which implication has been handled here is very different from the approach taken for the other binary operators, which essentially consists in splitting the formula around the operator and translating the two subparts separately. In contrast, the antecedent of the implication was "assumed", whilst the consequent was translated as usual. In fact, recalling that π'_2 translates X, and thus $\pi''_2 \equiv \pi'_2[Y/X]$ translates Y, we have $\pi_7 \equiv \pi''_2[X(\Pi)/\Pi]$. So, "assuming" the antecedent consisted simply in changing our set of reachable terms – which we shall from now on call the *past*, hence the notation Π. This is not an isolated observation; if π_0 denotes the translation (4.1)[p54] of $\Box(X \Rightarrow \bullet Y)$ given in the introduction,

$$\pi_0 \equiv [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \ \wedge \ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \ ,$$

then $\pi_0 \equiv \pi_7[\mathcal{R}^*(\Pi)/\Pi]$. Thus "updating" the past is enough of a tool to deal with some simple uses of $\Box$ and implication... but consider the following formula: $\bullet Y \Rightarrow X$. In that case the antecedent lies in the future, relatively to the consequent. Therefore, in order to deal with all cases, we need some means of making assumptions about both past and future. This is the goal of the *signatures* presented in Sec. 4.2.3[p62]. Examples of translations where the antecedent is in the future appear at the very end of Sec. 4.3[p73].

4.2.2 Choosing a Suitable Fragment of LTL

As mentioned in Cases 3 and 4 of the previous section, negation of complex formulæ is problematic, and we shall therefore work only with formulæ in a sanitised form, such that negation appears only on literals, that differs from traditional NNF in that implication remains allowed – and thus it is not a normal form at all. For instance, $(A \vee B) \Rightarrow C$, $(A \Rightarrow C) \wedge (B \Rightarrow C)$, and $(\neg A \wedge \neg B) \vee C$ are three equivalent formulæ, all sanitised. However, the first and second forms will be favoured over the third (the NNF), because those forms fit into the translation system presented hereafter. Some transformations of particular relevance are included into the translation rules themselves, in Sec. 4.3[p73], but this is essentially a straightforward preprocessing step, which we shall not detail.

Furthermore, there are operators – such as $\Diamond$ – for which it seems that no exact translation can be provided, since rewrite propositions are not expressive enough;

in particular, $\mathcal{R}^*(\Pi)$ hides all information regarding finite or infinite traces. Hence, none of the operators of the "Until" family $\{\Diamond, \mathbf{U}, \mathbf{W}, \mathbf{R}, \ldots\}$ can be translated exactly, and, as we are primarily concerned about exact translations, they will not be brought to play in what follows. Accordingly, we shall work chiefly within the following fragment of LTL, which will be denoted by $\mathcal{R}$-LTL:

$\mathcal{R}$-LTL: *Rewrite LTL*

$$\begin{aligned} \varphi := X \mid \neg X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid & \qquad X \in \wp(\mathcal{R}) \\ \bullet^m \varphi \mid \circ^m \varphi \mid \Box \varphi & \qquad m \in \mathbb{N}\,. \end{aligned}$$

4.2.3 Girdling the Future: Signatures

As discussed in Sec. 4.2.1[p59], Case 7, implication is handled by converting the antecedent φ of a formula $\varphi \Rightarrow \psi$ into "assumptions". Concretely, this consists in building a model of φ – called a *signature* of φ, written $\xi(\varphi)$ – which can be manipulated during the translation. This model will also be used to store sufficient information regarding the context in order to determine whether the translation ought to be "strong" or "weak", as sketched in Case 4.

$\xi(\varphi)$: *signature of* φ

The variety of signatures defined hereafter handles formulæ φ within the fragment $\mathcal{A}$-LTL ($\mathcal{A}$ for **a**ntecedent), which is $\mathcal{R}$-LTL without $\vee$ or $\Rightarrow$. The reasons for and consequences of this restriction will become clearer in Sec. 4.3[p73]; suffice it to say that handling $\vee$ and right-associative $\Rightarrow$ chains at the level of signatures is simply unnecessary, as they are easily dealt with through trivial transformations (cf. rules $(\vee^{\Rightarrow}_{\wedge})$[p75] and $(\Rightarrow_{\Sigma})$). Left-associative $\Rightarrow$ chains should be reformulated and generally entail an approximated translation.

$\mathcal{A}$-LTL: *Antecedent LTL*

This section defines signatures formally and presents a suitable map $\xi(\cdot) : \mathcal{A}\text{-LTL} \to \Sigma$, Σ being the space of signatures, whose correctness proof is broken down into eight main lemmata, and finally summarised by Thm. 4.15[p72].

The *informal* idea behind our definition of signatures is to capture information regarding the possible successive rewriting steps from the current language Π – our *past*, as we called it in Sec. 4.2.1[p59], Case 7. The empty signature encodes no information at all: all possibilities remain open. That is to say, starting from $t \in \Pi$, there may or may not be a rewriting transition $t \xrightarrow{r} t'$ and even if there is all that can be said at this point is that $r \in \mathcal{R}$; moreover, no further information is available about t' and its possible successors. But as more information is gained from antecedents, constraints will be added to the signature. Suppose that we are faced with a formula of the form

$$X \wedge \circ^1 Y \wedge \circ^2 \Box Z \implies \varphi\,,$$

then we only need to worry about translating φ assuming that no rewrite steps are taken (or not taken) in contradiction to the antecedent. In other words, for the purpose of translating φ, we assume that there is a rewriting transition from Π, and that it is by X, i.e. $\forall t \in \Pi,\ \exists r^t_1 \in X : t \xrightarrow{r^t_1} t_1$. Furthermore, we cannot directly assume that there is then a second transition, this time by Y, because "$\circ^1 Y$" does not imply existence; however it can be assumed that if there is a second transition, it must be activated by rules in Y. The last part of the antecedent, "$\circ^2 \Box Z$", tells us that, starting from the third transition, all rules involved must be within Z; but again there is no guarantee that any such transition exists. Put schematically, the

information gathered from this antecedent looks like this: starting from Π, we have successively transitions by X (exists), Y (maybe), Z (maybe), Z (maybe),... Once we have introduced the formal and notational apparatus, this information will be denoted by

$$\xi\big(X \wedge \circ^1 Y \wedge \circ^2 \Box Z\big) \;=\; \wr X, Y \fatsemi Z \mid \overline{\mathbb{N}}_1 \int .$$

The intuitive meaning of X, Y and Z in this formula should be relatively transparent at this point; the second component, $\overline{\mathbb{N}}_1$, encodes the length of the maximal rewrite words compatible with this signature. In this case, since we know that there must be a X-transition, the empty word λ – of length 0 – is not compatible. On the other hand, we have no further information about the potential existence of other transitions, so provided that a rewrite word w follows the progression X, Y, Z, Z, Z, ..., it is compatible as soon as its length is 1 or more; that is to say, as soon as $\#w \in \overline{\mathbb{N}}_1$. This notion of the compatibility of a rewrite word with a signature is what gives them precise semantics; it will be made explicit by the definition of *constrained words* below.

Signatures. A *signature* σ is an element of the space

$$\Sigma = \bigcup_{n \in \mathbb{N}} \Big[\big([\![1, n]\!] \cup \{\omega\} \big) \to \wp(\mathcal{R}) \Big] \times \wp(\overline{\mathbb{N}}) .$$

Σ: *set of signatures*

Core, Support, Domain, Cardinal. Let $\sigma = (f, S) \in \Sigma$; then the function f is called the *core of* σ, denoted by $\partial\sigma$, and S is called its *support*, written $\nabla\sigma$. The *domain of* σ is defined as $\operatorname{dom}\sigma = \operatorname{dom} f \setminus \{\omega\}$, and its *cardinal* is $\#\sigma = \operatorname{Card}(\operatorname{dom}\sigma)$.

$\partial\sigma$: *core of signature* σ
$\nabla\sigma$: *support of signature* σ
$\operatorname{dom}\sigma$: *domain of signature* σ
$\#\sigma$: *cardinal of signature* σ

Special Notations, Empty Signature. A signature $\sigma = (f, S)$ will be written either compactly as $\sigma = \wr f \mid S \int$, or *in extenso* as

$$\wr f(1), f(2), \ldots, f(\#\sigma) \fatsemi f(\omega) \mid S \int .$$

$\wr f \mid S \int$: *signature, compact*
$\wr f\,1 \ldots f\,\#\sigma \fatsemi f\,\omega \mid S \int$: *signature, in extenso*

Note that the example at the end of this string of definitions illustrates all this. A signature of special interest, which we denote by $\varepsilon = \wr \fatsemi \mathcal{R} \mid \overline{\mathbb{N}} \int$, is the *empty signature*. Let $k \in \mathbb{N}_1 \cup \{\omega\}$, then we write

$\varepsilon = \wr \fatsemi \mathcal{R} \mid \overline{\mathbb{N}} \int$: *empty signature*

$$\sigma[k] = \begin{cases} f(k) & \text{if } k \in \operatorname{dom}\sigma \\ f(\omega) & \text{if } k \notin \operatorname{dom}\sigma \end{cases} .$$

The notation $\sigma[k]$ is read "σ at (position) k" and is referred to as the *at operator*.

$\sigma[k]$: *signature "at" operator*

Constrained Words. The set of maximal rewrite words which satisfy the constraints encoded by a signature, as defined below, will be used over and over again throughout this section. It is this notion that assigns precise semantics to the signatures we build. The *maximal rewrite words of* $\mathcal{R}$*, originating in* Π *and constrained by* σ are defined by

$(\!|\Pi \fatsemi \sigma|\!)$: *words constrained by* σ

$$(\!|\Pi \fatsemi \sigma|\!) = \{ w \in (\!|\Pi|\!) \mid \#w \in \nabla\sigma \;\wedge\; \forall k \in \operatorname{dom} w,\; w(k) \in \sigma[k] \} .$$

At some later point in this chapter (Lem. 4.17[p78]), we shall start having to reason on the length of the words quite a bit, so for the sake of conciseness we write $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{m}$ the set $\{ w \in (\!|\Pi \fatsemi \sigma|\!) \mid \#w = m \}$, $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{S}$ the set $\{ w \in (\!|\Pi \fatsemi \sigma|\!) \mid \#w \in S \}$, and $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{\prec n}$ the set $\{ w \in (\!|\Pi \fatsemi \sigma|\!) \mid \#w \prec n \}$, for $(\prec) \in \{<, >, \leqslant, \geqslant\}$ and $n \in \overline{\mathbb{N}}$.

$(\!|\Pi \fatsemi \sigma|\!)^{\#}_{m}$: *words constrained by* σ*, of length* m

Example: Let $\sigma = \wr X, Y \S Z \mid \mathbb{N}_2 \wr$; then its core is the function $\partial\sigma = \{\, 1 \mapsto X, 2 \mapsto Y, \omega \mapsto Z \,\}$, its domain is $\operatorname{dom}\sigma = [\![1,2]\!]$, its support is $\nabla\sigma = \mathbb{N}_2$, its cardinal is $\#\sigma = 2$, and we have $\sigma[1] = X$, $\sigma[2] = Y$, $\sigma[3] = \sigma[4] = \cdots = \sigma[\omega] = Z$. Its constrained words are the maximal rewrite words of length at least 2, whose first two letters are in X and Y, respectively, and whose other letters are all in Z. ♢

Our objective in this section is in particular to define a map

$$\xi(\cdot) : \mathcal{A}\text{-LTL} \longrightarrow \Sigma$$

such that $\xi(\varphi)$ is a model of φ, in the sense that the maximal rewrite words constrained by $\xi(\varphi)$ are exactly those which satisfy φ. In formal terms, we expect $\xi(\cdot)$ to satisfy the following property, for all $\Pi \subseteq \mathcal{T}$ and $\varphi \in \mathcal{A}$-LTL:

$$(\!|\Pi \S \xi(\varphi)|\!) = \{\, w \in (\!|\Pi|\!) \mid w \models \varphi \,\}\,. \tag{4.4}$$

The map $\xi(\cdot)$ will have to be built inductively on the structure of its argument, and it so happens that all the tools needed to handle the base cases are already defined. Let us start by observing that, as one would expect, the empty signature carries no constraint at all, which bridges constrained words and maximal rewrite words:

∀ Lemma 4.2: No Constraints

It holds that $(\!|\Pi \S \varepsilon|\!) = (\!|\Pi|\!)$.

Proof. For this first proof, all steps have been detailed. We have

$$\begin{aligned}
(\!|\Pi \S \varepsilon|\!) &= (\!|\Pi \S \wr \S \mathcal{R} \mid \overline{\mathbb{N}} \wr|\!) && \text{(i)}\\
&= \{\, w \in (\!|\Pi|\!) \mid \#w \in \nabla\varepsilon \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \varepsilon[k] \,\} && \text{(ii)}\\
&= \{\, w \in (\!|\Pi|\!) \mid \#w \in \overline{\mathbb{N}} \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \mathcal{R} \,\} && \text{(iii)}\\
&= \{\, w \in (\!|\Pi|\!) \mid \top \ \wedge\ \top \,\} = \{\, w \in (\!|\Pi|\!) \,\} = (\!|\Pi|\!)\,, && \text{(iv)}
\end{aligned}$$

where step (i) proceeds by definition of the empty signature, step (ii) by definition of constrained rewrite words, step (iii) by definition of the "at operator" for signatures, and step (iv) rests on all lengths being in $\overline{\mathbb{N}}$, and all rules in $\mathcal{R}$. ■

As an immediate consequence of the above, the empty signature is a model of $\top$.

∀ Lemma 4.3: True

Taking $\xi(\top) = \varepsilon$ satisfies (4.4).

Proof. $(\!|\Pi \S \xi(\top)|\!) = (\!|\Pi \S \varepsilon|\!) = (\!|\Pi|\!) = \{\, w \in (\!|\Pi|\!) \mid w \models \top \,\}$. ■

Conversely, to handle $\bot$, we need a signature that rejects every possible rewrite word; in that case there are many possible, equally valid choices, the most straightforward of which is as follows:

Lemma 4.4: False

Taking $\xi(\bot) = \wr \varnothing \mid \varnothing \int$ satisfies (4.4).

Proof. The result rests on $x \in \varnothing$ being false for all x.

$$\begin{aligned}(\!|\Pi \mathbin{\wr} \xi(\bot)|\!) &= (\!|\Pi \mathbin{\wr} \wr \varnothing \mid \varnothing \int|\!) \\ &= \{ w \in (\!|\Pi|\!) \mid \#w \in \varnothing \wedge \forall k \in \operatorname{dom} w,\ w(k) \in \varnothing \} \\ &= \{ w \in (\!|\Pi|\!) \mid \bot \} = \{ w \in (\!|\Pi|\!) \mid w \models \bot \}. \end{aligned}$$

The introduction to this section has already dealt with an example more complicated than the literal "X" alone, from an intuitive perspective. The next lemma begins to show why the proposed translation was correct.

Lemma 4.5: Positive Literal

Taking $\xi(X) = \wr X \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}}_1 \int$ satisfies (4.4).

Proof. Straightforward by translating $\#w$ in terms of $\operatorname{dom} w$.

$$\begin{aligned}&(\!|\Pi \mathbin{\wr} \xi(X)|\!) = (\!|\Pi \mathbin{\wr} \wr X \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}}_1 \int|\!) \\ &= \{ w \in (\!|\Pi|\!) \mid \#w \in \nabla\sigma \wedge \forall k \in \operatorname{dom} w, w(k) \in \sigma[k] \} \\ &= \{ w \in (\!|\Pi|\!) \mid \#w \geqslant 1 \wedge w(1) \in X \wedge \forall k \in \operatorname{dom} w, k > 1 \Rightarrow w(k) \in \mathcal{R} \} \\ &= \{ w \in (\!|\Pi|\!) \mid 1 \in \operatorname{dom} w \wedge w(1) \in X \} \\ &= \{ w \in (\!|\Pi|\!) \mid (w, 1) \models X \} = \{ w \in (\!|\Pi|\!) \mid w \models X \}. \end{aligned}$$

Negative literals are translated in roughly the same way as positive ones, the main difference being that the length 0 is not excluded from their support.

Lemma 4.6: Negative Literal

Taking $\xi(\neg X) = \wr \mathcal{R} \setminus X \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}} \int$ satisfies (4.4).

Proof. It suffices to break up the $\forall k \in \operatorname{dom} w$:

$$\begin{aligned}&(\!|\Pi \mathbin{\wr} \xi(\neg X)|\!) = (\!|\Pi \mathbin{\wr} \wr \mathcal{R} \setminus X \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}} \int|\!) \\ &= \{ w \in (\!|\Pi|\!) \mid \#w \in \overline{\mathbb{N}} \ \wedge\ \forall k \in \operatorname{dom} w, w(k) \in \sigma[k] \} \\ &= \{ w \in (\!|\Pi|\!) \mid \forall k \in \operatorname{dom} w, w(k) \in \sigma[k] \} \\ &= \{ w \in (\!|\Pi|\!) \mid 1 \in \operatorname{dom} w \Rightarrow w(1) \notin X \ \wedge\ \forall k \in \operatorname{dom} w,\ k > 1 \Rightarrow w(k) \in \mathcal{R} \} \\ &= \{ w \in (\!|\Pi|\!) \mid 1 \in \operatorname{dom} w \Rightarrow w(1) \notin X \} = \{ w \in (\!|\Pi|\!) \mid w \models \neg X \}. \end{aligned}$$

Now that the base cases are all covered, we move on to the inductive cases, the first of which is conjunction. Let us take the simplest possible example: $X \wedge Y$. We have by Lemma 4.5

$$\xi(X) = \wr X \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}}_1 \int \quad \text{and} \quad \xi(Y) = \wr Y \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}}_1 \int,$$

but also, considering that $X \cap Y$ is a positive literal as well:

$$\xi(X \cap Y) = \wr X \cap Y \mathbin{\wr} \mathcal{R} \mid \overline{\mathbb{N}}_1 \int.$$

It should be intuitively pretty clear that $\xi(X \cap Y)$ encodes both the constraints of $\xi(X)$ and $\xi(Y)$; in that particular case it follows from the semantic equivalence of $w \models X \cap Y$ and $w \models X \ \wedge \ w \models Y$. However, the general idea that "conjunction" between signatures is translated by intersections stands on its own, as the next lemmata will show.

$\sigma \otimes \sigma'$: *signature product*

SIGNATURE PRODUCT. Let σ and σ' be two signatures; then their *product* is another signature defined as $\sigma \otimes \sigma' = \wr g \mid \nabla\sigma \cap \nabla\sigma' \int$, where

$$g = \left| \begin{array}{ccc} \operatorname{dom} \partial\sigma \cup \operatorname{dom} \partial\sigma' & \longrightarrow & \wp(\mathcal{R}) \\ k & \longmapsto & \sigma[k] \cap \sigma'[k] \end{array} \right. .$$

Note that as a consequence, $\forall k \in \mathbb{N}_1$, $(\sigma \otimes \sigma')[k] = \sigma[k] \cap \sigma'[k]$.

Example: Let us take the two signatures $\sigma = \wr X, Y \S Z \mid \mathbb{N}_2 \int$ and $\rho = \wr X' \S Z' \mid \mathbb{N}_3 \int$; then $\sigma \otimes \rho = \wr X \cap X', Y \cap Z' \S Z \cap Z' \mid \mathbb{N}_3 \int$. ♢

Signature product is fairly well-behaved with respect to constrained words, and can be "broken down" and replaced by the intersection of simpler constrained sets. The next lemma and its generalisation (4.6)[p69] show this to be true of finite products, and this property will later be generalised to infinite products (Lem. 4.12[p70]).

∀ Lemma 4.7: Breaking Finite Products

For any signatures $\sigma, \rho \in \Sigma$, and any language Π, we have $(\!|\Pi \S \sigma \otimes \rho|\!) = (\!|\Pi \S \sigma|\!) \cap (\!|\Pi \S \rho|\!)$.

Proof. $(\!|\Pi \S \sigma \otimes \rho|\!)$ is, by definition, the set of $w \in (\!|\Pi|\!)$ such that

$$\begin{aligned} & \#w \in \nabla\sigma \cap \nabla\rho \ \wedge \ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma[k] \cap \rho[k] \\ \iff & \#w \in \nabla\sigma \ \wedge \ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma[k] \\ & \wedge \ \#w \in \nabla\rho \ \wedge \ \forall k \in \operatorname{dom} w,\ w(k) \in \rho[k] \\ \iff & w \in (\!|\Pi \S \sigma|\!) \ \wedge \ w \in (\!|\Pi \S \rho|\!) \\ \iff & w \in (\!|\Pi \S \sigma|\!) \cap (\!|\Pi \S \rho|\!) \ . \end{aligned}$$

Thus $(\!|\Pi \S \sigma \otimes \rho|\!) = (\!|\Pi|\!) \cap (\!|\Pi \S \sigma|\!) \cap (\!|\Pi \S \rho|\!) = (\!|\Pi \S \sigma|\!) \cap (\!|\Pi \S \rho|\!)$. ■

∀ Lemma 4.8: Conjunction

Provided that the subformulæ $\xi(\varphi)$ and $\xi(\psi)$ satisfy (4.4), taking $\xi(\varphi \wedge \psi) = \xi(\varphi) \otimes \xi(\psi)$ satisfies (4.4).

Proof. Straightforward using the previous lemma:

$$\begin{aligned} (\!|\Pi \S \xi(\varphi \wedge \psi)|\!) &= (\!|\Pi \S \xi(\varphi) \otimes \xi(\psi)|\!) = (\!|\Pi \S \xi(\varphi)|\!) \cap (\!|\Pi \S \xi(\psi)|\!) \\ &= \{ w \in (\!|\Pi|\!) \mid w \models \varphi \} \cap \{ w \in (\!|\Pi|\!) \mid w \models \psi \} \\ &= \{ w \in (\!|\Pi|\!) \mid w \models \psi \ \wedge \ w \models \varphi \} \\ &= \{ w \in (\!|\Pi|\!) \mid w \models \psi \wedge \varphi \} \, . \end{aligned}$$ ■

We shall generalise results on signature products to infinitary cases later on, which will be necessary to encode $\Box\varphi$. Meanwhile, let us turn our attention to the strong

and weak next operators. Let us consider for instance a formula φ whose signature is given by

$$\xi(\varphi) = \wr W, X, Y \S Z \mid \overline{\mathbb{N}}_2 \int ,$$

and propose plausible candidates for $\xi(\circ^1\varphi)$ and $\xi(\bullet^1\varphi)$. Although we still lack all the formal tools to prove it, after numerous similar examples it is easy to derive that

$$\varphi = W \wedge \bullet^1 X \wedge \circ^2 Y \wedge \circ^3 \Box Z .$$

The lengths of the words $w \models \circ^1\varphi$ can therefore be enumerated. By the semantics of $\circ^1$, if $\#w = 0$ or $\#w = 1$, then w is automatically a model of $\circ^1\varphi$. Suppose that $\#w \geqslant 2$; then it must be the case that $w^2 \models \varphi$. In particular, because of the strong next this means that $\#w^2 \in \overline{\mathbb{N}}_2$, in other words $\#w^2 \geqslant 2$; thus, since $\#w^2 = \#w - 1$, we have $\#w \geqslant 3$. So the set of acceptable lengths is $[\![0,1]\!] \cup \overline{\mathbb{N}}_3 = \overline{\mathbb{N}} \setminus \{2\}$. Now, as far as the rewrite rules are concerned, the first rule, if it exists, can be anything; obviously the second one must live in W, the third in X, the fourth in Y, and all the subsequent rules must come from Z. So we have derived the following signature:

$$\xi(\circ^1\varphi) = \wr \mathcal{R}, W, X, Y \S Z \mid [\![0,1]\!] \cup \overline{\mathbb{N}}_3 \int .$$

It is clear that $\xi(\bullet^1\varphi)$ will have the same core as $\xi(\circ^1\varphi)$, but a different support. Indeed in that case the length 0 and 1 are clearly not suitable, while the rest of the previous reasoning still holds. Thus we have immediately

$$\xi(\bullet^1\varphi) = \wr \mathcal{R}, W, X, Y \S Z \mid \overline{\mathbb{N}}_3 \int .$$

The work done on this example is generalised in the next definitions, and this suffices to compute $\xi(\bullet^1\varphi)$ and $\xi(\circ^1\varphi)$ for all φ, as shown in the next lemma.

arithmetic overloading, +

Arithmetic Overloading. We overload the operator $+$ on the profile $\wp(\overline{\mathbb{N}}) \times \mathbb{N} \to \wp(\overline{\mathbb{N}})$ such that, for any $S \in \wp(\overline{\mathbb{N}})$ and $n \in \mathbb{N}$, we have

$$S + n = \{ k + n \mid k \in S \} .$$

Right Shifts. Let $\sigma \in \Sigma$, $m \in \mathbb{N}$ and $\mathcal{R}_1 = \mathcal{R}, \ldots, \mathcal{R}_m = \mathcal{R}$; then we define the *weak* m*-right shift of* σ as

$\sigma \triangleright m$: weak m-right shift of σ

$$\sigma \triangleright m = \wr \mathcal{R}_1, \ldots, \mathcal{R}_m, \partial\sigma(1), \ldots, \partial\sigma(\#\sigma) \S \partial\sigma(\omega) \mid [\![0, m]\!] \cup (\nabla\sigma + m) \int ,$$

while the *strong* m*-right shift of* σ is

$\sigma \blacktriangleright m$: strong m-right shift of σ

$$\sigma \blacktriangleright m = \wr \mathcal{R}_1, \ldots, \mathcal{R}_m, \partial\sigma(1), \ldots, \partial\sigma(\#\sigma) \S \partial\sigma(\omega) \mid (\nabla\sigma \setminus \{0\}) + m \int .$$

The only point in those definitions that was not readily apparent in the above example is the $\nabla\sigma \setminus \{0\}$ appearing in the support of the right shift. The reason for its introduction is best understood in the context of a formula φ whose signature admits zero in its support; when behind a strong next of level zero ($\bullet^0$), the only change in the support of the signature is the removal of the zero. Let us write down a few immediate equations that will come in useful in the proofs of Lem. 4.9, 4.10 and 4.13[p71]: for all $m \in \mathbb{N}$ and all $k \in \mathbb{N}_1$, if $k \leqslant m$, $(\sigma \blacktriangleright m)[k] = (\sigma \triangleright m)[k] = \mathcal{R}$ and if $k > m$, $(\sigma \blacktriangleright m)[k] = (\sigma \triangleright m)[k] = \sigma[k - m]$.

∀ Lemma 4.9: Weak Next

Provided that the signature of the subformula $\xi(\varphi)$ satisfies (4.4), taking $\xi(\circ^m \varphi) = \xi(\varphi) \triangleright m$ satisfies (4.4).

Proof. We write $\sigma = \xi(\varphi)$ and $\sigma_m = \xi(\varphi) \triangleright m$. Let $w \in (\!|\Pi|\!)$, then $w \models \circ^m \varphi$ iff

$$\begin{aligned}
&1 + m \notin \operatorname{dom} w \ \vee\ w^{1+m} \models \varphi \iff \#w \in [\![0, m]\!] \ \vee\ w^{1+m} \models \varphi \\
\iff &\#w \in [\![0, m]\!] \ \vee\ (\#w^{1+m} \in \nabla\sigma \ \wedge\ \forall k \in \operatorname{dom} w^{1+m},\ w^{1+m}(k) \in \sigma[k]) \\
\iff &\#w \in [\![0, m]\!] \ \vee\ (\#w - m \in \nabla\sigma \ \wedge\ \forall k \in [\![1 + m, \#w]\!],\ w(k) \in \sigma[k - m]) \\
\iff &(\#w \in [\![0, m]\!] \ \vee\ \#w \in \nabla\sigma + m) \ \wedge\ (\forall k \in [\![1 + m, \#w]\!],\ w(k) \in \sigma_m[k]) \\
\iff &\#w \in \nabla\sigma_m \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma_m[k] \iff w \in (\!|\Pi \fatsemi \sigma_m|\!)\ .
\end{aligned}$$

Note that $\#w^{1+m} = \#w - m$ only holds because we can safely assume, by the left member of the disjunction, that $\#w \in [\![0, m]\!]$, or in other words, $\#w > m$. ∎

∀ Lemma 4.10: Strong Next

Provided that the signature of the subformula $\xi(\varphi)$ satisfies (4.4), taking $\xi(\bullet^m \varphi) = \xi(\varphi) \blacktriangleright m$ satisfies (4.4).

Proof. We write $\sigma = \xi(\varphi)$ and $\sigma_m = \xi(\varphi) \triangleright m$. Let $w \in (\!|\Pi|\!)$, then $w \models \bullet^m \varphi$ iff

$$\begin{aligned}
&1 + m \in \operatorname{dom} w \ \wedge\ w^{1+m} \models \varphi \iff \#w > m \ \wedge\ w^{1+m} \models \varphi \\
\iff &\#w > m \ \wedge\ \#w \in \nabla\sigma + m \ \wedge\ \forall k \in \operatorname{dom} w^{1+m},\ w^{1+m}(k) \in \sigma[k] \\
\iff &\#w \in (\nabla\sigma + m) \setminus \{m\} \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma_m[k] \\
\iff &\#w \in (\nabla\sigma \setminus \{0\}) + m \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma_m[k] \\
\iff &\#w \in \nabla\sigma_m \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma_m[k] \iff w \in (\!|\Pi \fatsemi \sigma_m|\!)\ .
\end{aligned}$$

∎

Now there only remains to deal with the last inductive case: the $\Box$ operator. To this end, we recall Lem. 4.1[p57], whose first statement gives an equivalent expression of $\Box$ in terms of conjunction and weak next:

$$\Box\,\varphi \iff \bigwedge_{m=0}^{\infty} \circ^m \varphi\ .$$

We have already seen how signatures of formulæ involving those operators are computed, so it stands to reason that we should be able to simply use those previous results and write

$$\xi(\Box\,\varphi) = \bigotimes_{m=0}^{\infty} \left[\xi(\varphi) \triangleright m\right], \tag{4.5}$$

thus getting this last translation "for free". This is exactly the approach which we shall follow, but there remains work to do in order to assign a precise meaning to (4.5), prove its correctness with respect to (4.4), and derive a closed form of (4.5) in terms of $\xi(\varphi)$, so that we may actually use it in an algorithm. In order to do so, we must first establish the legitimacy of using a *finite* extended notation

$\bigotimes_{k=l}^{\infty} \sigma_k$: *signature product, finite big operator notation*

$$\bigotimes_{k=l}^{m} \sigma_k = \sigma_l \otimes \sigma_{l+1} \otimes \cdots \otimes \sigma_m$$

with the usual properties, which is provided by Remark 4.11.

∀ Remark 4.11: Extended Product Notation

The set of signatures Σ, equipped with the signature-product $\otimes$, forms a commutative monoid whose neutral element is ε.

Proof. The associativity and commutativity of $\otimes$ stem directly from those of $\cup$ and $\cap$. The neutrality of $\varepsilon = \langle\!\langle ; \mathcal{R} \mid \overline{\mathbb{N}} \rangle\!\rangle$ stems from that of $\overline{\mathbb{N}}$ (= $\nabla\varepsilon$) for $\cap$ within $\wp(\overline{\mathbb{N}})$, of $\mathcal{R}$ (= $\varepsilon[k], \forall k$) for $\cap$ within $\wp(\mathcal{R})$, and of $\varnothing$ (its domain) for $\cup$. ∎

All our previous results involving product behave as one would expect them to under extended notations; in particular, Lemma 4.7[p66] instantly generalises to

$$(\!|\Pi ; \bigotimes_{k=1}^{n} \sigma_k|\!) = \bigcap_{k=1}^{n} (\!|\Pi ; \sigma_k|\!) \qquad \forall \sigma_1, \ldots, \sigma_n \in \Sigma,\ n \in \mathbb{N}\,, \tag{4.6}$$

and it follows that Lemma 4.8[p66] becomes

$$\xi\left(\bigwedge_{k=1}^{n} \varphi_k\right) = \bigotimes_{k=1}^{n} \xi(\varphi_k) \qquad \forall \varphi_1, \ldots, \varphi_n \in \mathcal{A}\text{-LTL},\ n \in \mathbb{N}\,. \tag{4.7}$$

Infinite Products. This is still not enough, however, as (4.5) involves an *infinite* product, which is customarily defined based on finite products as follows: the infinite product $\bigotimes_{k=l}^{\infty} \sigma_k$ converges if and only if the associated sequence of partial products $(\bigotimes_{k=l}^{n} \sigma_k)_{n \in \mathbb{N}_l}$ converges, and in that case

$\bigotimes_{k=l}^{\infty} \sigma_k$: *signature product, infinite big operator notation*

$$\bigotimes_{k=l}^{\infty} \sigma_k = \lim_{n \to \infty} \bigotimes_{k=l}^{n} \sigma_k\,.$$

This definition rests upon the introduction of suitable notions of *convergence* and *limit* for sequences of signatures, to which we must now attend. Let $\rho = (\sigma_n)_{n \in \mathbb{N}}$ be an infinite sequence of signatures. It is said to be *convergent* if

(1) the sequence $(\nabla\sigma_n)_{n \in \mathbb{N}}$ converges towards a limit $\nabla\sigma_\infty$,

(2) for all $k \in \mathbb{N}_1$, the sequence $(\sigma_n[k])_{n \in \mathbb{N}}$ converges towards a limit $\sigma_\infty[k]$,

(3) the sequence of limits $(\sigma_\infty[k])_{k \in \mathbb{N}_1}$ itself converges towards a limit $\sigma_\infty[\infty]$.

We call the sequence $(\sigma_\infty[k])_{k \in \mathbb{N}_1}$ the *limit core*. It is not directly in the form of a bona fide signature core. However, its co-domain being $\wp(\mathcal{R})$, which is finite, there exists a rank $N \geqslant 0$ such that for all $k > N$, $\sigma_\infty[k] = \sigma_\infty[\infty]$, and thus, taking the smallest such N, we define the *limit* of ρ, which we denote by $\lim \rho$ or $\lim_{n \to \infty} \sigma_n$, or even more simply by σ_∞, as

limit core

$\sigma_\infty, \lim_{n \to \infty} \sigma_n$: *limit of a sequence of signatures*

$$\lim_{n \to \infty} \sigma_n = \langle\!\langle \sigma_\infty[1], \ldots, \sigma_\infty[N] ; \sigma_\infty[\infty] \mid \nabla\sigma_\infty \rangle\!\rangle\,.$$

Note that the core of the limit is equivalent to the limit core, in the intuitive sense that they define the same constrained words. Otherwise ρ is *divergent*, and its limit is left undefined.

divergent sequence of signatures

On that subject, the meticulous reader will have noticed that, throughout this section, we have omitted to mention that signatures are not unique, in the sense

that for instance $\wr X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 \int$, $\wr X, \mathcal{R} \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 \int$ and $\wr X, \mathcal{R}, \mathcal{R} \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 \int$ etcetera are all equally valid choices for $\xi(X)$. This slight ambiguity can easily be resolved by defining a notion of extensional equivalence between signatures and working with representatives of the classes. We have chosen to eschew this discussion entirely in the main text, as it is mostly a cosmetic consideration and would make the discussion more cumbersome. For future reference, *extensional equivalence* can be defined as $\sigma \equiv \rho \iff \nabla\sigma = \nabla\rho \wedge \forall k \in \mathbb{N}_1,\ \sigma[k] = \rho[k]$, and has the following fundamental characterisation: $\sigma \equiv \rho \iff \forall \Pi \subseteq \mathcal{T}, \forall \mathcal{R},\ (\!|\Pi \fatsemi \sigma|\!) = (\!|\Pi \fatsemi \rho|\!)$.

$\sigma \equiv \rho$: *extensional equivalence*

Example: Taking $X_i = X\ \forall i$, we have $\lim_{n\to\infty} \wr X_1, \dots, X_n \fatsemi \mathcal{R} \mid [\![1, n]\!] \int \equiv \wr \fatsemi X \mid \mathbb{N}_1 \int$. It is easy to build artificial sequences that fail (1), (2) or (3). ♢

Without further ado, we can bring this notion of convergence to bear and further generalise (4.6) to infinitary cases. This result will be central to the proof of correctness.

∀ Lemma 4.12: Breaking Infinite Products

For any language Π and any sequence $(\sigma_n)_{n\in\mathbb{N}}$ of signatures such that the infinite product $\bigotimes_{n=0}^{\infty} \sigma_n$ converges,

$$(\!|\Pi \fatsemi \bigotimes_{n=0}^{\infty} \sigma_n|\!) = \bigcap_{n=0}^{\infty} (\!|\Pi \fatsemi \sigma_n|\!)\ .$$

Proof. Let $\rho_m = \bigotimes_{n=0}^{m} \sigma_n$, for $m \in \overline{\mathbb{N}}$, and $\rho_\infty = \lim_{m\to\infty} \rho_m$. Let us compute the support of this limit

$$\nabla\rho_\infty = \lim_{m\to\infty} \nabla\rho_m = \lim_{m\to\infty} \nabla\left(\bigotimes_{n=0}^{m} \sigma_n\right) = \lim_{m\to\infty} \bigcap_{n=0}^{m} \nabla\sigma_n = \bigcap_{n=0}^{\infty} \nabla\sigma_n\ ,$$

and its limit core; let $k \in \mathbb{N}_1$ in

$$\rho_\infty[k] = \lim_{m\to\infty} \rho_m[k] = \lim_{m\to\infty} \left[\left(\bigotimes_{n=0}^{m} \sigma_n\right)[k]\right] = \lim_{m\to\infty} \bigcap_{n=0}^{m} \sigma_n[k] = \bigcap_{n=0}^{\infty} \sigma_n[k]\ .$$

So, a word $w \in (\!|\Pi \fatsemi \rho_\infty|\!)$ is a word of $(\!|\Pi|\!)$ such that

$$\begin{aligned}
&\#w \in \nabla\rho_\infty \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \rho_\infty[k] \\
\iff &\#w \in \textstyle\bigcap_{n=0}^{\infty} \nabla\sigma_n \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \bigcap_{n=0}^{\infty} \sigma_n[k] \\
\iff &\textstyle\bigwedge_{n=0}^{\infty} \left[\#w \in \nabla\sigma_n \ \wedge\ \forall k \in \operatorname{dom} w,\ w(k) \in \sigma_n[k]\right] \\
\iff &\textstyle\bigwedge_{n=0}^{\infty} w \in (\!|\Pi \fatsemi \sigma_n|\!) \iff w \in \bigcap_{n=0}^{\infty} (\!|\Pi \fatsemi \sigma_n|\!)\ .
\end{aligned}$$

■

We have just established that, provided that the infinite product is convergent, we can break it, which will be as essential to the correctness proof as finite product breaking (Lem. 4.7) was to the proof of Lem. 4.8. There remains to prove that the infinite product of (4.5) is indeed convergent. Equivalently, we can see this as a proof that the computation of $\xi(\Box\,\varphi)$ always terminates and yields a useable signature. We shall then finally be able to write the product in a closed form.

∀ Lemma 4.13: Shift-Product Convergence

Let $(\sigma_n)_{n\in\mathbb{N}}$ be any sequence of signatures, and $(\rho_n)_{n\in\mathbb{N}}$ its associated sequence of partial products $(\bigotimes_{i=0}^{n}\sigma_i)_{n\in\mathbb{N}}$. Then $(\rho_n)_{n\in\mathbb{N}}$ satisfies convergence criteria (1) and (2). Furthermore, if σ is a given signature and $\sigma_i = \sigma \blacktriangleright i$ or $\sigma_i = \sigma \triangleright i$, for any $i \in \mathbb{N}$, then criterion (3) is satisfied as well, and the infinite product $\bigotimes_{n=0}^{\infty}\sigma_n$ converges.

Proof. **(1)** For all $n \in \mathbb{N}$, $\nabla\rho_n = \bigcap_{i=0}^{n}\nabla\sigma_i$, thus it is clear that $\nabla\rho_n = \bigcap_{i=0}^{n}\nabla\sigma_i \supseteq \bigcap_{i=0}^{n}\nabla\sigma_i \cap \nabla\sigma_{n+1} = \bigcap_{i=0}^{n+1}\nabla\sigma_i = \nabla\rho_{n+1}$ or, in other words, $(\nabla\rho_n)_{n\in\mathbb{N}}$ is a (trivial) contracting sequence of finite sets. Therefore it converges towards $\bigcap_{i=0}^{\infty}\nabla\sigma_i$. **(2)** Let $k \in \mathbb{N}_1$; we have

$$\rho_n[k] = \left(\bigotimes_{i=0}^{n}\sigma_i\right)[k] = \bigcap_{i=0}^{n}\sigma_i[k]\,,$$

and thus $\rho_n[k] = \bigcap_{i=0}^{n}\sigma_i[k] \supseteq \bigcap_{i=0}^{n+1}\sigma_i[k] = \rho_{n+1}[k]$ and again, $(\rho_n[k])_{n\in\mathbb{N}}$ is a trivial contracting sequence of finite sets; therefore it converges towards a limit which we denote by $\rho_\infty[k] = \bigcap_{i=0}^{\infty}\sigma_i[k]$. **(3)** Suppose now that $\sigma_i = \sigma \triangleright i$ (resp. $\sigma_i = \sigma \blacktriangleright i$, the computation will be unchanged), we have

$$\begin{aligned}\rho_\infty[k] &= \bigcap_{i=0}^{\infty}\sigma_i[k] = \bigcap_{i=0}^{\infty}(\sigma \triangleright i)[k] = \left(\bigcap_{i=0}^{k-1}(\sigma \triangleright i)[k]\right) \cap \left(\bigcap_{i=k}^{\infty}(\sigma \triangleright i)[k]\right)\\ &= \left(\bigcap_{i=0}^{k-1}\sigma[k-i]\right) \cap \left(\bigcap_{i=k}^{\infty}\mathcal{R}\right) = \bigcap_{i=0}^{k-1}\sigma[k-i] = \bigcap_{i=1}^{k}\sigma[i]\,.\end{aligned}$$

Given that for all $i > \#\sigma$, $\sigma[i] = \sigma[\omega]$, it follows that for all $k > \#\sigma$, $\rho_\infty[k] = \bigcap_{i=1}^{\#\sigma+1}\sigma[i]$. Thus $(\rho_\infty[k])_{k\in\mathbb{N}_1}$ converges. This shows that the infinite product $\bigotimes_{n=0}^{\infty}\sigma_n$ is convergent. Specifically, we have the following limit:

$$\bigotimes_{n=0}^{\infty}\sigma_n = \wr\sigma[1], \sigma[1]\cap\sigma[2], \ldots, \bigcap_{i=1}^{k}\sigma[i], \ldots, \bigcap_{i=1}^{\#\sigma}\sigma[i] \,;\, \bigcap_{i=1}^{\#\sigma+1}\sigma[i] \mid \bigcap_{i=0}^{\infty}\nabla\sigma_i\int.$$

This is not completely a closed form yet, however, as the support $\nabla\rho_\infty = \bigcap_{i=0}^{\infty}\nabla\sigma_i$ remains expressed in terms of σ_i and an infinite intersection. Let us consider the case $\sigma_i = \sigma \triangleright i$, which is what we are really interested in. We have $0 \in \nabla\rho_\infty$, since for all $i \in \mathbb{N}$, $0 \in [\![0, i]\!] \subseteq \nabla\sigma_i$. This is coherent with the fact that $\lambda \models \Box\varphi$, for all $\varphi \in \mathrm{LTL}$. Furthermore, let $p \geqslant 1$ such that $p \notin \nabla\sigma$. For any $i \in \mathbb{N}$, it follows that $p+i \notin (\nabla\sigma+i)$, and since $p+i \notin [\![0, i]\!]$, we have $p+i \notin \nabla\sigma_i$ and finally, $p+i \notin \nabla\rho_\infty$. In other words, we have just shown that for all $p \geqslant 1$, $p \notin \nabla\sigma \Rightarrow \mathbb{N}_p \cap \nabla\rho_\infty = \varnothing$. Now let us take $p \geqslant 1$ such that $[\![1, p]\!] \subseteq \nabla\sigma$, then for all $i \in \mathbb{N}$ we have $[\![0, i]\!] \cup [\![1+i, p+i]\!] = [\![0, p+i]\!] \subseteq \nabla\sigma_i$; thus in particular $p \in \nabla\rho_\infty$. There remains to observe that trivially $+\infty \in \nabla\sigma \iff +\infty \in \nabla\rho_\infty$, and we can now summarise this into a closed form for $\nabla\rho_\infty$:

$$\nabla\rho_\infty = \bigcap_{i=0}^{\infty}\nabla(\sigma \triangleright i) = \{0\} \cup \left[\!\!\left[1, \min\left(\overline{\mathbb{N}}_1 \setminus (\nabla\sigma \cap \mathbb{N}_1)\right) - 1\right]\!\!\right] \cup \left(\nabla\sigma \cap \{+\infty\}\right).$$

For $\sigma_i = \sigma \blacktriangleright i$ the computation is much more direct, as we have $m \notin \nabla\sigma_m$, for all $m \in \mathbb{N}$. Thus in that case $\nabla\rho_\infty = \nabla\sigma \cap \{+\infty\}$. ■

We now have every tool we need to write down the computation of $\xi(\Box\,\varphi)$.

∀ Lemma 4.14: Always

Provided that the signature of the subformula $\xi(\varphi)$ satisfies (4.4), taking $\xi(\Box\,\varphi) = \bigotimes_{m=0}^{\infty}\left[\xi(\varphi) \rhd m\right]$ satisfies (4.4).

Proof. By application of Lem. 4.13, 4.12, 4.9[p68] and 4.1[p57], we have

$$\begin{aligned} & w \in \left(\!\left|\Pi \fatsemi \bigotimes_{m=0}^{\infty}\left[\xi(\varphi) \rhd m\right]\right|\!\right) \iff w \in \bigcap_{m=0}^{\infty}\left(\!\left|\Pi \fatsemi \xi(\varphi) \rhd m\right|\!\right) \\ \iff & \bigwedge_{m=0}^{\infty} w \in \left(\!\left|\Pi \fatsemi \xi(\varphi) \rhd m\right|\!\right) \iff \bigwedge_{m=0}^{\infty} w \in \left(\!\left|\Pi \fatsemi \xi(\circ^m\varphi)\right|\!\right) \\ \iff & \bigwedge_{m=0}^{\infty} w \models \circ^m\varphi \iff w \models \Box\,\varphi\,. \end{aligned}$$

■

Example: Let us compute the signatures of two common $\Box$-based formulæ; although we could use the closed form given in the proof of Lemma 4.13 directly, we shall show a few manual steps in order to better see how the result is built:

$$\begin{aligned} \xi(\Box\,X) &= \bigotimes_{m=0}^{\infty}\left[\xi(X) \rhd m\right] = \bigotimes_{m=0}^{\infty}\left[\wr X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 \int \rhd m\right] \\ &= \bigotimes_{m=0}^{\infty}\left[\wr \mathcal{R}_1, \ldots, \mathcal{R}_m, X \fatsemi \mathcal{R} \mid [\![0, m]\!] \cup (\overline{\mathbb{N}}_1 + m) \int\right] \\ &= \bigotimes_{m=0}^{\infty}\left[\wr \mathcal{R}_1, \ldots, \mathcal{R}_m, X \fatsemi \mathcal{R} \mid [\![0, m]\!] \cup \overline{\mathbb{N}}_{1+m} \int\right] \\ &= \bigotimes_{m=0}^{\infty}\left[\wr \mathcal{R}_1, \ldots, \mathcal{R}_m, X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}} \int\right] \\ &= \wr X \fatsemi X \mid \overline{\mathbb{N}} \int \equiv \wr \fatsemi X \mid \overline{\mathbb{N}} \int\,. \end{aligned}$$

Note that the closed form by itself yields $\wr X \fatsemi X \mid \overline{\mathbb{N}} \int$, which could in this case be manually simplified into the "prettier" $\wr \fatsemi X \mid \overline{\mathbb{N}} \int$. These two signatures are obviously equivalent in the sense discussed in the paragraph on extensional equivalence page 69, and we shall henceforth carry out similar simplifications as matter of course. As for the result itself, it is what was expected, as words of any length can satisfy $\Box\,\varphi$.

$$\begin{aligned} \xi(\Box \bullet^1 X) &= \bigotimes_{m=0}^{\infty}\left[\xi(\bullet^1 X) \rhd m\right] = \bigotimes_{m=0}^{\infty}\left[\wr \mathcal{R}, X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_2 \int \rhd m\right] \\ &= \bigotimes_{m=0}^{\infty}\left[\wr \mathcal{R}_1, \ldots, \mathcal{R}_m, \mathcal{R}, X \fatsemi \mathcal{R} \mid [\![0, m]\!] \cup \overline{\mathbb{N}}_{2+m} \int\right] \\ &= \wr \mathcal{R} \fatsemi X \mid \{0\} \cup \{+\infty\} \int\,. \end{aligned}$$

There again, the result should come as no surprise: the empty word satisfies $\Box \bullet^1 X$ vacuously, and as soon as there is one transition, then there must be another, and another… hence a word satisfying $\Box \bullet^1 X$ can only be either empty or infinite. ♢

Lemma 4.14 concludes our discussion of signatures. Figure 4.2 and Thm. 4.15 summarise the eight main lemmata of this section.

△ Theorem 4.15: Signatures

$$\xi(\top) = \langle\!\!\langle \S \mathcal{R} \mid \overline{\mathbb{N}} \rangle\!\!\rangle = \varepsilon \qquad \xi(\bot) = \langle\!\!\langle \S \varnothing \mid \varnothing \rangle\!\!\rangle$$
$$\xi(X) = \langle\!\!\langle X \S \mathcal{R} \mid \overline{\mathbb{N}}_1 \rangle\!\!\rangle \qquad \xi(\neg X) = \langle\!\!\langle \mathcal{R} \setminus X \S \mathcal{R} \mid \overline{\mathbb{N}} \rangle\!\!\rangle$$
$$\xi(\bullet^m \varphi) = \xi(\varphi) \blacktriangleright m \qquad \xi(\circ^m \varphi) = \xi(\varphi) \triangleright m$$
$$\xi(\varphi \wedge \psi) = \xi(\varphi) \otimes \xi(\psi) \qquad \xi(\Box\, \varphi) = \bigotimes_{m=0}^{\infty} \Big[\xi(\varphi) \triangleright m \Big]$$

Figure 4.2: Building signatures on $\mathcal{A}$-LTL.

For any $\Pi \subseteq \mathcal{T}$ and any $\varphi \in \mathcal{A}$-LTL, and given the map $\xi(\cdot) : \mathcal{A}\text{-LTL} \to \Sigma$ inductively defined in Fig. 4.2, it holds that

$$(\!|\Pi \S \xi(\varphi)|\!) = \{ w \in (\!|\Pi|\!) \mid w \models \varphi \}\,.$$

4.3 From Temporal Properties to Rewrite Propositions

The technical trek through signatures being now over, we come back to our overarching objective for the first problem, which is to translate temporal properties into rewrite propositions – and of course prove the translation's correctness. This will be accomplished by means of *translation rules* that are used in a way similar to the rules of a classical deduction system. Those rules are made up of *translation blocks*. We define the set $\mathfrak{B}$ of translation blocks as

translation rules

translation blocks

$$\mathfrak{B} = \left\{ \langle \Pi \S \sigma \Vdash \varphi \rangle \;\middle|\; \Pi \subseteq \mathcal{T},\ \sigma \in \Sigma,\ \varphi \in \text{LTL} \right\},$$

where each translation block actually encodes a statement according to the following semantics:

$$\langle \Pi \S \sigma \Vdash \varphi \rangle \quad \equiv \quad \forall w \in (\!|\Pi \S \sigma|\!),\ w \models \varphi\,. \tag{4.8}$$

An *exact translation rule* is a statement of the form

exact translation rule

$$\updownarrow \frac{\langle \Pi \S \sigma \Vdash \varphi \rangle \qquad P(\sigma, \varphi)}{\pi}\,,$$

where the precondition $P \in \Sigma \times \text{LTL} \to \mathbb{B}$ is a predicate on signatures and formulæ that will be omitted entirely if it is a tautology – which is the case in most rules – and π is a mixed rewrite proposition, generated by the following grammar:

$$\pi := \gamma \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \qquad \gamma := \ell = \varnothing \mid \ell \subseteq \ell \mid \Upsilon \qquad X \in \wp(\mathcal{R})$$
$$\ell := \Pi \mid \mathcal{T} \mid X(\ell) \mid X^{-1}(\ell) \mid X^*(\ell) \qquad \Upsilon \in \mathfrak{B}\,.$$

This is the grammar for rewrite propositions, as given in Sec. 4.1.3[p58], with the added production $\gamma := \Upsilon$, where Υ is a translation block. An exact translation rule has the following semantics:

$$\updownarrow \frac{\langle \Pi \S \sigma \Vdash \varphi \rangle \qquad P(\sigma, \varphi)}{\pi} \quad \equiv \quad P(\sigma, \varphi) \implies \big(\langle \Pi \S \sigma \Vdash \varphi \rangle \Leftrightarrow \pi \big)\,.$$

In one instance, we shall give an *under-approximated translation rule*, written and

under-approximated rule

defined in a similar manner as exact rules:

$$\uparrow \frac{\langle \Pi \fatsemi \sigma \Vdash \varphi \rangle \qquad P(\sigma, \varphi)}{\pi} \quad \equiv \quad P(\sigma, \varphi) \implies \big(\pi \Rightarrow \langle \Pi \fatsemi \sigma \Vdash \varphi \rangle\big) .$$

over-approximated rules

While *over-approximated translation rules* could obviously be defined as well, we have no use for them in the context of this work, and the focus is markedly on exact translations. In the following, the unqualified words "translation", "rule" etcetera will always refer to *exact* translations and rules. The modus operandi of the (exact) translation of a formula $\varphi \in \mathcal{R}$-LTL consists in starting with the initial translation block $\langle \Pi \fatsemi \varepsilon \Vdash \varphi \rangle$, and transforming it by successive application of valid exact translation rules until we have a pure rewrite proposition, that is to say until there are no translation blocks left at all. The resulting tree of rules, with $\langle \Pi \fatsemi \varepsilon \Vdash \varphi \rangle$ at the root, will be called a *derivation*. By definition of the translation rules, this means that the rewrite proposition on the leaves of the derivation is equivalent to the initial translation block, and by the next theorem, that block is itself equivalent to the statement that the system $\mathcal{R}$, given the initial language Π, satisfies the property φ. In other words, it is an exact translation in the sense given in our problem statement, Sec. 4.1.3[p58]. Complete examples of derivations are given in Sec. 4.5.1[p97].

derivation

Theorem 4.16: Translation Satisfaction

$\langle \Pi \fatsemi \varepsilon \Vdash \varphi \rangle \iff \mathcal{R}, \Pi \models \varphi.$

Proof. Recall that $(\!|\Pi \fatsemi \varepsilon|\!) = (\!|\Pi|\!)$ by Lemma 4.2[p64];

$$\begin{aligned} \langle \Pi \fatsemi \varepsilon \Vdash \varphi \rangle &\iff \forall w \in (\!|\Pi \fatsemi \varepsilon|\!),\ w \models \varphi \iff \forall w \in (\!|\Pi|\!),\ w \models \varphi \\ &\iff \mathcal{R}, \Pi \models \varphi . \end{aligned}$$

Without further ado, we can begin to state and prove a few of the simplest translation rules. All rules given hereafter are theorems. We start with the simplest possible rule that can be given.

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \top \rangle}{\top} \qquad (\top)$$

Proof. We have by definition $\langle \Pi \fatsemi \sigma \Vdash \top \rangle \Leftrightarrow \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \top \Leftrightarrow \top$.

Although simple, this rule proves useful later on (namely, in stable cases of rule $(\Box\, \hbar)$[p83]). Dealing with $\bot$ is both more delicate and less useful, so we leave it for the end.

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash X \wedge Y \rangle}{\langle \Pi \fatsemi \sigma \Vdash X \cap Y \rangle} \qquad (\wedge_X)$$

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash X \vee Y \rangle}{\langle \Pi \fatsemi \sigma \Vdash X \cup Y \rangle} \qquad (\vee_X)$$

Proof. This is a simple application of the semantics of $\wedge$ and X:

$$\begin{aligned} &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \wedge Y \\ \iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \ \wedge\ w \models Y \\ \iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \neq \lambda \ \wedge\ w(1) \in X \ \wedge\ w(1) \in Y \end{aligned}$$

$$\iff \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \neq \lambda \ \wedge\ w(1) \in X \cap Y$$
$$\iff \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \cap Y\,.$$

Therefore by combining the empty and non-empty cases we obtain

$$\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \wedge Y \iff \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \cap Y\,,$$

which proves rule $(\wedge_X)$. Rule $(\vee_X)$[p74] is proven in the exact same way by substituting $\vee$ for $\wedge$ and $\cup$ for $\cap$. ∎

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \varphi \wedge \psi \rangle}{\langle \Pi \fatsemi \sigma \Vdash \varphi \rangle \ \wedge\ \langle \Pi \fatsemi \sigma \Vdash \psi \rangle} \qquad (\wedge)$$

Proof. Conjunction distributes straightforwardly over the universal quantifier:

$$\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \varphi \wedge \psi \iff \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \varphi \wedge w \models \psi$$
$$\iff (\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \varphi) \wedge (\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \psi)\,.$$ ∎

As pointed out in Sec. 4.2.1[p59], disjunction does not enjoy the same privileges, and all we can do *in general* is state a very crude under-approximated translation rule.

$$\uparrow \frac{\langle \Pi \fatsemi \sigma \Vdash \varphi \vee \psi \rangle}{\langle \Pi \fatsemi \sigma \Vdash \varphi \rangle \vee \langle \Pi \fatsemi \sigma \Vdash \psi \rangle} \qquad (\vee_\uparrow)$$

Proof. We have trivially:

$$\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \varphi \quad \vee \quad \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \psi$$
$$\implies \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \varphi \vee \psi\,.$$ ∎

Please note however – and this cannot be over-emphasised – that $(\vee_\uparrow)$ is a shockingly coarse under-approximation which will not be made use of in this work, that it was only given here for the sake of completeness, and that it should only be considered as marginally better than nothing. Furthermore, there are some cases in which disjunction *can* be translated exactly; we have seen one such case already (albeit admittedly a very trivial one) in rule $(\vee_X)$, and two more useful cases will be introduced by the next few rules. Therefore, while seeking the next translation rule to apply in a derivation, $(\vee_\uparrow)$ should only be selected as the absolute last resort.

Recall that it was mentioned in Sec. 4.2.3[p62] that disjunction did not need to be handled in signatures at all, as this was best left to a translation rule. Specifically, disjunction in antecedents is handled by the following rule:

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash [\varphi \vee \varphi'] \Rightarrow \psi \rangle}{\langle \Pi \fatsemi \sigma \Vdash \varphi \Rightarrow \psi \rangle \ \wedge\ \langle \Pi \fatsemi \sigma \Vdash \varphi' \Rightarrow \psi \rangle} \qquad (\vee^{\Rightarrow}_{\wedge})$$

Proof. The result rests on the tautology $\big([\varphi \vee \varphi'] \Rightarrow \psi\big) \Leftrightarrow (\varphi \Rightarrow \psi) \wedge (\varphi' \Rightarrow \psi)$. The detailed steps, easy and very similar to previous proofs, are omitted. ∎

Note that neither ($\vee^{\Rightarrow}_{\wedge}$) nor similar rules based on tautologies – ($\vee_{X}$), ($\vee^{\neg}_{\Rightarrow}$) – are strictly necessary for the translation: the transformation could be done independently. They are nevertheless well worth a mention because they point out both limits and common modi operandi of the translation process. With this, and barring base cases and the many other obvious tautology-based rules which we are not going to state (commutation etcetera), we have exhausted the supply of translation rules which do not act on their signatures. The next rule, called the *rule of signature introduction*, is essential to any non-trivial derivation, and rests upon the definition of $\xi(\cdot)$ given in the previous section.

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \varphi \Rightarrow \psi \rangle \qquad \varphi \in \mathcal{A}\text{-LTL}}{\langle \Pi \fatsemi \sigma \otimes \xi(\varphi) \Vdash \psi \rangle} \tag{$\Rightarrow_{\Sigma}$}$$

Proof. We use the main property (4.4)[p64] of $\xi(\cdot)$ (cf. Theorem 4.15[p72]), as well as the (reverse) finite product-breaking Lemma 4.7[p66].

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models (\varphi \Rightarrow \psi) \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ (w \models \varphi) \Rightarrow (w \models \psi) \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \in (\!|\Pi \fatsemi \xi(\varphi)|\!) \Rightarrow w \models \psi \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!) \cap (\!|\Pi \fatsemi \xi(\varphi)|\!),\ w \models \psi \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma \otimes \xi(\varphi)|\!),\ w \models \psi\ .
\end{aligned}$$

The signature-introduction rule makes it possible to handle disjunction in some more cases, as the next rule will show.

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \varphi \vee \psi \rangle \qquad \exists \bar{\varphi} \in \mathcal{A}\text{-LTL} : \bar{\varphi} \Leftrightarrow \neg\varphi}{\langle \Pi \fatsemi \sigma \Vdash \bar{\varphi} \Rightarrow \psi \rangle} \tag{$\vee^{\neg}_{\Rightarrow}$}$$

Proof. Rests on a tautology: $\varphi \vee \psi \iff \neg\varphi \Rightarrow \psi$.

While rule ($\vee^{\neg}_{\Rightarrow}$) is technically trivial, it has the merit of clearly showing the importance of the form in which a temporal formula is given, as mentioned at the beginning of Sec. 4.2.2[p61]. The best form to use is any form that allows an exact translation – there may be several.

We have now run out of translation rules that we can state and prove easily using only previously established results and definitions. Recall to mind the discussion of "strength" and "weakness" of translations in Sec. 4.2.1[p59]; it was then said that this necessary bookkeeping would be handled by signatures. However, Sec. 4.2.1[p59] focused on translating antecedents and did not broach the subject; the use of signatures to handle the "mode of translation" (i.e. weak or strong) makes their intuitive meaning less obvious but, as we shall see, it comes out pretty naturally in the computations[(b)]. Let us see what weakness or strength of context mean as far as signatures are concerned, by considering the least invasive operators that introduce a new context: the weak- and strong-next operators of level zero, $\circ^0$ and

[(b)] In [Héam, Hugot & Kouchnarenko, 2012a] we used weak and strong intertwined semantics to keep track of the mode of translation, however this did not prevent weak and strong aspects from emerging in signatures. The translation was made simpler and more direct by removing them.

$\bullet^0$. If σ is the signature of some formula φ, then by definition of the weak and strong right shifts and by Lem. 4.9[p68] and 4.10, we have:

$$\xi\big(\circ^0\varphi\big) = \lceil \partial\sigma \mid \{0\} \cup \nabla\sigma \rfloor \quad \text{and} \quad \xi\big(\bullet^0\varphi\big) = \lceil \partial\sigma \mid \nabla\sigma \setminus \{0\} \rfloor \ .$$

In other words, what the context really changes – from the point of view of the signatures – is whether or not $0 \in \nabla\sigma$. We shall refer to a signature σ as *weak* if $0 \in \nabla\sigma$, and *strong* if $0 \notin \nabla\sigma$. This terminology is consistent with the operators of the above equations, and with the view that stronger signatures contain more information, i.e. define stronger constraints and reject more rewrite words. We shall now argue that the mode of translation should *mirror* the quality of the signature in the current translation block. That is to say, given a block $\langle \Pi \fatsemi \sigma \Vdash \varphi \rangle$, the translation of φ should be strong if σ is weak, and weak if σ is strong. Thus we shall transition from a strong translation mode (or "strong context") to a weak mode (or "weak context") by "strengthening" the current translation block's signature: if σ is a signature, then $\star\sigma = \lceil \partial\sigma \mid \nabla\sigma \setminus \{0\} \rfloor$ is its *strengthening*. Let us apply this reasoning on the atomic cases of our translation: the literals $\neg X$ and X, starting with the former; translating $\neg X$ is exactly like translating $\mathcal{R} \setminus X$ in a weak context.

$\star\sigma$: *strengthening of* σ

$$\updownarrow \ \frac{\langle \Pi \fatsemi \sigma \Vdash \neg X \rangle}{\langle \Pi \fatsemi \star\sigma \Vdash \mathcal{R} \setminus X \rangle} \qquad (\neg X)$$

Proof. The proof rests on the equality between the sets $(\!|\Pi \fatsemi \star\sigma|\!)$ and $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{>0}$, which is justified by the definition of $(\!|\Pi \fatsemi \sigma|\!)$.

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \neg X \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ 1 \in \operatorname{dom} w \Rightarrow w(1) \notin X \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ \#w > 0 \Rightarrow w(1) \in \mathcal{R} \setminus X \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ 0 \notin \nabla\sigma \Rightarrow (1 \in \operatorname{dom} w \wedge w(1) \in \mathcal{R} \setminus X) \\
\iff &\forall w \in (\!|\Pi \fatsemi \star\sigma|\!),\ w \models \mathcal{R} \setminus X \ .
\end{aligned}$$

∎

The case of the atom is actually passably more complicated, and we leave it for the very last. We now deal with the translation of the weak next operator; the simplest way to approach this is to recall the semantics of $w \models \circ^m\varphi$ in terms of suffixes, that is to say $w \models \circ^m\varphi \iff w^{1+m} = \lambda \ \vee \ w^{1+m} \models \varphi$. This suggests that, if the set of $(1+m)$-suffixes can be expressed as a set of constrained maximal rewrite words $(\!|\Pi' \fatsemi \sigma'|\!)$, then we shall simply need to ensure that those suffixes satisfy φ. In other words, the translation will be of the form $\langle \Pi' \fatsemi \sigma' \Vdash \varphi \rangle$. The initial language Π' is immediately determined: $(1+m)$-suffixes are obtained after m rewriting steps from Π, performed according to σ. This is a very common pattern, and deserves a compact notation.

Signature Iteration. Let $\Pi \subseteq \mathcal{T}$ a language, and $\sigma \in \Sigma$ a signature; then for $n \in \mathbb{N}$ we let $\Pi^n_\sigma = \sigma[n]\big(\sigma[n-1](\cdots\sigma[1](\Pi)\cdots)\big)$ be the n-*iteration* of the signature σ. More formally, it is defined recursively such that $\Pi^0_\sigma = \Pi$ and $\Pi^{n+1}_\sigma = \sigma[n+1](\Pi^n_\sigma)$.

Π^n_σ: *iteration of* σ, n *times*

With this notation, we have the initial language $\Pi' = \Pi^m_\sigma$. As for the signature σ', it is intuitively obtained by an operation which is dual to the right shifts seen in the previous section: on each step the leftmost constraints of σ are "consumed" into the language. Naturally, we call this operation the *left shift*.

arithmetic overloading, $-$

ARITHMETIC OVERLOADING. We overload the operator $-$ on the profile $\wp(\overline{\mathbb{N}}) \times \mathbb{N} \to \wp(\overline{\mathbb{N}})$ such that, for any $S \in \wp(\overline{\mathbb{N}})$ and $n \in \mathbb{N}$, we have

$$S - n = \{ k - n \mid k \in S \} \cap \overline{\mathbb{N}} .$$

$\sigma \triangleleft m$: *weak m-left shift of σ*

There is no strong left shift.

SHIFT LEFT. Let $\sigma \in \Sigma$, $m \in \mathbb{N}$, then we define the *m-left shift of σ* as

$$\sigma \triangleleft m = \wr \partial\sigma(m+1), \dots, \partial\sigma(\#\sigma) \fatsemi \partial\sigma(\omega) \mid \nabla\sigma - m \int .$$

Note that we have, in a fashion dual to right shifts, the property that for all $m \in \mathbb{N}$ and all $k \in \mathbb{N}_1$, $(\sigma \triangleleft m)[k] = \sigma[k+m]$. However, this time there is no need to define weak and strong versions; instead, the strengthening star will be used whenever needed. One could nevertheless write the strong left shift as $\sigma \blacktriangleleft m = \star(\sigma \triangleleft m)$, as in [Héam, Hugot & Kouchnarenko, 2012a].

Example: Let $\sigma = \wr X, Y \fatsemi Z \mid \mathbb{N}_2 \int$; then $\sigma \blacktriangleleft 1 = \sigma \triangleleft 1 = \wr Y \fatsemi Z \mid \mathbb{N}_1 \int$. ♢

Our earlier intuition about $(1+m)$-suffixes can now be formalised into the next lemma; note however the condition on the length of the words, which was not discussed above. It is a technicality: recall that by definition of suffixes, $w^{\#w+1} = w^{\#w+2} = w^{\#w+3} = \cdots = \lambda$. It is therefore necessary to exclude too-short words, otherwise the empty word would have to appear in $(\!|\Pi_\sigma^m \fatsemi \sigma \triangleleft m|\!)$ not only when a term of Π_σ^m cannot be rewritten, but also if any term of some Π_σ^n, $n < m$, could not be rewritten. This of course would be contrary to our definition of rewrite words.

∀ Lemma 4.17: Shifting Suffixes

Let σ be a signature and $\Pi \subseteq \mathcal{T}$ a language; then

$$(\!|\Pi_\sigma^m \fatsemi \sigma \triangleleft m|\!) = \left\{ w^{m+1} \;\middle|\; w \in (\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant m} \right\} .$$

Proof. **(1 : $\subseteq$)** Let $x \in (\!|\Pi_\sigma^m \fatsemi \sigma \triangleleft m|\!)$. There exists $u_m \in \Pi_\sigma^m$ such that $u_m \xrightarrow{x(1)} u_{m+1} \xrightarrow{x(2)} u_{m+2} \xrightarrow{x(3)} \cdots$, and either $u_{m+\#x} \notin \mathcal{R}^{-1}(\mathcal{T})$ or x is infinite. By definition of Π_σ^m, there exist $u_0, \dots, u_{m-1} \in \mathcal{T}$ such that $u_0 \in \Pi_\sigma^0 = \Pi, \dots, u_{m-1} \in \Pi_\sigma^{m-1}$ and $\rho_1, \dots, \rho_m \in \mathcal{R}$ such that $\rho_1 \in \sigma[1], \dots, \rho_m \in \sigma[m]$ and $u_0 \xrightarrow{\rho_1} u_1 \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_m} u_m$. Let us consider the word $w = \rho_1 \dots \rho_m x$; its length is $\#w = \#x + m$ and $\#x \in \nabla(\sigma \triangleleft m) = \nabla\sigma - m$, thus $\#w \in (\nabla\sigma - m) + m = \nabla\sigma \setminus [\![0, m-1]\!]$. Furthermore, for all $k \in [\![1, m]\!]$, we have by construction $w(k) = \rho_k \in \sigma[k]$, and for all $k \in [\![m+1, \#w]\!]$, $w(k) = x(k-m)$. By definition of $x \in (\!|\Pi_\sigma^m \fatsemi \sigma \triangleleft m|\!)$, for all $i \in \operatorname{dom} x$, $x(i) \in (\sigma \triangleleft m)[i] = \sigma[i+m]$, thus for all $k \in [\![m+1, \#w]\!]$, $w(k) = x(k-m) \in \sigma[k-m+m] = \sigma[k]$. So we have that for all $k \in \operatorname{dom} w$, $w(k) \in \sigma[k]$. Thus we have built a word $w \in (\!|\Pi \fatsemi \wr \partial\sigma \mid \nabla\sigma \setminus [\![0, m-1]\!] \int|\!)$ such that $w^{m+1} = x$. There only remains to remark that $w \in (\!|\Pi \fatsemi \wr \partial\sigma \mid \nabla\sigma \setminus [\![0, m-1]\!] \int|\!) \iff w \in (\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant m}$, and we can conclude this part. **(2 : $\supseteq$)** Let $x \in \left\{ w^{m+1} \;\middle|\; w \in (\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant m} \right\}$, and let $w \in (\!|\Pi \fatsemi \sigma|\!)$ such that $x = w^{m+1}$; by the same type of immediate arguments as for (1), $x \in (\!|\Pi_\sigma^m|\!)$. For all $k \in \operatorname{dom} w$, $w(k) \in \sigma[k]$, so for all $k \in \operatorname{dom} x$, $x(k) = w^{m+1}(k) = w(k+m) \in \sigma[k+m] = (\sigma \triangleleft m)[k]$. As

above, we have $\#w \in \nabla\sigma \setminus [\![0, m-1]\!]$, and since $\#x = \#w - m$, it follows that $\#x \in (\nabla\sigma \setminus [\![0, m-1]\!]) - m = \nabla\sigma - m = \nabla(\sigma \lhd m)$. Thus $x \in (\!|\Pi^m_\sigma \fatsemi \sigma \lhd m|\!)$. ∎

As announced, the translation rule is a forthright corollary of this lemma:

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle}{\langle \Pi^m_\sigma \fatsemi \star(\sigma \lhd m) \Vdash \varphi \rangle} \qquad (\circ^m)$$

Proof. We use Lemma 4.17[p78] in the third step; in this context the condition $\#w \geqslant m$ can be omitted because we only deal with cases where $\#w^{1+m} \geqslant 1$.

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \circ^m \varphi \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ \#w^{1+m} \geqslant 1 \implies w^{1+m} \models \varphi \\
\iff &\forall x \in \{w^{m+1} \mid w \in (\!|\Pi \fatsemi \sigma|\!)\},\ \#x \geqslant 1 \implies x \models \varphi \\
\iff &\forall x \in (\!|\Pi^m_\sigma \fatsemi \sigma \lhd m|\!),\ \#x \geqslant 1 \implies x \models \varphi \\
\iff &\forall x \in (\!|\Pi^m_\sigma \fatsemi \star(\sigma \lhd m)|\!),\ x \models \varphi\ .
\end{aligned}$$ ∎

Dealing with the strong next operator is not much more difficult, as its semantics can be expressed in terms of that of its weaker counterpart: $w \models \bullet^m \varphi \iff \#w > m \wedge w \models \circ^m \varphi$. The only novelty here is the condition $\#w > m$, which will be translated by excluding smaller lengths.

∀ Lemma 4.18

Let σ be a signature and $\Pi \subseteq \mathcal{T}$ a language; then for any $m \in \mathbb{N}$, it holds that $(\!|\Pi \fatsemi \sigma|\!)^\#_m = \varnothing$ iff $m \in \nabla\sigma \implies \Pi^m_\sigma \subseteq \mathcal{R}^{-1}(\mathcal{T})$.

Proof. **(1 : ⇒)** Suppose that $(\!|\Pi \fatsemi \sigma|\!)^\#_m = \varnothing$, and let $m \in \nabla\sigma$ such that $\exists u_m \in \Pi^m_\sigma : u_m \notin \mathcal{R}^{-1}(\mathcal{T})$. By definition of Π^m_σ, there exist $u_0, \ldots, u_{m-1} \in \mathcal{T}$ such that $u_0 \in \Pi^0_\sigma = \Pi, \ldots, u_{m-1} \in \Pi^{m-1}_\sigma$ and $\rho_1, \ldots, \rho_m \in \mathcal{R}$ such that $\rho_1 \in \sigma[1], \ldots, \rho_m \in \sigma[m]$ and $u_0 \xrightarrow{\rho_1} u_1 \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_m} u_m$. The condition $u_m \notin \mathcal{R}^{-1}(\mathcal{T})$ is equivalent to $\mathcal{R}(\{u_m\}) = \varnothing$, thus the rewrite word $w = \rho_1 \ldots \rho_m$ is maximal: $w \in (\!|\Pi|\!)$. Furthermore, for all $k \in \operatorname{dom} w$, $w(k) = \rho_k \in \sigma[k]$, and $\#w = m \in \nabla\sigma$, thus it satisfies σ, and we have $w \in (\!|\Pi \fatsemi \sigma|\!)$, and therefore $w \in (\!|\Pi \fatsemi \sigma|\!)^\#_m$, which is a contradiction. **(2 : ⇐)** Conversely, suppose that $w \in (\!|\Pi \fatsemi \sigma|\!)^\#_m$, then by definition of constrained words we must have $m \in \nabla\sigma$, and there must exist $u_0 \in \Pi$ and $u_m \in \Pi^m_\sigma$ such that $u_0 \xrightarrow{w} u_m$ and $\mathcal{R}(\{u_m\}) = \varnothing$. This contradicts $u_m \in \Pi^m_\sigma \subseteq \mathcal{R}^{-1}(\mathcal{T})$. ∎

△ Corollary 4.19: Length Rejection

Let $S \in \wp(\overline{\mathbb{N}})$, $\sigma \in \Sigma$ and $\Pi \subseteq \mathcal{T}$; it holds that $(\!|\Pi \fatsemi \sigma|\!)^\#_S = \varnothing$ iff $\bigwedge_{m \in S \cap \nabla\sigma} \Pi^m_\sigma \subseteq \mathcal{R}^{-1}(\mathcal{T})$ iff $(\bigcup_{m \in S \cap \nabla\sigma} \Pi^m_\sigma) \subseteq \mathcal{R}^{-1}(\mathcal{T})$.

Proof. The second equivalence is trivial; the first follows from Lemma 4.18[p79]:

$$\begin{aligned}
&\textstyle\bigwedge_{m \in S \cap \nabla\sigma} \left[\Pi^m_\sigma \subseteq \mathcal{R}^{-1}(\mathcal{T})\right] \\
\iff &\textstyle\bigwedge_{m \in S} \left[m \in \nabla\sigma \Rightarrow \Pi^m_\sigma \subseteq \mathcal{R}^{-1}(\mathcal{T})\right] \\
\iff &\textstyle\bigwedge_{m \in S} \left[(\!|\Pi \fatsemi \sigma|\!)^\#_m = \varnothing\right] \iff (\!|\Pi \fatsemi \sigma|\!)^\#_S = \varnothing\ .
\end{aligned}$$ ∎

Given those results, the following translation can quickly be proven:

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \bullet^m \varphi \rangle}{\langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle \ \wedge \bigwedge_{k \in [\![0,m]\!] \cap \nabla\sigma} \Pi_\sigma^k \subseteq \mathcal{R}^{-1}(\mathcal{T})} \qquad (\bullet^m)$$

Proof. Using Corollary 4.19:

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \bullet^m \varphi \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \circ^m \varphi \ \wedge\ \#w > m \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models \circ^m \varphi \ \wedge\ \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ \#w > m \\
\iff &\langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle \ \wedge\ \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ \#w > m \\
\iff &\langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle \ \wedge\ (\!|\Pi \fatsemi \sigma|\!)^\#_{[\![0,m]\!]} = \varnothing \\
\iff &\langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle \ \wedge\ \textstyle\bigwedge_{k \in [\![0,m]\!] \cap \nabla\sigma} \Pi_\sigma^k \subseteq \mathcal{R}^{-1}(\mathcal{T})\ .
\end{aligned}$$

The penultimate rules concern the $\Box$ operator; the approach that we shall follow is quite similar to that of Sec. 4.2.3 – see (4.5)[p68] – in that it is kicked off by Lem. 4.1[p57] and handling of conjunction —in this case, through rule ($\wedge$). We start by writing

$$\langle \Pi \fatsemi \sigma \Vdash \Box\, \varphi \rangle \Leftrightarrow \left\langle \Pi \fatsemi \sigma \Vdash \bigwedge_{m=0}^{\infty} \circ^m \varphi \right\rangle \Leftrightarrow \bigwedge_{m=0}^{\infty} \langle \Pi \fatsemi \sigma \Vdash \circ^m \varphi \rangle\ ,$$

and, after application of rule ($\circ^m$), we have

$$\langle \Pi \fatsemi \sigma \Vdash \Box\, \varphi \rangle \Leftrightarrow \bigwedge_{m=0}^{\infty} \langle \Pi_\sigma^m \fatsemi \star(\sigma \lhd m) \Vdash \varphi \rangle\ ,$$

but this falls short of a usable translation, at least in its current, infinite form. The usual intuition to overcome this kind of infiniteness is to hope for some kind of fixed point to be reached, which would give licence for the infinite conjunction to be trivially pared into a finite one. Unfortunately, this is not the case here, since there is no reason in general to expect that there should exist some h such that $\Pi_\sigma^h = \Pi_\sigma^{h+1}$. On the other hand, it seems reasonable that the signature may stabilise at some point, that is to say we may find some h such that $\star(\sigma \lhd h) = \star(\sigma \lhd (h+1))$, or more simply, h' such that $\sigma \lhd h' = \sigma \lhd (h'+1)$, and this may by itself provide sufficient ammunition to express the conjunction more compactly.

stable signature

STABILITY. A signature $\sigma \in \Sigma$ is called *left-stable*, or simply *stable*, if it satisfies the following three, equivalent conditions:

(1) $\sigma \lhd 1 = \sigma$,

(2) $\forall n \in \mathbb{N},\ \sigma \lhd n = \sigma$,

(3) $\#\sigma = 0 \ \wedge\ \nabla\sigma \in \{\varnothing, \{+\infty\}, \mathbb{N}, \overline{\mathbb{N}}\}$.

By definition of the left shift, it is immediate that $(\sigma \lhd n) \lhd 1 = \sigma \lhd (n+1)$, and the first equivalence (1) $\Leftrightarrow$ (2) follows. The implication (3) $\Rightarrow$ (1) is also a simple application of the definition. To obtain the converse (1) $\Rightarrow$ (3), notice that if $\#\sigma > 0$ then $\#\sigma \lhd 1 = \#\sigma - 1 \neq \#\sigma$, but if $\#\sigma = 0$ then $\#\sigma \lhd 1 = \#\sigma = 0$. Similarly, we need

to have

$$\begin{aligned}\nabla\sigma - 1 = \nabla\sigma &\iff \forall n \in \overline{\mathbb{N}},\quad n \in \nabla\sigma - 1 \Leftrightarrow n \in \nabla\sigma \\ &\iff \forall n \in \overline{\mathbb{N}},\quad n + 1 \in \nabla\sigma \Leftrightarrow n \in \nabla\sigma\,.\end{aligned}$$

Since $\infty + 1 = \infty$, we can have either $\infty \in \nabla\sigma$ or $\infty \notin \nabla\sigma$; furthermore, if $0 \in \nabla\sigma$ then $\nabla\sigma \cap \mathbb{N} = \mathbb{N}$, and if $0 \notin \nabla\sigma$ then $\nabla\sigma \cap \mathbb{N} = \varnothing$. All in all, there are only the four possibilities listed in (3).

A stable signature allows for an easy translation of $\Box\,\varphi$, which can be stated as the rule ($\Box_*$). This rule, once proven, will serve as a lemma for the proof of the more general ($\Box_\hbar$), which subsumes it in the translation system. It will still be used in examples when possible, as it is much simpler than ($\Box_\hbar$).

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \Box\,\varphi\rangle \qquad \sigma \text{ is stable}}{\langle \sigma[\omega]^*(\Pi) \fatsemi \star\sigma \Vdash \varphi\rangle} \tag{$\Box_*$}$$

A small intermediary remark is required for a complete proof:

Remark 4.20: Constrained Union

Let $\sigma \in \Sigma$, $I \subseteq \mathbb{N}$, and for each $i \in I$, $\Pi_i \subseteq \mathcal{T}$. Then $\bigcup_{i\in I} (\!|\Pi_i \fatsemi \sigma|\!) = (\!|\bigcup_{i\in I} \Pi_i \fatsemi \sigma|\!)$.

Proof. It is immediate from the definition that we have $\bigcup_{i\in I} (\!|\Pi_i|\!) = (\!|\bigcup_{i\in I} \Pi_i|\!)$. Likewise, we have by definition $(\!|\Pi \fatsemi \sigma|\!) = \{w \in (\!|\Pi|\!) \mid P(w,\sigma)\}$, where $P(w,\sigma)$ is some predicate depending only on w and σ, the details of which are irrelevant for this proof. We have $\bigcup_{i\in I} (\!|\Pi_i \fatsemi \sigma|\!) = \bigcup_{i\in I} \{w \in (\!|\Pi_i|\!) \mid P(w,\sigma)\} = \{w \in \bigcup_{n\in I} (\!|\Pi_i|\!) \mid P(w,\sigma)\} = \{w \in (\!|\bigcup_{n\in I} \Pi_i|\!) \mid P(w,\sigma)\} = (\!|\bigcup_{n\in I} \Pi_i \fatsemi \sigma|\!)$. ∎

Proof of rule ($\Box_*$). Assume that σ is stable.

$$\begin{aligned}&\langle \Pi \fatsemi \sigma \Vdash \Box\,\varphi\rangle \\ \Leftrightarrow &\textstyle\bigwedge_{m=0}^{\infty} \langle \Pi_\sigma^m \fatsemi \star(\sigma \lhd m) \Vdash \varphi\rangle && \Leftrightarrow \textstyle\bigwedge_{m=0}^{\infty} \langle \Pi_\sigma^m \fatsemi \star\sigma \Vdash \varphi\rangle \\ \Leftrightarrow &\forall m \in \mathbb{N},\ \forall w \in (\!|\Pi_\sigma^m \fatsemi \star\sigma|\!);\ w \models \varphi && \Leftrightarrow \forall w \in \textstyle\bigcup_{m=0}^{\infty} (\!|\Pi_\sigma^m \fatsemi \star\sigma|\!);\ w \models \varphi \\ \Leftrightarrow &\forall w \in (\!|\textstyle\bigcup_{m=0}^{\infty} \Pi_\sigma^m \fatsemi \star\sigma|\!);\ w \models \varphi && \Leftrightarrow \forall w \in (\!|\textstyle\bigcup_{m=0}^{\infty} \sigma[\omega]^m(\Pi) \fatsemi \star\sigma|\!);\ w \models \varphi \\ \Leftrightarrow &\forall w \in (\!|\sigma[\omega]^*(\Pi) \fatsemi \star\sigma|\!);\ w \models \varphi && \Leftrightarrow \langle \sigma[\omega]^*(\Pi) \fatsemi \star\sigma \Vdash \varphi\rangle\,.\end{aligned}$$

∎

However, rule ($\Box_*$) is of limited use by itself, as signatures have no particular reason to be stable. In the next paragraphs, we explore whether and how a signature can be stabilised, that is to say how to get a stable signature from an unstable one, and how to employ that to effect the translation of $\Box\,\varphi$ in the general case.

High Point. The *high point* of a signature $\sigma \in \Sigma$, denoted by $\hbar\sigma$, is defined according to either of the following equivalent statements: *high point*

(1) $\hbar\sigma = \min\{h \in \mathbb{N} \mid \sigma \lhd h \text{ is stable}\}$,

(2) $\hbar\sigma = \min\{h \in \mathbb{N}_{\#\sigma} \mid \nabla\sigma \supseteq \mathbb{N}_h \text{ or } \nabla\sigma \cap \mathbb{N}_h = \varnothing\}$.

The equivalence between those two definitions stems from the third characterisation of stability, because the stability of $\sigma \lhd \hbar\sigma$ entails $\#(\sigma \lhd \hbar\sigma) = 0$, which implies $\hbar\sigma \geqslant \#\sigma$, and $\nabla(\sigma \lhd \hbar\sigma) = \nabla\sigma - \hbar\sigma \in \{\varnothing, \{+\infty\}, \mathbb{N}, \overline{\mathbb{N}}\}$, hence $\nabla\sigma \supseteq \mathbb{N}_{\hbar\sigma}$ or $\nabla\sigma \cap \mathbb{N}_{\hbar\sigma} = \varnothing$. Note that since $\sigma \lhd 0 = \sigma$, the high point gives a fourth characterisation of stability

– which is the most convenient one in practice – as σ is stable if and only if $\hbar\sigma = 0$. There remains, however, that not all signatures have a high point; consider the counterexamples $\sigma_1 = \wr\S X \mid \{2k \mid k \in \mathbb{N}\}\int$ or $\sigma_2 = \wr\S X \mid \mathbb{P}\int$, where $\mathbb{P}$ is the set of prime numbers. We take the convention that in those cases $\hbar\sigma = +\infty$, and say that a signature σ is *stabilisable* if $\hbar\sigma \in \mathbb{N}$. It is fortunate that, while all signatures of Σ may not be stabilisable, in practice this is the case for all the signatures we shall need to deal with, as the next lemma will show.

∀ Lemma 4.21: Stability of $\xi(\cdot)$

The signature of any formula $\varphi \in \mathcal{A}$-LTL is stabilisable; in other words, $\hbar\xi(\varphi) \in \mathbb{N}$, $\forall \varphi \in \mathcal{A}$-LTL.

Proof. Recall the definition of $\xi(\cdot)$ given in Thm. 4.15[p72]; we show the result by induction on φ. The base cases are immediate:

$$\hbar\xi(\top) = \hbar\wr\S\mathcal{R} \mid \overline{\mathbb{N}}\int = \hbar\varepsilon = 0 \in \mathbb{N}, \qquad \hbar\xi(X) = \hbar\wr X \S \mathcal{R} \mid \overline{\mathbb{N}}_1\int = 1 \in \mathbb{N},$$
$$\hbar\xi(\bot) = \hbar\wr\S\varnothing \mid \varnothing\int = 0 \in \mathbb{N}, \qquad \hbar\xi(\neg X) = \hbar\wr\mathcal{R} \setminus X \S \mathcal{R} \mid \overline{\mathbb{N}}\int = 1 \in \mathbb{N}\,.$$

For the inductive cases, let us assume that $\hbar\xi(\varphi) \in \mathbb{N}$ and $\hbar\xi(\psi) \in \mathbb{N}$. We start with the weak next: $\hbar\xi(\circ^m\varphi) = \hbar(\xi(\varphi) \rhd m)$; we have $\#(\xi(\varphi) \rhd m) = \#\xi(\varphi) + m$ and therefore

$$\#\big([\xi(\varphi) \rhd m] \lhd [\hbar\xi(\varphi) + m]\big) = \max\{0, \#\xi(\varphi) + m - \hbar\xi(\varphi) - m\}\,.$$

Since $\hbar\xi(\varphi) \geqslant \#\xi(\varphi)$, $\#\big([\xi(\varphi) \rhd m] \lhd [\hbar\xi(\varphi) + m]\big) = 0$. As for the support, we have

$$\begin{aligned}
&\nabla\big([\xi(\varphi) \rhd m] \lhd [\hbar\xi(\varphi) + m]\big)\\
=&\big([\![0, m]\!] \cup (\nabla\xi(\varphi) + m)\big) - (\hbar\xi(\varphi) + m)\\
=&\big[\big([\![0, m]\!] \cup (\nabla\xi(\varphi) + m)\big) - m\big] - \hbar\xi(\varphi)\\
=&\nabla\xi(\varphi) - \hbar\xi(\varphi) = \nabla(\xi(\varphi) \lhd \hbar\xi(\varphi))\,,
\end{aligned}$$

and by induction hypothesis, $\nabla(\xi(\varphi) \lhd \hbar\xi(\varphi)) \in \{\varnothing, \{+\infty\}, \mathbb{N}, \overline{\mathbb{N}}\}$. Therefore, $[\xi(\varphi) \rhd m] \lhd [\hbar\xi(\varphi) + m]$ is stable, and it follows that $\hbar\xi(\circ^m\varphi) = \hbar(\xi(\varphi) \rhd m) \leqslant \hbar\xi(\varphi) + m \in \mathbb{N}$. It is easy (but optional for this proof) to see that we actually have the equality. Moving on to the strong next, we have $\hbar\xi(\bullet^m\varphi) = \hbar(\xi(\varphi) \blacktriangleright m)$; since $\#(\xi(\varphi) \blacktriangleright m) = \#(\xi(\varphi) \rhd m)$, it is immediate that $\#\big([\xi(\varphi) \blacktriangleright m] \lhd [\hbar\xi(\varphi) + m]\big) = 0$, and a fortiori $\#\big([\xi(\varphi) \blacktriangleright m] \lhd [\hbar\xi(\varphi) + m + 1]\big) = 0$. The "+1" will be needed for the support, as this case is somewhat different from the weak next: the same computation as before, using $\hbar\xi(\varphi) + m$, would yield $(\nabla\xi(\varphi) \setminus \{0\}) - \hbar\xi(\varphi)$, which is not enough to conclude. For this reason, we shall use $\hbar\xi(\varphi) + m + 1$:

$$\begin{aligned}
&\nabla\big([\xi(\varphi) \blacktriangleright m] \lhd [\hbar\xi(\varphi) + m + 1]\big) = \big([\nabla\xi(\varphi) \setminus \{0\}] + m\big) - (\hbar\xi(\varphi) + m + 1)\\
=&\big(\big[\big([\nabla\xi(\varphi) \setminus \{0\}] + m\big) - m\big] - 1\big) - \hbar\xi(\varphi) = ([\nabla\xi(\varphi) \setminus \{0\}] - 1) - \hbar\xi(\varphi)\\
=&[\nabla\xi(\varphi) - 1] - \hbar\xi(\varphi) = \nabla([\xi(\varphi) \lhd 1] \lhd \hbar\xi(\varphi)) = \nabla([\xi(\varphi) \lhd \hbar\xi(\varphi)] \lhd 1)\,.
\end{aligned}$$

By the induction hypothesis we have immediately $\nabla([\xi(\varphi) \lhd \hbar\xi(\varphi)] \lhd 1) \in \{\varnothing, \{+\infty\}, \mathbb{N}, \overline{\mathbb{N}}\}$. Finally, $\hbar\xi(\bullet^m\varphi) = \hbar(\xi(\varphi) \blacktriangleright m) \leqslant \hbar\xi(\varphi) + m + 1 \in \mathbb{N}$.

In the case of the product, we have easily $\hbar\xi(\varphi \wedge \psi) = \hbar(\xi(\varphi) \otimes \xi(\psi)) \leqslant \max[\hbar\xi(\varphi), \hbar\xi(\psi)]$. By definition of signature product, $\#[\xi(\varphi) \otimes \xi(\psi)] = \max(\#\xi(\varphi), \#\xi(\psi))$, thus

$$\begin{aligned}&\#([\xi(\varphi) \otimes \xi(\psi)] \lhd \max(\hbar\xi(\varphi), \hbar\xi(\psi)))\\ =&\max\big(0, \max[\#\xi(\varphi), \#\xi(\psi)] - \max[\hbar\xi(\varphi), \hbar\xi(\psi)]\big) = 0\,.\end{aligned}$$

As for the support, we derive

$$\begin{aligned}&\nabla([\xi(\varphi) \otimes \xi(\psi)] \lhd \max(\hbar\xi(\varphi), \hbar\xi(\psi)))\\ =&\nabla(\xi(\varphi) \otimes \xi(\psi)) - \max(\hbar\xi(\varphi), \hbar\xi(\psi))\\ =&(\nabla\xi(\varphi) \cap \nabla\xi(\psi)) - \max(\hbar\xi(\varphi), \hbar\xi(\psi))\\ =&[\nabla\xi(\varphi) - \max(\hbar\xi(\varphi), \hbar\xi(\psi))] \cap [\nabla\xi(\psi) - \max(\hbar\xi(\varphi), \hbar\xi(\psi))]\\ =&[\nabla\xi(\varphi) - \hbar\xi(\varphi)] \cap [\nabla\xi(\psi) - \hbar\xi(\psi)] \in \{\varnothing, \{+\infty\}, \mathbb{N}, \overline{\mathbb{N}}\}\,,\end{aligned}$$

as this four-element set is closed by intersection. There only remains the case of $\hbar\xi(\Box\,\varphi) = \hbar\big(\bigotimes_{m=0}^{\infty}[\xi(\varphi) \rhd m]\big)$ for which we come back to the closed form given at the end of the proof of Lem. 4.13[p71]. The closed form shows that the cardinal remains unchanged by the infinite product: $\#\big(\bigotimes_{m=0}^{\infty}[\xi(\varphi) \rhd m]\big) = \#\xi(\varphi)$, therefore $\hbar\xi(\Box\,\varphi) \geqslant \#\xi(\varphi)$. The closed expression of the support is passably complicated, but can be sufficiently summarised for our purposes by the inclusion

$$\nabla\left(\bigotimes_{m=0}^{\infty}\Big[\xi(\varphi) \rhd m\Big]\right) \subseteq [\![0, N]\!] \cup \{+\infty\}\,,$$

for some $N \in \overline{\mathbb{N}}$. Either N is finite or it is not. If $N = +\infty$, then the support belongs to $\{\mathbb{N}, \overline{\mathbb{N}}\}$, and is stable by any left-shift – in particular by $\max(\hbar\xi(\varphi), N)$. If $N \in \mathbb{N}$, then $\nabla\big(\bigotimes_{m=0}^{\infty}[\xi(\varphi) \rhd m]\big) - \max(\hbar\xi(\varphi), N) \in \{\varnothing, \{+\infty\}\}$. Thus we know that $\hbar\xi(\Box\,\varphi) \leqslant \max(\hbar\xi(\varphi), N) \in \mathbb{N}$. We conclude by a summary of the inductive cases:

$$\begin{aligned}&\hbar\xi(\circ^m\varphi) = \hbar\xi(\varphi) + m \in \mathbb{N} && \hbar\xi(\varphi \wedge \psi) \leqslant \max[\hbar\xi(\varphi), \hbar\xi(\psi)] \in \mathbb{N}\\ &\hbar\xi(\bullet^m\varphi) \leqslant \hbar\xi(\varphi) + m + 1 \in \mathbb{N} && \hbar\xi(\Box\,\varphi) \leqslant \max(\hbar\xi(\varphi), N) \in \mathbb{N}\,.\end{aligned}$$ ■

With this in place, the solution to a general translation of $\Box\,\varphi$ is suggested by the second statement of Lem. 4.1[p57]. The conjunction of weak-next operators needs only be unwound up to a certain, arbitrary rank, and we now know that there always exists a rank beyond which the signature stabilises, and therefore beyond which translation is no longer a problem. Hence we have the following complete rule, which supersedes the less general rule $(\Box_*)$[p81]:

$$\updownarrow\ \frac{\big\langle \Pi \fatsemi \sigma \ \Vdash\ \Box\,\varphi\big\rangle \qquad \hbar\sigma \in \mathbb{N}}{\left\langle \Pi \fatsemi \sigma \ \Vdash\ \bigwedge_{k=0}^{\hbar\sigma-1} \circ^k\varphi \right\rangle \wedge \big\langle \sigma[\omega]^*(\Pi^{\hbar}_{\sigma}\sigma) \fatsemi \star(\sigma \lhd \hbar\sigma) \ \Vdash\ \varphi\big\rangle} \qquad (\Box_{\hbar})$$

Proof. This is a rather direct corollary of Lem. 4.1[p57]'s second statement and of rules $(\Box_*)$[p81], $(\wedge)$[p75], $(\top)$[p74] and $(\circ^m)$[p79]:

$$\begin{aligned}
&\langle \Pi \fatsemi \sigma \Vdash \Box\varphi\rangle \iff \left\langle \Pi \fatsemi \sigma \Vdash \left[\bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi\right] \wedge \circ^{\hbar\sigma} \Box\varphi \right\rangle \\
\iff &\left\langle \Pi \fatsemi \sigma \Vdash \bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi \right\rangle \wedge \left\langle \Pi \fatsemi \sigma \Vdash \circ^{\hbar\sigma} \Box\varphi \right\rangle \\
\iff &\left\langle \Pi \fatsemi \sigma \Vdash \bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi \right\rangle \wedge \left\langle \Pi_\sigma^{\hbar\sigma} \fatsemi \star(\sigma \lhd \hbar\sigma) \Vdash \Box\varphi \right\rangle \\
\iff &\left\langle \Pi \fatsemi \sigma \Vdash \bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi \right\rangle \wedge \left\langle [\star(\sigma \lhd \hbar\sigma)][\omega]^*(\Pi_\sigma^{\hbar\sigma}) \fatsemi \star(\sigma \lhd \hbar\sigma) \Vdash \varphi \right\rangle \\
\iff &\left\langle \Pi \fatsemi \sigma \Vdash \bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi \right\rangle \wedge \left\langle \sigma[\omega]^*(\Pi_\sigma^{\hbar\sigma}) \fatsemi \star(\sigma \lhd \hbar\sigma) \Vdash \varphi \right\rangle .
\end{aligned}$$

Note that rule $(\top)$ is used when $\hbar\sigma = 0$, in which case the conjunction $\bigwedge_{k=0}^{\hbar\sigma-1} \circ^k \varphi$ is simply $\top$. This brings us back to the case of $\bot$, which was mentioned earlier. Unlike $\top$, $\bot$ is never introduced by the rules themselves, since we never have to deal with potentially empty disjunctions. And it is not particularly useful for the user either, since termination can be enforced through other means (the atom $\varnothing$, for instance), and formulæ can easily be simplified beforehand to remove $\bot$ through some basic tautologies. While some such tautologies make for useful translation rules – $(\vee^{\Rightarrow}_{\wedge})$, $(\vee_X)$, $(\vee^{\neg}_{\Rightarrow})$ – this is not the case here. Nevertheless, for the sake of completeness, we give a sketch of what the translation rule would be like. In the following, the map $\xi^{-1}(\cdot) : \Sigma \to \mathcal{A}$-LTL acts as an inverse (up to equivalence) for our signature-builder $\xi(\cdot)$. More specifically, it satisfies the conditions $\xi\big(\xi^{-1}(\sigma)\big) \equiv \sigma$ and $w \models \xi^{-1}(\xi(\varphi)) \Leftrightarrow w \models \varphi$.

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash \bot\rangle}{\langle \Pi \fatsemi \varepsilon \Vdash \neg\xi^{-1}(\sigma)\rangle} \qquad (\bot)$$

Proof. This rests on the first-order tautology $\forall_x, (P(x) \Rightarrow \bot) \Leftrightarrow \forall_x, \neg P(x)$.

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!), \ w \models \bot && \Leftrightarrow \forall w \in (\!|\Pi \fatsemi \sigma|\!), \ \bot \\
\Leftrightarrow &\forall w \in (\!|\Pi|\!), \ w \notin (\!|\Pi \fatsemi \sigma|\!) && \Leftrightarrow \forall w \in (\!|\Pi|\!), \ w \not\models \xi^{-1}(\sigma) .
\end{aligned}$$

Since $(\bot)$ is not actually useful, as said above, there is no need to go to the trouble of explicitly building $\xi^{-1}(\cdot)$, though it is clear that such a map exists. In practice, one can simply "replay" the derivation in reverse order, and reconstitute the original formula through the calls to $\xi(\cdot)$ in instances of $(\Rightarrow_\Sigma)$. In all cases, it should be noted that $\xi^{-1}(\sigma)$ does not necessarily yield a translatable formula, so using $(\bot)$ brings no advantage compared to preprocessing. Thus it remains best to remove $\bot$s before the translation.

The very last case of this section is that of the atom X, and it is by far the trickiest to translate, although this difficulty is mitigated by the reuse of previously established lemmata. Let us start by acquiring some intuition: recall two previous results in the case $\sigma = \varepsilon$:

$$\begin{aligned}
&\langle \Pi \fatsemi \varepsilon \Vdash \neg X\rangle \Leftrightarrow \langle \Pi \fatsemi \star\varepsilon \Vdash \mathcal{R} \setminus X\rangle && (\neg X)_{[p77]}, \ \sigma = \varepsilon \\
&\langle \Pi \fatsemi \varepsilon \Vdash \neg X\rangle \Leftrightarrow X(\Pi) = \varnothing && 1, \text{Sec. } 4.2.1_{[p59]} .
\end{aligned}$$

The substitution of $\mathcal{R} \setminus X$ for X in the right-hand sides of the above equivalences immediately yields the following translation of the atom X in the special but common case when $\sigma = \star\varepsilon$:

$$\langle \Pi \fatsemi \star\varepsilon \Vdash X \rangle \Leftrightarrow [R \setminus X](\Pi) = \varnothing \,. \tag{$X^\star_\varepsilon$}$$

Bearing in mind the first two cases of Sec. 4.2.1[p59], this should look quite familiar – and indeed it is π_2, our first attempt at translating the positive literal. As seen then, an additional condition was needed in order to ensure existence of the transition: $\Pi \subseteq \mathcal{R}^{-1}(\mathcal{T})$. With the additional notions and notations introduced since then, we can couch that in terms of a translation rule:

$$\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash X \rangle}{\langle \Pi \fatsemi \star\varepsilon \Vdash X \rangle \ \wedge \ \Pi \subseteq \mathcal{R}^{-1}(\mathcal{T})} \,, \tag{X_ε}$$

which bears a striking resemblance to another, recently introduced rule: $(\bullet^m)$[p80]. In both cases, a strong translation defers most of the work to its weakened counterpart, and merely adds an existential clause. Another way of seeing the existence of a transition is as the *rejection* of some word lengths, and it is this perspective that we shall adopt. A concrete way of interpreting the above rule is to say that it exchanges the presence of 0 in the support of the signature for a statement rejecting the existence of 0-length maximal rewrite words. As for the "weak" part of the translation, $[R \setminus X](\Pi) = \varnothing$, it excludes all words of length 1 or more that do not start with a rule of X. While this partition of lengths may appear artificial in this case, it becomes more clearly marked in the next example. Let us consider the formula

$$\varphi = X \wedge \bullet^1 Y \wedge \bullet^2 Z \implies A \,.$$

The intuition under its translation is to generate the assertion that any maximal rewrite word which satisfies the antecedent, but does not satisfy the consequent, cannot exist. What would such a word look like? Starting with the initial language Π, it is obtained by successive applications of X, Y, and Z, followed by arbitrary many other applications of any rule in $\mathcal{R}$. Furthermore, its first rule is not in A. In other words, any word built on $Z(Y([X \setminus A](\Pi)))$ would satisfy those criteria. Thus we have the following translation:

$$\pi \equiv Z(Y([X \setminus A](\Pi))) = \varnothing \,.$$

Notice that this excludes only the words of length 3 or more which are built according to the succession $X \setminus A$, Y, Z. It is perfectly possible to have, for instance a maximal word $w = \rho$ of length 1, with $\rho \in X \setminus A$ and $t_0 \in \Pi \xrightarrow{\rho} t_1$. While it may violate the consequent, w does not actually satisfy the antecedent: because of the strong next of level 2 $\bullet^2$, a length of at least 3 is required for that – in terms of support, we have $\overline{\mathbb{N}}_3$. Let us alter the formula φ a little bit and see how it affects the translation; we take

$$\varphi' = X \wedge \circ^1 Y \wedge \circ^2 Z \implies A \,. \tag{4.9}$$

Clearly all the words previously excluded must remain so, but this time the antecedent is more lax in its requirements: words will now satisfy it that did not for φ. Therefore more words may have to be excluded, such as w, which does still violate the consequent, but no longer the antecedent, now compatible with a length

of 1 – the new support is $\overline{\mathbb{N}}_1$. The solution is *not* to write $[X \setminus A](\Pi)$, as it is perfectly acceptable to have a word that starts like w, provided that it can be extended into a word that violates the antecedent. For instance if t_1 can be rewritten, and then only by a rule in $\mathcal{R} \setminus Y$, then the antecedent does not apply. One step farther, we could have $t_0 \xrightarrow{\rho} t_1 \xrightarrow{Y} t_2$, provided that t_2 cannot be rewritten by Z – failing that, there would exist a word that violates φ'. The natural way to translate those ideas is to assume the succession of rules $X \setminus A$, Y, Z, and to systematically reject the lengths that are compatible with the antecedent, yielding the following:

$$\begin{aligned}\pi' \equiv\ & Z(Y([X \setminus A](\Pi))) = \varnothing \\ & \wedge\ [X \setminus A](\Pi) \subseteq \mathcal{R}^{-1}(\mathcal{T})\ \wedge\ Y([X \setminus A](\Pi)) \subseteq \mathcal{R}^{-1}(\mathcal{T})\,. \end{aligned} \tag{4.10}$$

By the same token, supposing now that we had to deal with $\varphi'' = X \wedge \bullet^1 Y \wedge \circ^2 Z \Rightarrow A$, then the translation would instantly come to mind: $\pi'' \equiv Z(Y([X \setminus A](\Pi))) = \varnothing \wedge Y([X \setminus A](\Pi)) \subseteq \mathcal{R}^{-1}(\mathcal{T})$. At this point, the general method has become clear: if, assuming $\neg A$ for the first move, a length is in the support, and the words so built could be extended into something that violates the antecedent, then reject it by enforcing rewritability. However, if no possible extension of the words could violate the antecedent, then assert emptiness of its target language. This does of course raise the question of whether it is at all possible to reach a point where no extension could violate the antecedent, and if so, what that point is. Recalling the discussion of stability made earlier, that is certainly the case if the signature stabilises onto ε. In order to write in a compact way the assumption that the first transition is not by A, we overload the set difference operator $\setminus$ on the profile $\Sigma \times \wp(\mathcal{R}) \to \Sigma$ such that, for any $\sigma \in \Sigma$ and $X \subseteq \mathcal{R}$, we have

$$\sigma \setminus X = \sigma \otimes \xi(\neg X) \equiv \wr \sigma[1] \setminus X, \sigma[2], \sigma[3], \ldots, \sigma[\min(\#\sigma, 1)] \fatsemi \partial\sigma(\omega) \mid \nabla\sigma \int .$$

We can now write the translation rule in the case where stabilisation is done on ε:

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash X \rangle \qquad (\sigma \setminus X) \lhd \hbar(\sigma \setminus X) = \varepsilon}{\Pi^{\hbar(\sigma\setminus X)}_{\sigma\setminus X} = \varnothing \ \wedge \bigwedge_{k \in \nabla\sigma, k=0} \Pi^{k}_{\sigma\setminus X} \subseteq \mathcal{R}^{-1}(\mathcal{T})} \quad \hbar(\sigma\setminus X)-1 \tag{$X_\hbar$}$$

Note that rules ($X^\star_\varepsilon$) and (X_ε) are in fact special cases of the above. The proof of this formula will be done with the help of the following small lemma:

Lemma 4.22

Let $\sigma \in \Sigma$ and $h \in \mathbb{N}$ such that $\sigma \lhd h = \varepsilon$; then $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant h} = \varnothing$ iff $\Pi^h_\sigma = \varnothing$.

Proof. It suffices to characterise that property in terms of h-suffixes:

$$(\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant h} = \varnothing \iff \left\{ w^{h+1} \;\middle|\; w \in (\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant h} \right\} = \varnothing\,.$$

Therefore, all we have to do is invoke Lem. 4.17[p78], and we obtain $(\!|\Pi \fatsemi \sigma|\!)^{\#}_{\geqslant h} = \varnothing \Leftrightarrow (\!|\Pi^h_\sigma \fatsemi \sigma \lhd h|\!) = \varnothing \Leftrightarrow (\!|\Pi^h_\sigma \fatsemi \varepsilon|\!) = \varnothing \Leftrightarrow (\!|\Pi^h_\sigma|\!) = \varnothing \Leftrightarrow \Pi^h_\sigma = \varnothing$. ∎

Proof of rule ($X_\hbar$). A simple characterisation is derived through Lem. 4.7[p66]:

$$\begin{aligned}
&\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \models X \iff \forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \not\models \neg X \\
\iff &\forall w \in (\!|\Pi \fatsemi \sigma|\!),\ w \notin (\!|\Pi \fatsemi \xi(\neg X)|\!) \iff (\!|\Pi \fatsemi \sigma|\!) \cap (\!|\Pi \fatsemi \xi(\neg X)|\!) = \varnothing \\
\iff &(\!|\Pi \fatsemi \sigma \otimes \xi(\neg X)|\!) = \varnothing \iff (\!|\Pi \fatsemi \sigma \setminus X|\!) = \varnothing\,.
\end{aligned}$$

It then becomes possible to reason on the length of the words; most of the work is done by invoking Lem. 4.22[p86] and Cor. 4.19[p79];

$$\begin{aligned}
&(\!|\Pi \fatsemi \sigma \setminus X|\!) = \varnothing \iff \forall k \in \overline{\mathbb{N}},\ (\!|\Pi \fatsemi \sigma \setminus X|\!)^{\#}_{k} = \varnothing \\
\iff &(\!|\Pi \fatsemi \sigma \setminus X|\!)^{\#}_{\geqslant \hbar(\sigma\setminus X)} = \varnothing \ \wedge\ (\!|\Pi \fatsemi \sigma \setminus X|\!)^{\#}_{< \hbar(\sigma\setminus X)} = \varnothing \\
\iff &\Pi^{\hbar(\sigma\setminus X)}_{\sigma\setminus X} = \varnothing \ \wedge\ \bigwedge\nolimits_{k\in\nabla\sigma, k=0}^{k=\hbar(\sigma\setminus X)-1} \Pi^{k}_{\sigma\setminus X} \subseteq \mathcal{R}^{-1}(\mathcal{T})\,.
\end{aligned}$$ ■

Cases when the signature does not stabilise onto ε – which correspond to $\sigma(\omega) \subsetneq \mathcal{R}$ – require an approximated approach to the same extent as a translation of $\Diamond$ does, and thus exceeds the scope of the present discussion. Complete examples of derivations using the rules of this section are given in Sec. 4.5.1[p97].

4.4 Generating an Approximated Procedure

We now have the tools to translate the original verification problem into an equivalent rewrite proposition, at least for a somewhat large gamut of temporal properties. The difference between the original undecidable problem statement in terms of a system and a temporal property, and the no less undecidable reformulation as a rewrite proposition, is that the latter is much more amenable to being transformed into a (possibly approximated) procedure. The reader will have noticed that, in the previous section, the rewrite system $\mathcal{R}$ was seen as a black box, whose particular properties were of no relevance. The same was true of the initial language Π; the focus rested entirely on the temporal property φ, while decidability and representation of the languages involved by automata were ignored. In this section, we focus solely on those aspects.

4.4.1 Juggling Assumptions and Expressive Power

Our objective here is to translate a rewrite proposition π into a decision or positive approximated procedure δ – we call this operation "procedure generation". For the sake of clarity, we attribute a truth value to δ, which is computable and conflated to the result of its execution, that is to say, such that δ is true if the execution of the procedure generates a positive answer, and false if it does not answer. With this convention, we have $\delta \Rightarrow \pi$; this fits into our overall objective because if δ answers positively we can then conclude that π holds. Since π is *at worst* an under-approximated translation, we have in any case $\pi \implies \mathcal{R}, \Pi \models \varphi$; thus a positive answer of δ is enough to conclude that the system satisfies the expected property. More precisely, what will really be generated is not merely a single procedure δ,

but a set of different theorems of the form "under such assumptions on the rewrite system, δ decides (resp. approximates) π".

We shall not use the same pseudo-code notations as in the introduction, as they distract from the similarity of structure between a rewrite proposition and its generated procedures – thus hiding the important differences. Instead, we use almost the same notations. The grammar of rewrite propositions has been given in Sec. 4.1.3[p58]. That of (possibly) approximated procedures is quite similar, but their semantics, although linked, are very different.

$$\delta := \gamma \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \qquad \gamma := \alpha = \varnothing \mid \alpha \subseteq \alpha \qquad X \in \wp(\mathcal{R}) \,.$$
$$\alpha := \Pi \mid \mathcal{T} \mid X(\alpha) \mid X^{-1}(\alpha) \mid X^{(*)}(\alpha) \mid \natural\alpha$$

In the context of a rewrite proposition, the cases $\Pi \mid \mathcal{T} \mid X(\ell) \mid X^{-1}(\ell)$ mean exactly what is written: a tree language. In the context of a procedure, however, the same notation stands for an arbitrary tree automaton that accepts this same language. Two *kinds* of automata are considered in this chapter: vanilla tree automata (TA) and tree automata with global equality constraints (TA$^=$), but the method can certainly be extended to involve other varieties. In the same way, while the cases $\ell = \varnothing \mid \ell \subseteq \ell$ stand for comparisons of languages in rewrite propositions, in the context of procedures the corresponding cases designate either decision or positive approximated procedures for emptiness and inclusion (respectively) of the automata involved.

Actually, the only case where there is an approximation at this level is inclusion when TA$^=$ are involved [Filiot et al., 2008]. There is also the special case $X(\ell) = \varnothing$, where ℓ is a TA$^=$-language, which corresponds to a special algorithm [Courbis et al., 2009, Prp. 6], which will be handled in due time, i.e. during procedure generation, at the end of this section. Note that this overloading of notation – which is unlike [Héam, Hugot & Kouchnarenko, 2012a], where procedure names such as `isEmpty` were used, – does not introduce any ambiguity, so long as the context is kept clear. It also has the advantage of avoiding the introduction of new notations, and of making it visually clearer what transformations the initial rewrite proposition undergoes. This brings us to the cases $X^{(*)}(\alpha) \mid \natural\alpha$, which are unique to positive approximated procedures – note the different stars: $*$ in ℓ versus $(*)$ in α. The crux of the undecidability of π is the potential presence of $X^*(\ell)$; however there are well-known methods which, given a regular language ℓ, compute a TA recognising a regular over-approximation (that is to say, a regular superset) of $X^*(\ell)$, which is what we denote by $X^{(*)}(\alpha)$. We have discussed that at some length in Chapter 3[p41]. As for $\natural\alpha$, which we call here the "constraint relaxation of α", it is simply the underlying TA $\mathrm{ta}(\mathcal{A})$ obtained by removing all the equality constraints from the TA$^=$ α. It is immediate that $\mathcal{L}(\natural\alpha) \supseteq \mathcal{L}(\alpha)$, thus we have a regular over-approximation again – but a very crude one, this time. Relaxations will be avoided inasmuch as possible.

Of course there are implicit sanity rules which must be respected in order for the generated δ to be a valid positive approximated procedure. For instance, $\natural\alpha \subseteq \beta$ positively approximates $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$, but $\alpha \subseteq \natural\beta$ does not. Granted, that would be a silly way of approximating inclusion; it is merely an example to illustrate that one has to be careful to use over-approximations only where it does not break the procedure. This has to be kept in mind during the generation of the procedure.

Note that, as it depends in part upon the kinds of automata involved, it is best to deal with it separately, rather than attempting to incorporate those rules in the grammar of δ.

Note that, since $X \subseteq \mathcal{R}$ is a set of rules, it is also a rewrite system. In Chapter 3[p41], the notation $\mathcal{R}$ was just the default variable name for a rewrite system, while in this chapter $\mathcal{R}$ is a very specific TRS, encoding the behaviour of the program, circuit, protocol,... under consideration. This context should be kept in mind when comparing statements from this chapter and statements from elsewhere in this thesis.

So far we have only touched briefly on the question of the automata representation of a language, but it is central to the decision/approximation phase. The main question is the nature of an automaton α built to accept a language ℓ: could it simply be a TA? must it be a $\mathrm{TA}^=$? must it be something even more expressive? We can require of the user that the input Π be a regular tree language, and there is no question that $\mathcal{T}$ is regular in any case. It is also known that $X^{-1}(\mathcal{T})$ is accepted by a $\mathrm{TA}^=$ [Courbis et al., 2009, Prp. 5], as well as $X(\ell)$, provided that ℓ is regular [Courbis et al., 2009, Prp. 7]. Furthermore, properties of the rewrite systems can favourably influence the expressive power required: for instance, if X is left-linear, then $X^{-1}(\mathcal{T})$ is only a regular language. Moreover, there is a wide variety of properties of a rewrite system X under which $X^*(\ell)$ is actually regular, provided that ℓ itself is regular; refer to Chapter 3[p41] for more information on that topic. Thus we can gain the following insights:

(1) The expressive power required to encode a language ℓ depends on a set of assumptions on the inductive sub-parts of ℓ. Each assumption belongs to one of three possible categories:

  a. the expressive power of a sub-language ℓ (regular, $\mathrm{TA}^=$, or beyond),

  b. linearity or other regularity-preserving properties of a TRS X,

  c. presence of an over-approximation or a positive approximated procedure.

(2) Procedure generation needs to build that set of assumptions inductively.

(3) Actually, in theory different combinations of assumptions may be chosen, which lead to potentially different procedures.

To illustrate this, let us consider $\ell = X(Y(\Pi))$. The sub-language Π is regular by hypothesis; $Y(\Pi)$ requires a $\mathrm{TA}^=$ (in general), thus ℓ cannot be expressed with a $\mathrm{TA}^=$. There are two possible paths: we can either make an assumption, or introduce an approximation. For instance, if we assume that Y preserves regularity through one-step rewriting (e.g. if it is linear), then $Y(\Pi)$ is regular, and it follows that ℓ is accepted by a $\mathrm{TA}^=$. Contrariwise, if we make no such assumption, we can still proceed by computing $\alpha^+ = X(\natural Y(\Pi))$, which is also a $\mathrm{TA}^=$, but then we have $\mathcal{L}(\alpha^+) \supseteq \ell$, and we need to keep in mind (i.e. add to our assumptions) that we are using an over-approximation. When considering the procedure generation in an abstract way, both those paths must be considered; however it is clear that one is preferable to the other. Generally, the path which minimises the use of approximations will be considered the most desirable. This is tantamount to minimising the expressive power required at each step. Of course, if Y does *not* have any nice properties then there is no choice as to which path to take.

We shall deal with the question of expressive power and the related problem of detecting approximations by means of a simple inference system, referred to as "kind inference", working recursively on an automaton α built by generation from a rewrite language ℓ. Let $\mathfrak{K} = \{\mathrm{TA}, \mathrm{TA}^=\}$ be the set of all "kinds" of tree automata considered for the procedure generation. As mentioned before, there is nothing

$\mathfrak{K}$: *kinds of automata*

foreordained in this particular choice: a real-world implementation of procedure generation might incorporate many other kinds of tree automata, so long as they play nicely with rewrite systems. Now let $\mathfrak{P} = \{\,\mathsf{left\text{-}lin}, \mathsf{reg\text{-}pres}, \mathsf{reg\text{-}pres}^*\,\}$ be the set of possible properties of a rewrite system, where left-lin stands for left-linear (regularity-preserving for backwards-rewriting of $\mathcal{T}$), reg-pres is a place-holder for any property or conjunction of properties that entail preservation of regularity through one-step forward rewriting (i.e. linear, etc), and reg-pres* is a similar place-holder for preservation of regularity wrt. reachability – i.e. ground, right-linear & monadic, left-linear & semi-monadic, decreasing, etcetera; see Fig. 3.2 [(c)]. Then we let $\mathfrak{A}$ be the set of all possible assumptions, defined as

$\mathfrak{P}$: properties of a system

$\mathfrak{A}$: assumptions

$$\mathfrak{A} = \{\,\alpha : k \mid k \in \mathfrak{K}\,\} \uplus \{\,X : p \mid X \in \wp(\mathcal{R}), p \in \mathfrak{P}\,\} \uplus \{\,\alpha : +\,\} \uplus \{\,\gamma : +\,\}\,,$$

where α is any automaton-expression and γ any test as defined by the grammar given at the beginning of the section. The four disjoint sets in this definition correspond to the four kinds of assumptions listed earlier. In the case of a test γ, $\gamma : +$ means that it is an approximation instead of an exact test. A *kind-inference rule* is either *simple* or a *chain* of simple rules; a simple rule is of the form

kind-inference rule

$$\Gamma \vdash \Delta \qquad \text{where } \Gamma, \Delta \in \wp(\mathfrak{A})\,,$$

and the meaning that, whenever all the assumptions within Γ hold, then so do all those within Δ. Regarding notations, the sets' braces will be omitted when writing the rules, and the comma is taken to mean set union: e.g. "Γ, Γ', a" means "$\Gamma \cup \Gamma' \cup \{a\}$". A chain-rule is a $\lhd$-separated sequence of simple rules,

$$\Gamma_1 \vdash \Delta_1 \lhd \Gamma_2 \vdash \Delta_2 \lhd \ldots \lhd \Gamma_n \vdash \Delta_n\,,$$

such that the antecedents form a chain wrt. inclusion: $\Gamma_i \subseteq \Gamma_{i+1}$. A chain-rule behaves as one of the simple rules in the chain. Given the assumptions Γ, the rule that applies is $\Gamma_k \vdash \Delta_k$, such that $\Gamma_k \subseteq \Gamma$, and either $k = n$ or $\Gamma_{k+1} \not\subseteq \Gamma$. Chain-rules are useful in the common cases where making some more assumptions – on the left, e.g. about a rewrite system – prevents us from having to make some *other* assumptions – e.g. that such language is over-approximated. Given a set of rules S – simple and chains – and a set of assumptions Γ, *one-step deduction* is written $\Gamma \vdash^1 \Delta$, and holds iff there is in S either a simple rule $\Gamma' \vdash \Delta'$ such that $\Gamma' \subseteq \Gamma$ and $\Delta \subseteq \Delta'$, or a chain-rule whose active simple rule, given Γ, satisfies the same properties. A *deduction*, written $\Gamma_0 \vdash^* \Delta$, can be seen as extending this by reflexivity and transitivity, and is defined as follows:

one-step deduction

deduction

$$\Gamma_0 \vdash^* \Delta \quad \text{iff} \quad \Delta \subseteq \Gamma_0 \;\vee\; \exists \Gamma_1, \ldots, \Gamma_n : \Gamma_n \supseteq \Delta \;\wedge\; \forall k \in [\![1, n]\!],\ \textstyle\bigcup_{i=0}^{k-1} \Gamma_i \vdash^1 \Gamma_k\,.$$

Let us now state the axioms of the kind-inference system, which are simply a summary of the properties that have been informally mentioned earlier. We start by the most immediate ones:

$$\vdash \Pi : \mathrm{TA} \qquad\qquad \vdash \mathcal{T} : \mathrm{TA}\,.$$

The first rule is a practical hypothesis. The second is trivially true. A third basic rule one might want to state is $\alpha : \mathrm{TA} \vdash \alpha : \mathrm{TA}^=$, which would reflect the fact that

[(c)] The practical implementation detects the specific properties that ensures this, which we abstract here. Note that this aspect will keep improving as new regularity-preserving classes of TRS are discovered and implemented.

TA are technically degenerate cases of $\mathrm{TA}^=$ where the set of constraints is empty – in that sense any tree automaton is technically also a $\mathrm{TA}^=$. However, this rule is not included in the system, as the assumption $\alpha : \mathrm{TA}$ is taken to mean that α recognises a tree language that is known to be regular, and therefore we can and do assume that α is in fact a TA, while $\alpha : \mathrm{TA}^=$ means that α accepts a $\mathrm{TA}^=$-language that *may* be regular, but which has no particular reason to be as far as is known, and therefore it must be *assumed* that α is indeed a strict $\mathrm{TA}^=$. With this in mind, we move on to forward rewriting, for which there are two main cases in a chain-rule: either the rewrite system preserves regularity, or it does not, in which case we use a $\mathrm{TA}^=$ [Courbis et al., 2009, Prp. 7]. Forward rewriting cannot a priori be done on a $\mathrm{TA}^=$-language while staying within the allowed kinds of tree automata – TA and $\mathrm{TA}^=$ – therefore there is no rule in that case:

$$\alpha : \mathrm{TA} \vdash X(\alpha) : \mathrm{TA}^= \lhd \qquad\qquad \alpha : \mathrm{TA}, X : \mathsf{reg\text{-}pres} \vdash X(\alpha) : \mathrm{TA} \ .$$

Let us go this this chain rule in detail, and in plain English. If α is a BUTA ($\alpha : \mathrm{TA}$), then we deduce ($\vdash$) that one-step rewriting by an arbitrary $X \subseteq \mathcal{R}$ can be performed, and the result is recognised by a TAGE, and we must assume that a TAGE is strictly needed ($X(\alpha) : \mathrm{TA}^=$). However, under the additional assumption ($\lhd$, $\alpha : \mathrm{TA}$ still assumed) that X preserves regularity through one-step rewriting ($X : \mathsf{reg\text{-}pres}$), we can do better, and deduce instead that the result is accepted by a BUTA ($X(\alpha) : \mathrm{TA}$).

Backwards rewriting is similarly captured by a two-rules chain, hinging on left linearity [Courbis et al., 2009, Prp. 5]:

$$\vdash X^{-1}(\mathcal{T}) : \mathrm{TA}^= \lhd \qquad\qquad X : \mathsf{left\text{-}lin} \vdash X^{-1}(\mathcal{T}) : \mathrm{TA} \ .$$

Note that we do not go to the trouble of dealing with the more general case $X^{-1}(\ell)$, simply because the derivations of Sec. 4.3[p73] are such that no translation rule can yield rewrite propositions that require it. Our first approximated case is constraint relaxation, with two independent rules:

$$\alpha : \mathrm{TA} \vdash \natural\alpha : \mathrm{TA} \qquad\qquad \alpha : \mathrm{TA}^= \vdash \natural\alpha : \mathrm{TA}, \natural\alpha : + \ .$$

By the first rule, relaxing the constraints of a TA (that is to say a $\mathrm{TA}^=$ whose set of constraints is already empty), does not do much; the result still is a TA —the same as before. A well-done procedure generation should avoid such useless relaxations, which means that this rule should never be used in practice and could be omitted without damage. The second rule simply states that relaxing the constraints of a $\mathrm{TA}^=$ results in a TA, and introduces an over-approximation. Note that this second part is only true in general. One might construct special instances of $\mathrm{TA}^=$ where relaxation does not introduce any approximation, for instance if the constraints involve states that are unreachable. However, as far as procedure generation is involved, it *must be assumed* that an over-approximation has taken place, lest false theorems be generated. Note that the two rules are not in competition, as $\alpha : \mathrm{TA}$ and $\alpha : \mathrm{TA}^=$ are contradictory assumptions. The other approximated case is reachability over-approximation, captured by a two-rules chain.

$$\alpha : \mathrm{TA} \vdash X^{(*)}(\alpha) : \mathrm{TA}, X^{(*)}(\alpha) : + \lhd \qquad \alpha : \mathrm{TA}, X : \mathsf{reg\text{-}pres}^* \vdash X^{(*)}(\alpha) : \mathrm{TA} \ .$$

In the general case (first rule), the use of some over-approximation algorithm is required; however in the best case (second rule), it can safely be assumed that no

approximation has taken place, and we have $\mathcal{L}\left(X^{(*)}(\alpha)\right) = X^*(\ell)$. Note that it may be possible in the future to use similar reachability over-approximation techniques for $TA^=$-languages, prompting the addition of a rule of the form $\alpha : TA^= \vdash X^{(*)}(\alpha) : TA^=, X^{(*)}(\alpha) : +$, but there are no such results in the literature as yet.

Lastly, we must not forget to encode the fact that over-approximation propagates through forward rewriting, and contaminates tests:

$$\alpha : + \vdash X(\alpha) : +, (\alpha = \varnothing) : +, (\alpha \subseteq \beta) : + ,$$

which in turn contaminate the entire procedure, confining it to approximation:

$$\gamma : + \vdash (\gamma \vee \gamma') : +, (\gamma \wedge \gamma') : + .$$

Even without any prior approximation, an inclusion test must be a positive approximated procedure if its right-hand side is a $TA^=$, and cannot be a TA; in that case the usual method ($\alpha \cap \beta^c = \varnothing$) cannot apply, since $TA^=$ cannot be complemented [Filiot et al., 2008]. This is expressed by the rule:

$$\beta : TA^= \vdash (\alpha \subseteq \beta) : + . \tag{4.11}$$

This system is not complete – and no implementation of it can be, since it would need to know *all* possible regularity-preserving properties, for instance – but the rules above are quite sufficient for our immediate purposes. Indeed, we can now proceed to write the procedure generation itself. As mentioned before, the generation is to be done inductively on the structure of the input rewrite proposition. However, it should be noted that our intent is to explore all possible paths, and generate a theorem for each of them. More precisely, given a rewrite proposition π, we want to obtain the set of all possible couples Δ, δ, such that $\Delta \subseteq \mathfrak{A}$ and δ is the best (possibly) approximated procedure under the assumptions Δ. By "the best" we mean "minimising the use of approximations"; we do not consider regular under-approximations. Such a couple Δ, δ can be regarded as the theorem "If Δ, then δ positively approximates π.". This differs from an actual implementation in the sense that an implementation would check on-the-fly which property the rewrite system actually satisfies, and then explore that sole branch; this is expressed immediately as a recursive function. Since using functions in this discussion would force us to manipulate sets of answers everywhere, we use a recursive *relation* instead, given as a set of rules. Here is what a *procedure-generation rule* looks like in the most general case:

procedure-generation rule

$$\Gamma \fatsemi [\ell_1 \rightarrowtail \Delta_1, \delta_1; \ell_2 \rightarrowtail \Delta_2, \delta_2; \ldots] \fatsemi P \fatsemi \pi \rightrightarrows \Delta \fatsemi \delta .$$

Let us start by the parts that are not optional: $\Gamma \subseteq \mathfrak{A}$ is a set of assumptions, in practice only pertaining to properties of the rewrite system, i.e. excluding kind and over-approximation assumptions; π is the rewrite proposition (resp. sub-part of a rewrite proposition, such as a language ℓ or a comparison γ) being converted; δ is the corresponding procedure (resp. sub-part of a procedure, such as an automaton α or simple comparison γ), and Δ is the set of assumptions under which δ is constructed. The optional parameter P is a predicate which must be satisfied in order for the rule to apply; it is mostly used for kind inference statements. The optional list of patterns of the form "$\ell_k \rightarrowtail \Delta_k, \delta_k$" between brackets serves to name *possible* recursive calls; the ℓ_k are supposed to be direct sub-components of π, and

the pair Δ_k, δ_k corresponds to any one possible result of the procedure generation for ℓ_k, under the assumptions Γ. The simplest rules need neither of those options:

$$\Gamma \fatsemi \Pi \rightrightarrows \Gamma \fatsemi \Pi \qquad\qquad \Gamma \fatsemi \mathcal{T} \rightrightarrows \Gamma \fatsemi \mathcal{T} \,.$$

In both cases, the language on the left simply becomes the automaton on the right, and no assumption is required or introduced. The case of backwards rewriting is also quite simple, though it requires two rules:

$$\Gamma \fatsemi X^{-1}(\mathcal{T}) \rightrightarrows \Gamma \fatsemi X^{-1}(\mathcal{T}) \qquad\qquad \Gamma \fatsemi X^{-1}(\mathcal{T}) \rightrightarrows \Gamma, X : \mathsf{left\text{-}lin} \fatsemi X^{-1}(\mathcal{T}) \,.$$

In the first rule, no extra assumption is introduced, which means that the resulting automaton will need to be a $\mathrm{TA}^=$ in general. In the second rule, the introduction of the assumption $X : \mathsf{left\text{-}lin}$ means that it will only be a TA. Depending on which rule is chosen, the subsequent derivation will yield different procedures (the first choice may lead to constraints relaxations that are unneeded in the second, for instance), different assumptions, and thus different theorems. This is a very common pattern; in fact, that situation occurs at every point of the generation where the presence or absence of some properties of the rewrite system changes the required kind of the automaton. To avoid writing all the different possible rules manually, we "factor" them by putting such properties between angle-brackets: $\langle p_1, \ldots, p_n \rangle$. A rule where this syntax appears is short for the 2^n rules obtained by choosing all possible subsets of those properties. Thus the last two rules can be written simply as:

$$\Gamma \fatsemi X^{-1}(\mathcal{T}) \rightrightarrows \Gamma, \langle X : \mathsf{left\text{-}lin} \rangle \fatsemi X^{-1}(\mathcal{T}) \,.$$

Forward rewriting needs to be more general than the backwards case, thus the next rules are recursive:

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA} \;\; \fatsemi X(\ell) \rightrightarrows \Gamma, \Delta, \langle X : \mathsf{reg\text{-}pres} \rangle \fatsemi X(\alpha)$$
$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA}^= \fatsemi X(\ell) \rightrightarrows \Gamma, \Delta, \langle X : \mathsf{reg\text{-}pres} \rangle \fatsemi X(\natural\alpha) \,.$$

Translated into plain English, the first rule means that, given the assumptions Γ, and further assuming that the language ℓ, under the same assumptions Γ, can be converted into the automaton α, with the resulting assumptions Δ, and also assuming that it can be deduced from those new assumptions Δ that α can be a simple TA, the language $X(\ell)$ can be represented by the automaton $X(\alpha)$, under the union of our starting assumptions Γ, and the assumptions Δ made during the generation of α. Furthermore, whether or not X is regularity-preserving influences the kind of automaton obtained, and thus leads to different branches. Note that explicitly returning the union $\Gamma \cup \Delta$, as was done here, is not strictly necessary, since we take care that no rule ever removes any assumptions – they can only be added, – and thus $\Delta \supseteq \Gamma$.

Note that $\Delta \vdash^* \alpha$ is a deduction, which may take several steps, and not just the application of a single kind inference rule.

The second rule deals with the case where the automaton α can only be a $\mathrm{TA}^=$; in that case, we need to make an over-approximation by relaxing the constraints of α before forward rewriting. Then again, different branches must be explored depending on whether X is regularity-preserving. We have the same kind of pattern for reachability:

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA} \;\; \fatsemi X^*(\ell) \rightrightarrows \Gamma, \Delta, \langle X : \mathsf{reg\text{-}pres}^* \rangle \fatsemi X^{(*)}(\alpha)$$
$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA}^= \fatsemi X^*(\ell) \rightrightarrows \Gamma, \Delta, \langle X : \mathsf{reg\text{-}pres}^* \rangle \fatsemi X^{(*)}(\natural\alpha) \,.$$

Depending on whether X is regularity-preserving for reachability, $X^{(*)}(\alpha)$ (resp. $X^{(*)}(\natural\alpha)$) will be exact or over-approximated (resp. over- and twice-over–approximated). We are now done with the construction of automata, and move on to the generation of tests γ, starting with inclusion.

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha; \ell' \rightarrowtail \Delta', \alpha'] \fatsemi \Delta' \not\vdash^* \alpha' : + \fatsemi \ell \subseteq \ell' \rightrightarrows \Gamma, \Delta, \Delta' \fatsemi \alpha \subseteq \alpha' \,. \tag{4.12}$$

Since this is the first rule with multiple recursive call patterns, let us take the time to translate it into plain English. We generate a positive approximated procedure for the language inclusion $\ell \subseteq \ell'$. To do so, we start by computing the automata accepting those languages, using the generation rules which we have already seen, and under the assumptions Γ. We obtain two automata α and α', and a bunch of assumptions Δ and Δ' about them. If, based on the way in which α' has been generated, we need not assume that it only captures an over-approximation of ℓ' ($\Delta' \not\vdash^* \alpha' : +$), then we can proceed and invoke a decision or positive approximated procedure for containment between α and α', and know that this approximates $\ell \subseteq \ell'$. The output is this procedure, along with everything we know so far, that is to say the assumptions Γ with which we started, and the deductions which were made in the recursive calls, Δ and Δ'. Note that the rule does not apply if $\Delta' \vdash^* \alpha' : +$, because if $\ell'' = \mathcal{L}(\alpha') \supseteq \ell'$, then even if we could decide $\alpha \subseteq \alpha'$, we would know at best that $\ell \subseteq \ell''$, from which it does not follow that $\ell \subseteq \ell'$. Fortunately, no translation rule actually generates rewrite propositions such that ℓ' cannot be captured by a TAGE, so this is a moot point; nevertheless, the generation rule would be wrong without this caveat. Furthermore, recall that if $\alpha' : \mathrm{TA}^=$, then $(\alpha \subseteq \alpha') : +$ will be deduced by the other rules regarding approximations, specifically $(4.11)_{[p92]}$, hence there is no call to pay it mind here.

The only other kind of simple test is emptiness testing:

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA} \vee \Delta \vdash^* \alpha : \mathrm{TA}^= \fatsemi \ell = \varnothing \rightrightarrows \Gamma, \Delta \fatsemi \alpha = \varnothing \,.$$

This is the immediate case: an automaton α accepting ℓ is built, and regardless of whether α has constraints or not, an emptiness test is run on it; the algorithmic complexity changes (ExpTime vs linear time), but this is a decision procedure in both cases. There is another case, where it is possible to test emptiness of a language *without* being able to build the corresponding automaton [Courbis et al., 2009, Prp. 6], given by the next rule.

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA}^= \fatsemi X(\ell) = \varnothing \rightrightarrows \Gamma, \Delta \fatsemi X(\alpha) = \varnothing \,.$$

In that case $X(\alpha)$ cannot be built, since it is not even known whether it is at all recognisable by a $\mathrm{TA}^=$. One possibility, covered by previous rules, is to actually test $X(\natural\alpha) = \varnothing$, which introduces an approximation; but in that case it is a terrible idea, as $X(\ell) = \varnothing$ iff $\ell \cap \mathcal{R}^{-1}(\mathcal{T}) = \varnothing$, which is decidable thanks to $\mathrm{TA}^=$ being closed by intersection. The above rule reflects that fact, and "$X(\alpha) = \varnothing$," where α is a $\mathrm{TA}^=$, is to be taken to abstract the above test. Note that no approximation will be deduced for $X(\alpha)$; in fact, the previous rules simply have nothing to say about $X(\alpha)$, since it is not an automaton of any allowed kind, but merely a notation that only takes meaning in the context of an emptiness test. Again, this could be reflected explicitly in the grammar, at the cost of introducing new symbols. Lastly, let us emphasise that both paths (exact intersection-emptiness test and constraint

relaxation) will be explored by the system, and yield two different theorems. It is easy to discriminate between the two, as only the assumptions generated by the second theorem will enable the deduction of $(X(\alpha) = \varnothing) : +$, and not the first. In a practical implementation, the best option is always chosen if this special rule is applied with a higher priority than the more general forward-rewriting rules. Before concluding, let us not forget the rules for conjunctions and disjunctions of tests, which are trivial given our convention regarding the truth value of a positive approximated procedure, mentioned at the beginning of this section:

$$\Gamma \fatsemi [\gamma \rightarrowtail \Delta, \delta; \gamma' \rightarrowtail \Delta', \delta'] \fatsemi \gamma \wedge \gamma' \rightrightarrows \Gamma, \Delta, \Delta' \fatsemi \delta \wedge \delta'$$
$$\Gamma \fatsemi [\gamma \rightarrowtail \Delta, \delta; \gamma' \rightarrowtail \Delta', \delta'] \fatsemi \gamma \vee \gamma' \rightrightarrows \Gamma, \Delta, \Delta' \fatsemi \delta \vee \delta' \,.$$

A basic implementation in `OCaml` of some of the rules of this section, sufficient to run on a few examples, is available on the web:

`http://lifc.univ-fcomte.fr/~vhugot/RWLTL`.

4.4.2 Optimisation of Rewrite Propositions

A complete translation chain is now defined. However, it is plain that a useful implementation of such a chain should be given every chance to avoid introducing approximations and inflating algorithmic complexity. In light of the previous section, it appears that the best way of doing so is to take every opportunity to notice regularity-preserving properties of the rewrite system —or more accurately, of the "atoms" $X \subseteq \mathcal{R}$ which appear in the rewrite proposition. Let us take a quick look at the concrete properties abstracted by $\mathsf{reg\text{-}pres}, \mathsf{reg\text{-}pres}^*$ and $\mathsf{left\text{-}lin}$: linearity (left/and right), monadicity and semi-monadicity, being ground, being decreasing… All those properties share a common pattern: X satisfies *property of a rewrite system* if all rules of X satisfy *property of a rewrite rule*. Therefore, if $X, Y \subseteq \mathcal{R}$ are two rewrite systems such that $X \subseteq Y$, and Y satisfies one of those properties, so does X. Since it is always in our best interests to manipulate atoms that satisfy as many of these properties as possible, it is advisable that the rewrite proposition given as input to the procedure generation use atoms as small as possible wrt. set inclusion.

The translation rules, such as they have been defined earlier, are not optimal in that respect. Recall for instance the translation of X. In Sec. 4.2.1[p59], the following rewrite proposition was given:

$$[\mathcal{R} \setminus X](\Pi) = \varnothing \ \wedge \ \Pi \subseteq X^{-1}(\mathcal{T}) \,,$$

but the translation rule (X_ε)[p85], which is a particular case of the general rule $(X_\hbar)$[p86], yields instead

$$[\mathcal{R} \setminus X](\Pi) = \varnothing \ \wedge \ \Pi \subseteq \mathcal{R}^{-1}(\mathcal{T}) \,.$$

By the above arguments, X is more likely to be left-linear than $\mathcal{R}$, and therefore, the rewrite proposition generated by the translation rules is actually worse than the handwritten one. Recall the argument for the simplification, given in Sec. 4.2.1[p59]: starting with the second version, and then looking at both conjuncts, we observed that, since $[\mathcal{R} \setminus X](\Pi) = \varnothing$, it follows that $\mathcal{R}^{-1}(\mathcal{T}) = X^{-1}(\mathcal{T})$; only then was the

substitution made. Other similar optimisations could be done, for instance when translating $\Box X$; we gave $[\mathcal{R} \setminus X](\mathcal{R}^*(\Pi)) = \varnothing$ in Sec. 4.2.1[p59], and it turns out that this is also the result of the derivation using rules $(X_\hbar)$[p86] and $(\Box\,\hbar)$[p83]… but we can do better. The above proposition is equivalent to $[\mathcal{R} \setminus X](X^*(\Pi)) = \varnothing$, which is, again, quite preferable from the viewpoint of procedure generation. Could these optimisations be integrated into the translation rules themselves? Let us look at rule $(X_\hbar)$[p86], and take (4.9)[p85]

$$\varphi' = X \wedge \circ^1 Y \wedge \circ^2 Z \implies A \tag{4.9}$$

and its translation (4.10)[p86]

$$\begin{aligned}\pi' \equiv\ & Z(Y([X \setminus A](\Pi))) = \varnothing \\ & \wedge\ [X \setminus A](\Pi) \subseteq \mathcal{R}^{-1}(\mathcal{T})\ \wedge\ Y([X \setminus A](\Pi)) \subseteq \mathcal{R}^{-1}(\mathcal{T})\end{aligned} \tag{4.10}$$

as examples again. An optimisation of the same nature as that made for the translation of X is applicable here: one could advantageously write $Y([X \setminus A](\Pi)) \subseteq [\mathcal{R} \setminus Z]^{-1}(\mathcal{T})$ for the second length-rejection statement. The same reasoning does not extend to the first one, though. In general, the last length-rejection statement can be optimised if the signature-iteration involved in it directly precedes the one appearing in the emptiness statement. Let us recall rule $(X_\hbar)$:

$$\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash X \rangle \qquad (\sigma \setminus X) \lhd \hbar(\sigma \setminus X) = \varepsilon}{\hbar(\sigma\setminus X)-1} \qquad \Pi_{\sigma\setminus X}^{\hbar(\sigma\setminus X)} = \varnothing \ \wedge \bigwedge_{k\in\nabla\sigma, k=0} \Pi_{\sigma\setminus X}^{k} \subseteq \mathcal{R}^{-1}(\mathcal{T}) \tag{$X_\hbar$}$$

Applying the above, we obtain an optimised version of $(X_\hbar)$:

$$\begin{aligned}\updownarrow \frac{\langle \Pi \fatsemi \sigma \Vdash X \rangle \qquad (\sigma \setminus X) \lhd \hbar(\sigma \setminus X) = \varepsilon}{\hbar(\sigma\setminus X)-2} \ .\\ \wedge \bigwedge_{k\in\nabla\sigma, k=0} \Pi_{\sigma\setminus X}^{k} \subseteq \mathcal{R}^{-1}(\mathcal{T}) \\ \wedge \bigwedge_{k\in\nabla\sigma\cap\{\hbar(\sigma\setminus X)-1\}} \Pi_{\sigma\setminus X}^{k} \subseteq \Big[\mathcal{R} \setminus (\sigma \setminus X)\big[\hbar(\sigma \setminus X)\big]\Big]^{-1}(\mathcal{T}) \\ \wedge \Pi_{\sigma\setminus X}^{\hbar(\sigma\setminus X)} = \varnothing\end{aligned}$$

Integrating our second example of optimisation – for $\Box X$ – into the translation rules seems more difficult, because this time there are two rules involved, $(X_\hbar)$ and $(\Box\,\hbar)$, neither of which has a full view of the rewrite proposition being written. This is a general remark: each part of a rewrite proposition gives information that may help to optimise another part of it. Indeed, if the input temporal property is a conjunction of many sub-properties, nothing prevents the derivation from ultimately yielding $[\mathcal{R} \setminus X](\Pi) = \varnothing \ \wedge\ \ldots\ \wedge\ \Pi \subseteq \mathcal{R}^{-1}(\mathcal{T})$, both statements stemming from completely independent parts of the input formula, instead of being the product of a single application of $(X_\hbar)$; the optimisation, although as valid as ever, will not be performed in that case. The same applies when length rejection statements are introduced by other rules, such as $(\bullet^m)$[p80]; there is an example of that in Sec. 4.5.1[p97]. Thus tinkering with rules for the purpose of optimisation seems unequal to the challenge of detecting all possible improvements. Optimisation is likely best kept to an intermediate, specialised processing phase, wedged between translation and procedure generation; only there can it have a full view of the rewrite proposition.

Note that there are doubtless many kinds of possible optimisations besides the two pointed out in this section, the exploration of which exceeds the scope of this discussion.

4.5 Examples & Discussion of Applicability

The informal question of whether the proposed verification chain is applicable in the real world rests on two separate issues. The first, and most important, is whether the fragment of LTL which can be handled is actually sufficiently large to describe relevant properties of systems. The second is whether the quality of the resulting positive approximated procedures is likely to be acceptable. To address the first issue, we take a look at the kinds of temporal patterns which can be translated, and attempt to quantify how useful they might be, based on the comprehensive study done on a large number of specifications in [Dwyer et al., 1999]. The second issue is dependant on both the depth of the temporal formula that needs to be checked, and the properties of the rewrite system under consideration. Thus we look at some existing TRS models in the literature, specifically a model for the Java Virtual Machine and Java bytecode in [Boichut et al., 2007], a model for the Needham–Schroeder protocol in [Genet & Klay, 2000], and a model for CCS specifications without renaming in [Courbis, 2011]. As a prelude to these discussions, we give three examples of derivations, using the temporal patterns of [Courbis et al., 2009].

4.5.1 Examples: Three Derivations

The three temporal properties below, while simple, are varied enough to test every main translation rule. The unused rules consist mostly of the tautology-based simplifications, such as $(\vee^{\Rightarrow}_{\wedge})_{[p75]}$, $(\vee^{\neg}_{\Rightarrow})_{[p76]}$, $(\wedge_X)_{[p74]}$, $(\vee_X)$, etcetera; all occasionally handy, especially on longer formulæ, but ultimately inessential. The first pattern, $\Box(X \Rightarrow \bullet^1 Y)$, has already been discussed and illustrated in the introduction; its translation into a rewrite proposition is obtained through a straight-forward, five-step derivation:

$$\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \Box(X \Rightarrow \bullet^1 Y)\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi \star\varepsilon \Vdash X \Rightarrow \bullet^1 Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi (\!| X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}_1} |\!) \Vdash \bullet^1 Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi (\!| X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}_1} |\!) \Vdash \circ^1 Y\rangle}{\updownarrow \frac{\langle X(\mathcal{R}^*(\Pi)) \fatsemi \star\varepsilon \Vdash Y\rangle}{[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing}(X^\star_\varepsilon)}(\circ^m)}(\bullet^m)}(\Rightarrow_\Sigma)}(\Box_*) \wedge X(\mathcal{R}^*(\Pi)) \subseteq \mathcal{R}^{-1}(\mathcal{T})\,.$$

The result can and should then be optimised into $[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \wedge X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T})$, which is the expected final translation. Recalling the discussion of optimisation in Sec. 4.4.2[p95], this is a very typical case where the optimised version of rule $(X_\hbar)$ is completely useless, as both parts of the formula are generated at different points. Hence the usefulness of a dedicated optimisation phase.

For the second phase, that is to say procedure generation, one cannot expect as

nice and readable a derivation as for the first phase. Indeed, besides branching on the structure of the rewrite proposition, one would require a new branch each time there is a choice to make – and even for this simple proposition, there are quite a lot of choices. The resulting tree structure is hard to represent on paper. Instead, we shall effect the generation "from the inside out", unwinding the recursivity and keeping track of the branching possibilities manually. Let us deal with $[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing$ first; we build the automaton accepting the language, step by step, and without making any a priori assumptions: thus we have $\Gamma = \varnothing$. Our first step is the initial language: $\varnothing \fatsemi \Pi \rightrightarrows \varnothing \fatsemi \Pi$; there is only this one possibility. The second step is $\mathcal{R}^*(\Pi)$, which is handled by the rule for reachability, applied to our particular case:

$$\varnothing \fatsemi [\Pi \rightarrowtail \varnothing, \Pi] \fatsemi \varnothing \vdash^* \Pi : \mathrm{TA} \fatsemi \mathcal{R}^*(\Pi) \rightrightarrows \langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle \fatsemi \mathcal{R}^{(*)}(\Pi) \ .$$

Here is our first branching: either X is regularity-preserving (for reachability) or it is not. Third step: $X(\mathcal{R}^*(\Pi))$, using the rule for forward-rewriting. Let us explore the first branch, where $\langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle = \varnothing$:

$$\begin{aligned} \varnothing \fatsemi [\mathcal{R}^{(*)}(\Pi) \rightarrowtail \varnothing, \mathcal{R}^{(*)}(\Pi)] \fatsemi \varnothing \vdash^* \mathcal{R}^{(*)}(\Pi) : \mathrm{TA} \fatsemi X(\mathcal{R}^{(*)}(\Pi)) \\ \rightrightarrows \langle X : \mathsf{reg\text{-}pres} \rangle \fatsemi X(\mathcal{R}^{(*)}(\Pi)) \ . \end{aligned}$$

And now, the second branch, where $\langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle = \{\mathcal{R} : \mathsf{reg\text{-}pres}^*\}$, which we write Δ in an effort to save a bit of space:

$$\begin{aligned} \varnothing \fatsemi [\mathcal{R}^{(*)}(\Pi) \rightarrowtail \Delta, \mathcal{R}^{(*)}(\Pi)] \fatsemi \Delta \vdash^* \mathcal{R}^{(*)}(\Pi) : \mathrm{TA} \fatsemi X(\mathcal{R}^{(*)}(\Pi)) \\ \rightrightarrows \Delta, \langle X : \mathsf{reg\text{-}pres} \rangle \fatsemi X(\mathcal{R}^{(*)}(\Pi)) \ . \end{aligned}$$

In both cases, this creates a new branch, depending on whether X is regularity-preserving for one step rewriting, for a total of four different possibilities, which one could summarise as

$$\langle \mathcal{R} : \mathsf{reg\text{-}pres}^*, X : \mathsf{reg\text{-}pres} \rangle, X(\mathcal{R}^*(\Pi)) \ .$$

On to the fourth step; this time, there are two possible rules that may apply. One could go on and compute the automaton for $[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi)))$, applying the forward-rewriting rule again; however, this is not possible in all branches. To simplify, let us state that the presence or absence of $\mathcal{R} : \mathsf{reg\text{-}pres}^*$ is moot at this point; it only influences whether $\mathcal{R}^{(*)}(\Pi)$ is an approximation. On the other hand, $X : \mathsf{reg\text{-}pres}$ influences the kind of automaton generated for $X(\mathcal{R}^*(\Pi))$, thus we really just need to discriminate between $\Delta \in \langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle$ and $\Delta \in \{X : \mathsf{reg\text{-}pres}\} \cup \langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle$ – writing Δ the assumptions accumulated in the current branch. In the first case(s), we have $\Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA}^=$, which requires the use of a constraint relaxation $\natural$:

$$\begin{aligned} \varnothing \fatsemi [X(\mathcal{R}^*(\Pi)) \rightarrowtail \Delta, X(\mathcal{R}^{(*)}(\Pi))] \fatsemi \Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA}^= \fatsemi \\ [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) \rightrightarrows \varnothing, \Delta, \langle \mathcal{R} \setminus Y : \mathsf{reg\text{-}pres} \rangle \fatsemi [\mathcal{R} \setminus Y](\natural X(\mathcal{R}^{(*)}(\Pi))) \ . \end{aligned}$$

In the second case(s), we have $\Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA}$, which obviates the need for such:

$$\begin{aligned} \varnothing \fatsemi [X(\mathcal{R}^*(\Pi)) \rightarrowtail \Delta, X(\mathcal{R}^{(*)}(\Pi))] \fatsemi \Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA} \fatsemi \\ [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) \rightrightarrows \varnothing, \Delta, \langle \mathcal{R} \setminus Y : \mathsf{reg\text{-}pres} \rangle \fatsemi [\mathcal{R} \setminus Y](X(\mathcal{R}^{(*)}(\Pi))) \ . \end{aligned}$$

Overall, this creates a third binary branching at this step; but there is more. As mentioned before, there is another possibility: one could use the special rule for testing emptiness without actually attempting to compute the automaton. While this could in principle be done in every case, the rule as written only applies if $\Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA}^=$ – which makes sense since it is not indispensable when the automaton can be built explicitly – and we have thus, for $\Delta \in \langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle$:

$$\varnothing \fatsemi [X(\mathcal{R}^*(\Pi)) \rightarrowtail \Delta, X(\mathcal{R}^{(*)}(\Pi))] \fatsemi \Delta \vdash^* X(\mathcal{R}^{(*)}(\Pi)) : \mathrm{TA}^= \fatsemi$$
$$[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \rightrightarrows \Delta \fatsemi [\mathcal{R} \setminus Y](X(\mathcal{R}^{(*)}(\Pi))) = \varnothing \,.$$

Let us count the cases: three binary branching account for $2^3 = 8$ cases. Furthermore, we have just seen that, in two cases of the second level, a different operation is permitted, which introduces no further branching, and concludes the generation – as far as $[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing$ is concerned. Therefore there are in total $8 + 2 = 10$ cases, of which 2 are concluded. Now, considering the first 8 cases, there remains to generate the emptiness test, which is a direct application of the relevant rule —with Δ being as usual the set of assumptions generated so far, and writing $\ell = [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi)))$ and $\alpha = [\mathcal{R} \setminus Y](X(\mathcal{R}^{(*)}(\Pi)))$, we have in all cases

$$\varnothing \fatsemi [\ell \rightarrowtail \Delta, \alpha] \fatsemi \Delta \vdash^* \alpha : \mathrm{TA} \vee \Delta \vdash^* \alpha : \mathrm{TA}^= \fatsemi \ell = \varnothing \rightrightarrows \Delta \fatsemi \alpha = \varnothing \,.$$

The rule applies whenever $\Delta \vdash^* \alpha : \mathrm{TA} \vee \Delta \vdash^* \alpha : \mathrm{TA}^=$, that is to say in all cases, and thus introduces no further branching nor any new assumptions, preserving the final count of 10 cases. Not all those cases are equally interesting; discarding the path that are strictly worse than others, there remain only two relevant possibilities: $\langle \mathcal{R} : \mathsf{reg\text{-}pres}^* \rangle$. Indeed, the following steps can be done without introducing any new approximations, for all X and Y. Thus what we have generated so far is a decision procedure if $\mathcal{R} : \mathsf{reg\text{-}pres}^*$, and merely a positive approximated procedure otherwise. Note that this analysis of the cases can easily be automated: for each generated set of assumptions Δ, complete Δ by kind inference into $\Delta' = \Delta \cup \{\, a \mid \Delta \vdash^* a \,\}$; only the cases where Δ' is minimal wrt. inclusion have to be considered. Let us move on to the second part of the rewrite proposition: $X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T})$. The automaton accepting $X(\mathcal{R}^*(\Pi))$ is built exactly as before – again, there are four cases. $Y^{-1}(\mathcal{T})$ is built in one step, and introduces a binary branching:

$$\varnothing \fatsemi Y^{-1}(\mathcal{T}) \rightrightarrows \varnothing, \langle Y : \mathsf{left\text{-}lin} \rangle \fatsemi Y^{-1}(\mathcal{T}) \,.$$

Let us recall the rule for inclusion $(4.12)_{[p94]}$:

$$\Gamma \fatsemi [\ell \rightarrowtail \Delta, \alpha; \ell' \rightarrowtail \Delta', \alpha'] \fatsemi \Delta' \not\vdash^* \alpha' : + \fatsemi \ell \subseteq \ell' \rightrightarrows \Gamma, \Delta, \Delta' \fatsemi \alpha \subseteq \alpha' \,.$$

Fortunately, $\langle Y : \mathsf{left\text{-}lin} \rangle \not\vdash^* Y^{-1}(\mathcal{T}) : +$, so the rule applies, preserving the two cases. If $\Delta' = \{Y : \mathsf{left\text{-}lin}\}$, then $\Delta' \vdash^* Y^{-1}(\mathcal{T}) : \mathrm{TA}$, and the inclusion test is approximated, as shown by kind inference; if $\Delta' = \varnothing$, then $\Delta' \vdash^* Y^{-1}(\mathcal{T}) : \mathrm{TA}^=$, and it is a decision. Overall, we have a decision procedure if $Y : \mathsf{left\text{-}lin}$ and $\mathcal{R} : \mathsf{reg\text{-}pres}^*$, and a positive approximated procedure otherwise. Formally, a total of 12 cases (theorems) have been generated, though most are redundant. It should be apparent by now that procedure generation is easy to understand and implement, although cumbersome to write – even in the somewhat bowdlerised version above, where all the kind-inference derivations were kept implicit. Thus we shall focus on the first phase (translation into rewrite propositions) for the remaining examples.

Our second example pattern, $\neg Y \ \wedge \ \Box(\bullet^1 Y \Rightarrow X)$, states that rules of Y may only appear after rules of X – it features a typical case of an implication whose antecedent is in the future wrt. its consequent, and showcases the generality of rule $(X_\hbar)_{[p86]}$. A four-steps derivation suffices:

$$\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \neg Y \ \wedge \ \Box(\bullet^1 Y \Rightarrow X)\rangle}{\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \neg Y\rangle}{\updownarrow \frac{\langle \Pi \fatsemi \star\varepsilon \Vdash \mathcal{R} \setminus Y\rangle(X^\star_\varepsilon)}{Y(\Pi) = \varnothing}}(\neg X)} \wedge \updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \Box(\bullet^1 Y \Rightarrow X)\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi \star\varepsilon \Vdash \bullet^1 Y \Rightarrow X\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi \wr \mathcal{R}, Y \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_2 \int \Vdash X\rangle(X_\hbar)}{Y([\mathcal{R} \setminus X](\mathcal{R}^*(\Pi))) = \varnothing}}(\Rightarrow_\Sigma)}(\Box_*)}(\wedge)$$

Procedure generation for the resulting rewrite proposition,

$$Y(\Pi) = \varnothing \ \wedge \ Y([\mathcal{R} \setminus X](\mathcal{R}^*(\Pi))) = \varnothing \,,$$

is strictly simpler than for the previous example. In this case, the property is decided if $\mathcal{R} : \mathsf{reg\text{-}pres}^*$, and positively approximated otherwise.

The third example property, $\Box(X \Rightarrow \circ^1 \Box \neg Y)$, states that any use of a rule in X precludes the subsequent use of any rule in Y, for the remaining execution. Applying the translation rules as usual, we obtain the following derivation – writing $\ell = X(\mathcal{R}^*(\Pi))$ in order to save some space:

$$\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \Box(X \Rightarrow \circ^1 \Box \neg Y)\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi \star\varepsilon \Vdash X \Rightarrow \circ^1 \Box \neg Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\Pi) \fatsemi \wr X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 \int \Vdash \circ^1 \Box \neg Y\rangle}{\updownarrow \frac{\langle \ell \fatsemi \star\varepsilon \Vdash \Box \neg Y\rangle}{\updownarrow \frac{\langle \ell \fatsemi \star\varepsilon \Vdash \circ^0 \neg Y\rangle}{\updownarrow \frac{\langle \ell \fatsemi \star\varepsilon \Vdash \neg Y\rangle}{\updownarrow \frac{\langle \ell \fatsemi \star\varepsilon \Vdash \mathcal{R} \setminus Y\rangle(X^\star_\varepsilon)}{Y(X(\mathcal{R}^*(\Pi))) = \varnothing}}(\neg X)}(\circ^m)} \wedge \updownarrow \frac{\langle \mathcal{R}(\ell) \fatsemi \varepsilon \Vdash \Box \neg Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\mathcal{R}(\ell)) \fatsemi \star\varepsilon \Vdash \neg Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(\mathcal{R}(\ell)) \fatsemi \star\varepsilon \Vdash \mathcal{R} \setminus Y\rangle(X^\star_\varepsilon)}{Y(\mathcal{R}^*(\mathcal{R}(X(\mathcal{R}^*(\Pi))))) = \varnothing}}(\neg X)}(\Box_*)}(\Box_\hbar)}(\circ^m)}(\Rightarrow_\Sigma)}(\Box_*)$$

This, while correct, is not the best possible translation. It is indeed noticeable that both statements overlap: overall they have the form $Y(\ell) = \varnothing$ and $Y(\mathcal{R}^+(\ell)) = \varnothing$, which would gain to be combined into $Y(\mathcal{R}^*(\ell)) = \varnothing$. This inefficiency can be corrected by noticing that, since $\lambda \models \Box\varphi$, for all formulæ φ, it holds that

$$\langle \Pi \fatsemi \sigma \Vdash \Box\varphi\rangle \iff \langle \Pi \fatsemi \star\sigma \Vdash \Box\varphi\rangle \,,$$

and thus, when translating a $\Box$, the star can simply be ignored in practice. In this particular case, doing so proves useful, since ε is stable, while $\star\varepsilon$ is not. Using this remark, the nicer translation is obtained immediately; starting again at the fourth step, we apply the "un-starred" version of $(\Box_*)$ instead of the overly general $(\Box_\hbar)$:

$$\updownarrow \frac{\langle X(\mathcal{R}^*(\Pi)) \fatsemi \star\varepsilon \Vdash \Box \neg Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(X(\mathcal{R}^*(\Pi))) \fatsemi \star\varepsilon \Vdash \neg Y\rangle}{\updownarrow \frac{\langle \mathcal{R}^*(X(\mathcal{R}^*(\Pi))) \fatsemi \star\varepsilon \Vdash \mathcal{R} \setminus Y\rangle(X^\star_\varepsilon)}{Y(\mathcal{R}^*(X(\mathcal{R}^*(\Pi)))) = \varnothing}}(\neg X)}(\Box_*)$$

The resulting translation, $Y(\mathcal{R}^*(X(\mathcal{R}^*(\Pi)))) = \varnothing$, can be optimised into

$$Y([\mathcal{R} \setminus Y]^*(X([\mathcal{R} \setminus X]^*(\Pi)))) = \varnothing \,,$$

following a slight generalisation of the second optimisation idea in Sec. 4.4.2$_{[p95]}$ – a generalisation which is easy to automate and justify. Ultimately, this property can be decided under the assumptions $\mathcal{R} : \mathsf{reg\text{-}pres}^*$ (or, more precisely, $\mathcal{R} \setminus X : \mathsf{reg\text{-}pres}^*$ and $\mathcal{R} \setminus Y : \mathsf{reg\text{-}pres}^*$), and $X : \mathsf{reg\text{-}pres}$; it is positively approximated otherwise.

	Scope					
Pattern	Global	Before	After	Between	Until	*Support* ⩽
Absence	**41**	5	12	18	9	48%
Universality	**110**	1	**5**	2	1	96%
Existence	12	1	4	8	1	0%
Bound Existence	0	0	0	1	0	0%
Response	**241**	1	**3**	0	0	99%
Precedence	**25**	0	1	0	0	96%
Resp. Chain	8	0	0	0	0	0%
Prec. Chain	1	0	0	0	0	0%
Support ⩽	95%	0%	32%	0%	0%	83%

Figure 4.3: Partially supported patterns from [Dwyer et al., 1999].

4.5.2 Coverage of Temporal Specification Patterns

A survey was conducted in [Dwyer et al., 1999] on a large number (555) of specifications, from many different sources and application domains. They were classified according to which pattern and scope – as in [Dwyer, Avrunin & Corbett, 1998] – each specification was an occurrence of. In this section we examine briefly which of those pattern/scope combinations are amenable to checking using our method – often assuming that the pattern atoms P, Q etcetera correspond to simple formulæ – and tally the percentages of real-world cases (per the survey) each such combination accounts for. Note that this has to be a somewhat optimistic estimation because of the assumption on the simplicity of atoms, without which one could simply choose an atom complicated enough to make *any* pattern untranslatable. In this section, we only consider the first phase (translation into rewrite proposition), and do not accept approximated translations.

Absence patterns are supported for *global* scopes ($\Box\,\neg P$) and *after* scopes ($\Box(Q \Rightarrow \Box\,\neg P)$) —the second derivation of the previous section was an example of that. In both cases, P should follow the grammar $P := X \mid \neg X \mid P \wedge P \mid P \vee P$, so that it always boils down to an atom $P \subseteq \mathcal{R}$ after liberal application of rules $(\wedge_X)_{[p74]}$, $(\vee_X)$ and $(\neg X)_{[p77]}$. Note that the latter rule can be applied without caution because, in both cases, P is under the scope of a $\Box$ operator, which, as we have seen in the previous section, renders moot the introduction of a star performed by rule $(\neg X)$. As for Q, it should be an $\mathcal{A}$-LTL formula. The other scopes (*Before*, *Between*, and *Until*) involve heavy uses of $\Diamond$ and $\mathbf{U}$, and thus cannot be translated exactly —in fact we cannot handle those scopes for any pattern.

Universality is very similar to *Absence*, and is handled for *global* scopes ($\Box\,P$) and *after* scopes ($\Box(Q \Rightarrow \Box\,P)$). Again, Q should be $\mathcal{A}$-LTL, but this time – thanks to the absence of negations – there is no particular restriction on P.

Response is partially supported; *global* scopes are of the form $\Box(P \Rightarrow \Diamond S)$ and *after* scopes of the form $\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond S))$. Although both use a $\Diamond$ operator, this is not strictly needed in some practical cases. For instance, the first formula of the previous section was a response pattern, but instead of stating that there would eventually be a response, it was more specific and used a next operator to

assert that the response would take place on the next step. Thus the following response patterns are supported: $\Box(P \Rightarrow \circ^k S)$, $\Box(P \Rightarrow \bullet^k S)$, $\Box(Q \Rightarrow \Box(P \Rightarrow \circ^k S))$ and $\Box(Q \Rightarrow \Box(P \Rightarrow \bullet^k S))$, for $P, Q \in \mathcal{A}$-LTL, and without any specific restriction on S.

Similarly, *Precedence* is partially supported for *global* scopes: $(\Box \neg P) \vee (\neg P \mathbf{U} S)$ is not directly translatable, but whenever the exact number of steps is known, a pattern of the form $\Box(\bullet^k P \Rightarrow S)$ suffices. Again, $P \in \mathcal{A}$-LTL, and S has no special restriction.

The remaining patterns (*Existence*, *Bounded Existence*, *Response Chain* and *Precedence Chain*) are not translatable, and using explicit numbers of steps in those cases would likely be less useful in those cases than it can be for *Response* and *Precedence*

Table 4.3 is a summary of the results of the survey in [Dwyer et al., 1999], that shows to which pattern/scope combination each of the 555 specifications belongs. The combinations which are handled to some extent by an exact translation into rewrite propositions have their numbers shown in bold face. For each pattern, the last cell of the line gives the proportion of cases belonging to a supported (or partially supported) scope. For each scope, the last cell of the column gives the proportion of cases belonging to a supported (or partially supported) pattern. The cell at the bottom-right gives the proportion of cases which belong to a supported (or partially supported) pattern/scope combination, out of all cases which corresponded to a recognisable pattern/scope (511).

What should be taken from this table is *certainly not* the final 83%, which is grossly optimistic. It should be kept in mind that the numbers in boldface constitute upper bounds. For instance, it is unknown how many of the global response patterns were formulated – or could have been reformulated – in terms of exact steps, and that is the single largest category. Nevertheless, the table shows that the patterns and scopes that the method targets are actually the ones which are most likely to matter in practice.

4.5.3 Encodings: Java Byte-Code, Needham–Schroeder & CCS

Even if the temporal property under consideration admits of an exact translation into rewrite proposition, there is no guarantee that the corresponding positive approximated procedure will be fine-grained enough to be of any use. How fine or coarse it is depends upon two factors: the depth of the temporal formula, and the properties of the TRS. There are three main constructions which introduce approximations: $\mathcal{R}^*(\ell)$, Π^n_σ, and $\ell \subseteq \ell'$, where ℓ' is not regular. The first is the best-controlled source of approximations, which draws upon considerable resources in the existing literature. The second can be very crude: let us write $\Pi^n_\sigma = X_n(\cdots X_2(X_1(\Pi))\cdots)$. If it cannot be assumed that each X_i is regularity-preserving for one-step rewriting, then the generated automaton will have to be the $\mathrm{TA}^=$ $X_n(\natural X_{n-1}(\cdots \natural X_2(\natural X_1(\Pi))\cdots))$. That amounts to a total of $n-1$ constraint relaxations in a row, each of which is a very crude approximation. The third only occurs in length rejection statements, of the form $\ell \subseteq \mathcal{R}^{-1}(\mathcal{T})$, and can be a problem as soon as $\mathcal{R}$ is not left-linear. It is thus clear that, in order for the method to perform well, either the temporal formula must be kept quite simple, or the TRS must have some nice properties. We look at some existing non-trivial TRS models

in the literature in order to see what properties they satisfy, and thus get an idea of how applicable the method might be in their application domains.

Let us look at a TRS modelling Java Virtual Machine and Java bytecode semantics, given in [Boichut et al., 2007] and implemented for automatic generation from Java bytecode in [Barré et al., 2009], of which an example rule is given below:

$$\begin{aligned} &\mathit{xframe}(\mathit{add}, \mathtt{m}, \mathtt{pc}, \mathit{stack}(\mathtt{b}, \mathit{stack}(\mathtt{a}, \mathtt{s})), \mathtt{l}) \to \\ &\qquad\qquad \mathit{frame}(\mathtt{m}, \mathit{next}(\mathtt{pc}), \mathit{stack}(\mathit{xadd}(\mathtt{a}, \mathtt{b}), \mathtt{s}), \mathtt{l}) \ . \end{aligned}$$

By construction, all the rules of the system are left-linear: this is actually a constraint under which the authors were working, in order to accommodate the completion algorithm they were using. Left-linearity is not a crippling constraint in many cases, as basic functional definitions and pattern-matchings in programming languages are naturally left linear. This is good, since it means that length-rejection statements and their inclusions need not be approximated, and removes one source of coarseness. Looking further at the rules encoding Java bytecode semantics, it turns out that they are actually also right-linear. The TRS is linear overall, and that means that any atom $X \subseteq \mathcal{R}$ is regularity-preserving for one-step rewriting: languages of the form Π^n_σ are regular and can be computed exactly. This leaves only instances of $\mathcal{R}^*(\ell)$ as a potential source of approximation. That one is unavoidable, but fortunately it is also the "best" kind of approximation, as mentioned earlier. All in all, the method is likely to apply quite well to TRS from this problem domain.

Contrariwise, the TRS encoding the Needham–Schroeder protocol proposed in [Genet & Klay, 2000] is neither left- nor right-linear. Here is an example rule – slightly modified to remove notations irrelevant to this discussion:

$$\begin{aligned} &\mathit{goal}(x, y) \to \mathit{join}(\mathit{goal}(x, y), \\ &\qquad\qquad \mathit{mesg}(x, y, \mathit{encr}(\mathit{pubkey}(y), x, \mathit{cons}(N(x, y), \mathit{cons}(x, \mathit{null}))))) \ . \end{aligned}$$

It is likely that the encodings of most cryptographic protocols should present similar challenges, given how often they rely on tests of equalities – of keys, agents etcetera, which precludes left-linearity, – and repetition of information – for instance the agent may appear both in clear and encrypted, which precludes right-linearity. It is of course possible to alleviate this problem by abstracting the TRS – as done in [Genet & Klay, 2000] over agents and nonces, – but this introduces another layer of approximation. Choosing whether to abstract at the level of the TRS, or to incur coarser approximations later on, is therefore a compromise that is best made on a case-by-case basis.

A TRS encoding of CCS specifications without renaming is given in [Courbis, 2011]; two of the rewrite rules involved are reproduced below:

$$\begin{aligned} \mathit{Com}(r, \mathit{Sys}(x, p)) &\to \mathit{Sys}(x, \mathit{Com}(r, p)) \\ \mathit{Com}(\mathit{Sys}(x, p), \mathit{Sys}(\mathit{bar}(x), r)) &\to \mathit{Sys}(\tau, \mathit{Com}(p, r)) \ . \end{aligned}$$

As in [Boichut et al., 2007], although right-linearity was not a design constraint, it turns out in practice that all the rules are right-linear. Conversely, while left-linearity was a requirement, for the same reasons as before, two of the rules – such as the second rule reproduced above – are not left-linear. The solution used in [Courbis, 2011] consists in replacing each of those non-left-linear rules ρ by the set

of rules obtained by substituting an action for x in ρ. Since any given CCS program makes use of only a finite number of actions, the resulting TRS is therefore finite, and linear.

Another way to deal with that would have been to notice that both non–left-linear rules are of the same form as $(3.3)_{[p50]}$, and can therefore be handled through the use of language intersections. Either way, as for Java bytecode, this leaves $\mathcal{R}^*(\ell)$ as the sole potential source of approximation, and the method seems therefore well-suited to this domain as well.

4.6 Conclusions & Perspectives

The systematic generation of (possibly) approximated procedures to determine whether a given term-rewriting system satisfies a specific linear temporal logic property is addressed in two steps: first, an equivalent rewrite proposition is generated; second, a (possibly) approximated procedure is derived from the rewrite proposition. The first step is achieved through a set of translation rules relying on the notion of signatures, or models of the sub-formulæ appearing as antecedents of the general LTL formula. The second step is served by procedure generation rules, whose main aim is to juggle with the expressive power required to encode the tree languages involved, and to manage approximations. A survey of the existing literature indicates that the method targets widely used patterns of (safety) properties, and that linearity properties which alleviate coarseness in the positive approximated procedure are naturally met by existing TRS models in some problem domains (e.g. bytecode, CCS semantics).

Regarding approximations, the focus of the present work is on exact translations for the first step, relegating all approximations to the second, where the body of work existing on approximating regular tree languages comes into play. Future works should focus on extending the first step with fine under-approximated translations wherever exactness is not achievable, in particular with operators of the “until” family —though the method we are currently considering requires a slightly more expressive variety of rewrite propositions. Integrating equational theories in the same framework, if feasible, would also increase the method’s applicability. The end goal is of course the integration of our proposals into the verification chain dedicated to the automatic analysis of security-/safety-critical applications.

— Part III —

Efficiently Solving Decision Problems for Tree Automata with Global Constraints

Chapter 5
A Brief History of Constraints

Contents

—Where we meet a lot of automata with constraints, old and new.

Tree automata with global constraints, which are an significant component of our model-checking method, are a rather recent development of automata theory. The present chapter completes the curt presentation that was afforded them in Sec. 2.5[p35] by putting them into a larger context. It provides a few motivating examples for the study of constraints and fleshes out the history, state of the art, and taxonomy of extended models of tree automata. The reader uninterested in such things may want to skip directly to the very last section, which introduces a few notational conventions that will prove convenient in this part of the thesis – and particularly indispensable in the next chapter, where our own contributions to the study of the algorithmic complexity of decision problems for global constraints are presented.

More in the Appendix!

The reader interested in a more exhaustive survey will find that two other classes – DAG and memory automata – are presented in the Appendix, Section 11.1[p191].

The primary reference for this chapter is of course [Comon et al., 2008], which provides an extensive survey of automata with positional constraints, though it does not touch on global constraints. On this latter subject, we would point the reader towards [Filiot et al., 2010; Vacher, 2010] both as primary literature and surveys.

5.1 Tree Automata With Positional Constraints

A powerful motivation for the study of constraints was encountered in Part II: as we have mentioned, regularity is not preserved through rewriting, even when a

single step is involved. That is to say, $\mathcal{R}$ being some rewrite system, $\mathcal{R}(\ell)$ is not – in general – regular, even if ℓ itself is a regular tree language. That this is the case is quite clear when considering non-linear rewrite rules, such as $g(x) \to f(x,x)$. With the reminder that the language of ground terms of $f(x,x)$ is denoted by $L_=$ (2.4)[p35], it is immediate that

$$\{\, g(x) \to f(x,x) \,\}(\mathcal{T}(\mathbb{A})) = L_= \,,$$

and while $\mathcal{T}(\mathbb{A})$ is trivially regular, $L_=$ is, as we have already seen, known to be non-regular [Comon et al., 2008], although it is easily recognisable by a TAGE. This example makes two important general points: first, non-linearity is problematic, and second, crafting extended models of automata capable of testing equality of subterms presents itself as a natural solution to that problem. This observation is not novel by any means, nor is the interest in tackling non-linearity of terms and rules, which pervades logic, equational programming, etcetera. Under that impetus, the first class of tree automata extended with such capabilities, *tree automata with local equality and disequality constraints* (*TALEDC*, and originally *RATEG*) was proposed more than three decades ago in the thesis [Mongy, 1981], supervised by Max Dauchet.

tree automata with local equality and disequality constraints
TALEDC

5.1.1 The Original Proposal

A TALEDC is a BUTA whose transition rules carry additional information: indeed, each rule is coupled with a set C of constraints of the form $\alpha \cong \beta$ or $\alpha \ncong \beta$, where α and β are some positions. No provision is made at this stage as to whether those positions "exist" in some sense; they are simply strings of integers. Encoding positions starting with 1, as is the convention throughout most of this document, they need simply satisfy $\alpha, \beta \in \mathbb{N}_1^*$. We call these *positional constraints* so as to distinguish them from the global constraints on states seen in sections 2.5[p35] and 5.2[p111]. Notation-wise, positional constraints are generally affixed to the left-hand side of the rules or to the arrow, as follows:

positional constraints

$$\sigma(p_1, \ldots, p_n)[C] \to q \quad \text{or} \quad \sigma(p_1, \ldots, p_n) \to_C q \,.$$

For the sake of brevity, an empty set of constraints is not represented:

$$\sigma(p_1, \ldots, p_n) \to_\varnothing q \quad \text{is written} \quad \sigma(p_1, \ldots, p_n) \to q \,.$$

The execution – or run – of a TALEDC is defined as a direct extension to that of a BUTA. The mapping $\rho : \mathcal{P}(t) \to Q$ is a run on t if, for all $\alpha \in \mathcal{P}(t)$, there is a transition $t(\alpha)(\rho(\alpha.1), \ldots, \rho(\alpha.n))[C] \to \rho(\alpha) \in \Delta$ such that, for all constraints $\beta \cong \gamma \in C$ (resp. $\beta \ncong \gamma \in C$), $\alpha.\beta, \alpha.\gamma \in \mathcal{P}(t)$ and $t|_{\alpha.\beta} = t|_{\alpha.\gamma}$ (resp. $t|_{\alpha.\beta} \neq t|_{\alpha.\gamma}$). For instance, the transition rule

$$f(q_1, q_2, q_3)[13 \cong 2, 12 \ncong 13] \to q$$

may only apply at position α when $t|_{\alpha.13} = t|_{\alpha.2}$ and $t|_{\alpha.12} \neq t|_{\alpha.13}$. This class is quite expressive; it can easily accommodate $L_=$, for instance, with $Q = \{q, q_f\}$, $F = \{q_f\}$ and the following transitions:

$$\Delta = \{\, a \to q, b \to q, f(q,q) \to q, f(q,q)[1 \cong 2] \to q_f \,\} \,. \tag{5.1}$$

TALEDC are closed by union and intersection, but unfortunately not by complementation; moreover, the emptiness problem is undecidable, which is a crippling shortcoming for some applications. This class is still actively studied, however. For instance a recent result [Godoy, Giménez, Ramos & Àlvarez, 2010] showed that any member of the positive subclass can be complemented into a member of the negative subclass, and conversely.

The negative/positive terminology introduced in section 2.5 generalises to all classes of automata with constraints. Positive = only equalities; negative = only disequalities.

5.1.2 A Stable Superclass With Propositional Constraints

In using the "superclass" angle of attack, we follow the unifying approach of [Comon et al., 2008]. This slightly deviates from the historical definitions, but provides a much smoother overview wrt. closure properties of the other classes below.

As could be expected, the limitations of TALEDC elicited interest in more tractable classes. One way of approaching that is to trade some degree of expressive power for decidability of emptiness. The other angle of attack, if one is interested in closure properties rather than in decidability, is to go the opposite way and look for a stable superclass. It so happens that there is a painless way in which to achieve stability: it suffices to generalise the sets of constraints C, where all atomic constraints $c_1, \ldots, c_n \in C$ can be seen as being taken conjunctively as $c_1 \wedge c_2 \wedge \cdots \wedge c_n$, into more general propositional formulæ:

$$C := \beta \cong \gamma \mid \beta \ncong \gamma \mid C \wedge C \mid C \vee C \mid \neg C \mid \ldots . \tag{5.2}$$

The satisfaction of such constraints is defined in the obvious way. This class, which we shall call *tree automata with propositional local equality and disequality constraints* (*TAPLEDC*), is studied in [Comon et al., 2008, Chap. 4, as AWEDC], and is shown to be determinisable in exponential time – with up to exponential size increase – and closed by all boolean operations in the same way as run-of-the-mill BUTA – linearly for union, quadratically for intersection, and exponentially for complementation. Indeed, TAPLEDC is simply the closure of TALEDC by complementation; this is vindicated by the intuition that one needs to negate constraints in order to effect determinisation. Membership is decidable in polynomial time – or even in linear time in the deterministic case – but emptiness is, of course, undecidable, as befits a superclass of TALEDC.

tree automata with propositional local equality and disequality constraints
TAPLEDC

A TAPLEDC is still deterministic if two transitions have the same left-hand side, provided that the conjunction of their constraints is not satisfiable.

5.1.3 Constraints Between Brothers

Let us now examine classes for which emptiness is decidable. Instead of directly looking for subclasses of TALEDC, we shall use TAPLEDC as a starting point, and in so doing obtain decidability without necessarily losing the aforementioned cosy closure properties. The restrictions that we are about to invoke are indeed orthogonal to whether the constraints are taken conjunctively or not.

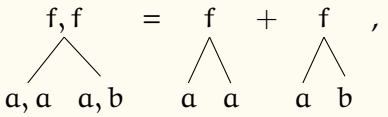
The first decidable subclass to be discovered is founded on the observation that one does not always need constraints over the full range of possible positions. The automaton (5.1) illustrates this by showing that the ur-example $L_=$ is expressible using a constraint over direct children. The class obtained by restricting TALEDC to constraints $\beta \cong \gamma \in C$ (resp. $\beta \ncong \gamma \in C$) such that $|\beta| = |\gamma| = 1$ is called *tree automata with equality and disequality constraints between brothers* (*TABB*), and was first studied in [Bogaert & Tison, 1992].

tree automata with equality and disequality constraints between brothers
TABB

As desired, emptiness is decidable, though not cheaply so in the non-deterministic case: the problem is ExpTime-hard. The proof is similar to that for TAGE [Filiot et al.,

2010, 2008], and we shall see another version of this argument – albeit in a more restrictive context – in the next chapter. TABB inherit from TAPLEDC the closure properties by all standard boolean operations, without any modification of the algorithms – determinisation of TAPLEDC rearranges the propositional constraint formula but does not alter the nature of its atomic constraints. Additionally, in the deterministic case, emptiness is actually testable in polynomial time.

Generally speaking, almost all standard decidability results about BUTA carry over to TABB – albeit with increased complexities – and the linearity conditions wrt. rewriting etcetera are relaxed into conditions of "shallow" non-linearity. Exceptions to that arise chiefly in the domain of automata on tuples of finite trees [Comon et al., 2008, Sec. 3.2]. For instance, TABB are not closed under projection and cylindrification. This mainly reflects the fact that positional constraints do not mix well with overlapping trees; for instance it holds that $1 \not\approx 2$ for the tree



where $+$ symbolises tree overlapping, but this is no longer the case for the projection on the first component. More recently, TABB have also been extended to unranked trees [Wong & Löding, 2007].

5.1.4 Reduction Automata

reduction automata
RA

The second subclass, *reduction automata* (*RA*), was introduced in [Dauchet, Caron & Coquidé, 1995], and is a bit more delicate to define. The intuition is that there is a bound on the number of equality constraints – but not of disequality constraints – that may be invoked in any individual run of the automaton. This is accomplished by means of an ordering of the states, and an additional condition on the transitions. Indeed, an RA is a TAPLEDC equipped with a partial ordering on states $(\preccurlyeq) \subseteq Q^2$ such that, for all transitions $\sigma(p_1, \ldots, p_n)[C] \to q \in \Delta$, it must hold that $p_i \preccurlyeq q$, for all $i \in [\![1, n]\!]$. That is to say, q is an upper bound of $\{p_1, \ldots, p_n\}$ wrt. $\preccurlyeq$. Furthermore, if any of the atoms of C is an equality constraint, then the upper bound must be strict: $p_i \prec q$, for all i. To show that a particular TAPLEDC is an RA, it suffices to exhibit a suitable ordering on the states. By way of example, the automaton (5.1) accepting $L_=$ can be shown to be an RA simply by taking $q \prec q_f$.

RA are closed under union and intersection, but it has yet to be determined whether they are closed under complementation. It is, however, known that complete, deterministic RA are closed under complementation. The history of emptiness decision for RA is slightly convoluted. It was claimed in [Dauchet et al., 1995] that emptiness was decidable, a claim that stood for more than a decade, until [Jacquemard, Rusinowitch & Vigneron, 2008] contradicted it in the non-deterministic case by reducing the halting problem for 2-counter machines. Nevertheless, decidability does hold in the complete and deterministic case; note that the proof of [Dauchet et al., 1995] pertained to that case, and that the claim that the argument could be generalised to the non-deterministic case was only made en passant. The proof only provides an impracticably high upper bound on the complexity, though; the lower bound is unknown. Finiteness is decidable as well, under the same conditions of completeness and determinism.

[Comon et al., 2008, Thm. 4.4.9] claims finiteness to be decidable, without restriction, but that is probably a remnant from the 2007, pre-[Jacquemard et al., 2008] version. See the next chapter for a reduction of emptiness to finiteness.

RA have deep applications in the domain of rewriting; let us just mention two important results which illustrate why they are called "reduction" automata. Given any term rewriting system $\mathcal{R}$, the set of ground $\mathcal{R}$–normal-forms can be represented by an RA – the construction is exponential in time and size. Therefore, one can deduce from the results on RA that, for any TRS $\mathcal{R}$, it is decidable whether the language of ground $\mathcal{R}$–normal-forms is empty, and it is also decidable whether it is finite. Another important result in that field is that it is even decidable whether it is regular.

Further known restrictions of RA sport better decidability results whilst preserving very useful capabilities. The negative subclass of RA (*-RA*) – or equivalently of TAPLEDC, since the ordering on states is moot in the negative case – is expressive enough to accept ground $\mathcal{R}$–normal-forms, but emptiness can be decided in exponential time. Moreover, deterministic RA whose constraints do not overlap (*noRA*) admit of a polynomial time emptiness test.

-RA

noRA

It is worth noting that RA, along with BUTA, have recently been extended to equality modulo theories in [Jacquemard et al., 2008].

5.1.5 Reduction Automata Between Brothers

The two subclasses RA and TABB can be merged into the more general class known as *generalised reduction automata* (*GRA*), introduced in [Caron, Comon, Coquidé, Dauchet & Jacquemard, 1994], which is defined as a TAPLEDC such that, for all transitions $\sigma(p_1, \ldots, p_n)[C] \to q \in \Delta$, q is an upper bound of $\{p_1, \ldots, p_n\}$, and, if any of the atoms of C is an equality constraint $\beta \cong \gamma$ such that $|\beta| = |\gamma| > 1$, then the upper bound must be strict. In other word, this relaxes the bound of RA on the number of equality tests, but only where tests between brothers are concerned. This class inherits the closure and decidability results of RA. This is the most general known class of automata with positional constraints for which emptiness is decidable – again, under conditions of completeness and determinism.

generalised reduction automata

GRA

GRA actually predate RA by a few months; GRA generalised TABB and another class called encompassment automata. The relationships are given here in hindsight.

Same remark as above concerning the unspecified claim of decidability of [Comon et al., 2008, p133]

5.2 Tree Automata With Global Constraints

In comparison to positional constraints, the use of global constraints is a very recent development. While positional constraints have many applications, as the previous section attests, they are intrinsically limited in some ways that hamper their applicability in certain contexts, in particular regarding tree patterns for XML. The nub of their shortcomings in that field stems from their inability to test equality of subterms that may be arbitrarily far from one another. A typical example is the case of a bibliography and a pattern $X(\mathsf{author}(x), \mathsf{author}(x))$, where X is a binary context. The set of ground terms of such a pattern cannot be represented with positional constraints, as there is no telling how far the first instance of author may be from the other one. The emergence of XML as a lingua franca of Internet services, file formats, databases and more has renewed interest in formalisms capable of handling such cases. Driven by those motivations, the first variety

of automata with global constraints was initially introduced in [Filiot, Talbot & Tison, 2007] as a means to prove decidability of a sizeable fragment of TQL, a query logic for semi-structured data. The authors of that paper have continued their study of TAGED in [Filiot et al., 2008, 2010; Filiot, 2008]. We have already given the definition for this class in section 2.5[p35]. They are closed by union and intersection, but not by complementation; they are not determinisable. Membership testing is NP-complete, and emptiness as well as finiteness are ExpTime-complete for the positive subclass, while emptiness for the negative subclass is in NExpTime – a result that can be refined into NP-hardness by noting the connexion to DAG automata, which is presented in section 11.1.1. Whether emptiness is decidable for the general class remained an open question for a few years – until the works cited in the next section – but it was shown decidable in 2NExpTime for *vertically bounded TAGED* (*vbTAGED*), i.e. TAGED for which the number of disequality constraints along any root-to-leaf path is bounded. Universality and inclusion are undecidable.

vertically bounded TAGED

vbTAGED

5.2.1 Generalisation to Propositional Constraints and More

TAGED were extended in [Barguñó, Creus, Godoy, Jacquemard & Vacher, 2010] following the same modus operandi as the generalisation of TALEDC in TAPLEDC. Instead of having two sets of constraints (equalities and disequalities), the automaton is equipped with a boolean formula of constraints, as in (5.2). Since this is presently the most general class of automata with global – equality and disequality – constraints, it is simply called *tree automata with global constraints* (*TAGC*). It is proven in [Barguñó et al., 2010] that emptiness is decidable for TAGC, and therefore for TAGED. The authors pushed the inquiry even further, by considering global arithmetic constraints over the number $|q|$ of occurrences of a given state q during a run, or even over the number $\|q\|$ of distinct subterms evaluated in q during a run. The arithmetic constraints are expressed as linear inequalities of the form

tree automata with global constraints

TAGC

$$\sum_{q\in Q} a_q|q| \geqslant b \quad \text{or} \quad \sum_{q\in Q} a_q\|q\| \geqslant b \quad \text{with } a_q, b \in \mathbb{Z}, \forall q \in Q\,.$$

Tree automata with such constraints – and only such constraints – are known as *Parikh tree automata* [Klaedtke & Rueß, 2002], or *linear constraint tree automata* [Bojanczyk, Muscholl, Schwentick & Segoufin, 2009], and emptiness is known to be decidable for this class. However, adding global equality constraints to Parikh automata (*Parikh+E*) immediately breaks decidability. It is possible to remedy that by restricting arithmetic constraints to *natural* linear inequalities, by adding the proviso that the coefficients a_q and a all have the same sign. Equivalently, the coefficients are restricted to natural numbers, and the constraints are of the form

Parikh tree automata

Parikh+E

$$\sum_{q\in Q} a_q f(q) \gtrless b \quad \text{with } a_q, b \in \mathbb{N}, f \in \{|\cdot|, \|\cdot\|\}, (\gtrless) \in \{\geqslant, \leqslant\}\,.$$

The class of Parikh automata with natural linear constraints as well as global equality and disequality constraints (*NParikh+ED*) has decidable emptiness. Moreover, the capabilities of TABB can be added to that mix, yielding tree automata capable of simultaneously testing global equalities, disequalities, and natural linear inequalities, as well as local positional constraints between brothers (*NParikh+EDB*), while preserving decidability of emptiness.

NParikh+ED

NParikh+EDB

Another interesting generalisation that appears – silently – in [Barguñó et al., 2010] is the fact that the disequality constraints are defined as $(2.8)_{[p36]}$, and not as (2.7). Thus they are not necessarily irreflexive, which is very useful in that it allows the expression of key constraints, for instance in the context of an XML database. The disequality constraint $q_{key} \not\cong q_{key}$ straightforwardly enforces the uniqueness of the subterms under each occurrence of q_{key}, yet such a constraint would have been invalid using the original definition of disequalities in [Filiot et al., 2008], for which irreflexivity was mandatory. It has been shown in [Vacher, 2010, Lemma 2.17] that allowing reflexive disequality constraints – and of course altering the definition of satisfaction to range over distinct positions, as done in $(2.8)_{[p36]}$ – strictly extends the expressive power of the model. We write *TAGErD* and *TAGrD* the classes TAGED and TAGD altered to use this new definition.

TAGErD
TAGrD

5.2.2 Rigid Tree Automata

While the original motivation under the development of tree automata with global constraints was the verification of semi-structured documents and databases, they have found applications beyond that domain. In particular, the subclass of *rigid tree automata* (*RTA*) [Jacquemard, Klay & Vacher, 2009, 2011; Vacher, 2010] has shown itself to be of interest for the verification of cryptographic protocols. An RTA is a TAGE whose constraints may only be diagonal, that is to say, $p \cong q \implies p = q$; the states q such that $q \cong q$, and the constraints themselves, are called *rigid*, hence the name of the class. This apparently benign restriction has consequences which make RTA an eminently practical subclass. First, it has the same expressive power as TAGE, as any TAGE can be "diagonalised" – the downside being that the construction is exponential. Many applications, for instance in the verification of cryptographic protocols, do not require the general notion of constraints; it is natural in that field to require subterms built by the same state to be equal. The comparative lack of conciseness of RTA compared to TAGE is therefore not necessarily an obstacle, nor even an inconvenience. Second, emptiness and finiteness become decidable in polynomial time – linear time for emptiness, which is a far cry from the ExpTime-completeness of the same problems for TAGE.

rigid tree automata
RTA

The intuition under that spectacular efficiency is that rigid states, unlike general equality constraints, do not interfere with the standard accessibility or marking algorithms used to decide emptiness. Such algorithms typically attempt to exhibit a witness, that is to say a term accepted by the automaton, which is built by associating to each state q a term t_q evaluating in q – marking q with that witness – until no new state can be so marked. The question is then whether any final state is marked. While there may be many possible witnesses – for instance a witness t_p of p may be a or b if $a \to p, b \to p \in \Delta$ – one of them suffices; with $f(p,p) \to q \in \Delta$, one may build $t_q = f(a,a)$ or $t_q = f(b,b)$, it does not matter which. By definition, the witness is accepted by a run where all states can be considered rigid, and whether the automaton is a BUTA or an RTA is therefore irrelevant.

The other decision problems mentioned in this section are not affected by the restriction to rigid constraints, nor are the closure and determinisation properties; all are identical for TAGED, TAGE, and RTA. Intersection-emptiness is ExpTime-complete, as for BUTA.

visibly rigid tree automata
VRTA

One weakness of RTA is that, like TAGED and TAGE, they cannot be determinised. To palliate that shortcoming, the class of *visibly rigid tree automata* (*VRTA*) is defined by analogy to visibly pushdown automata. Whether or not a state is rigid is made immediately visible by looking at the input symbol, by means of a partial function $\nu : \mathbb{A} \to \operatorname{dom} \cong$, such that for any rule $\sigma(p_1, \ldots, p_n) \to q \in \Delta$, either $\sigma \in \operatorname{dom} \nu$ and $q = \nu(\sigma)$, or $\sigma \notin \operatorname{dom} \nu$ and $q \in Q \setminus \operatorname{dom} \cong$. This restrictions allows VRTA to be determined; the construction is exponential in time and size. VRTA have other good properties in that there is a reasonably powerful class of TRS – linear and invisibly pushdown TRS – for which rewrite closure becomes decidable, in the sense that it can be tested whether $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, for a TRS $\mathcal{R}$ and a VRTA $\mathcal{A}$.

Linear and invisibly pushdown TRS are capable of encoding non-trivial cryptographic protocols, and the expressive power of VRTA renders exact a number of operations that methods based on regular tree languages have to approximate, thereby avoiding a number of false alarms. For instance, it becomes possible to model a local memory within which agents may store previously read messages.

5.3 Synthetic Taxonomy of Automata With Constraints

At this point, we have seen a large number of classes of automata, most of them extensions of BUTA with constraints-testing abilities. The appendix to this thesis also features other classes of closely related automata, namely DAG automata in Sec. 11.1.1[p191], and tree automata with one memory – and some variants – in Sec. 11.1.2[p193]. In Part IV, we shall meet a few more strains from the tree-walking family, this time without constraints. Figure 5.1[p116] offers a synthetic view of the classes of automata appearing in this thesis. Each node corresponds to a class, and transitions describe the superclass/subclass relation. There are three types of transitions: plain arrows

are used if the superclass is strictly more expressive than the subclass. When two classes are equally expressive, each of the mutual transitions is either dotted or dashed. A dashed transition means that there is a linear language-preserving transformation from one class to the other, whereas a dotted transition means that the transformation may lead to an exponential blow-up.

Since decidability of the emptiness problem is such an important criterion for a class of automata, the style of each node reflects the status of the corresponding class in that respect. There are four different styles:

They correspond respectively to: **(1)** maximal class for which emptiness is decidable; **(2)** subclass of a decidable class, which therefore inherits decidability from its parent – though it might have its own decidability proofs or more efficient decision

procedures; **(3)** class for which emptiness is decidable only in the complete and deterministic case; **(4)** class for which emptiness is undecidable.

Note that word automata appear along with tree automata in the figure; their expressive powers can be compared in the sense described in the first chapter, by seeing words as unary trees. Contrariwise, relationships between context-free word acceptors and tree acceptors do not appear: although one could see a PDA as more powerful than a BUTA, for instance, doing so would require a different viewpoint on words-as-trees, immaterial for this dissertation.

5.4 Notations: Modification of an Automaton

The next chapter features a number of somewhat heavy automata constructions. Here we define some notations to make them easier to understand and avoid repetitions.

The reader may have noticed that, in order to avoid having to give explicit tuple representations of automata, we have already taken to assume in this document that the attributes of an automaton $\mathcal{A}$, when left unspecified, are the usual $\langle \mathbb{A}, Q, F, \Delta, \cong, \not\cong, \ldots \rangle$. So Q, for instance, refers to the states of the automaton on which our interest is currently focused – unless otherwise stated. This is an obvious convention that simplifies exposition, but it is only unambiguous when dealing with automata one at a time. In order to keep some similar degree of convenience with constructions involving several automata, without opening the door to ambiguity, we define here a systematic "object-like' naming scheme for TAGED – since it is the class we use most. Any TAGED $\mathcal{X}$ is assumed to have attributes of the form $\langle \mathcal{X}{:}\mathbb{A}, \mathcal{X}{:}Q, \mathcal{X}{:}F, \mathcal{X}{:}\Delta, \mathcal{X}{:}\cong, \mathcal{X}{:}\not\cong \rangle$.

It is often convenient to describe an automaton as being almost the same as another one, except for one or a few attributes. The modification of an existing TAGED is written $\wr \mathcal{X} \mid$ <*modifs*>$\int$, where <*modifs*> is a comma-separated list of attribute changes. For brevity, within the scope of $\wr \mathcal{X} \mid \cdots \int$ any unqualified attribute x stands for $\mathcal{X}{:}x$. For instance, $\wr \mathcal{X} \mid \cong := \varnothing \int$ is the bare tree automaton associated with the TAGE $\mathcal{X}$, or $\mathrm{ta}(\mathcal{X})$. Modifications of the form "$x := f(x)$" will just be written "$f(x)$"; for instance $\wr \mathcal{X} \mid Q \setminus \{q\} \int$ is $\mathcal{X}$ from which the state q has been removed, as with "$Q := Q \setminus \{q\}$" (or even "$\mathcal{X}{:}Q := \mathcal{X}{:}Q \setminus \{q\}$"). Of course in this example the modification "$F \setminus \{q\}$" is completely omitted, as it is implied by "$Q \setminus \{q\}$", given that by definition $\mathcal{X}{:}F \subseteq \mathcal{X}{:}Q$. The same goes for the removal of all the rules of $\mathcal{X}{:}\Delta$ and constraints of $\mathcal{X}{:}\cong$ that used q.

Those notations are used in the next chapter, where we study the class of $\mathrm{TA}^{=}_{k}$, TAGE whose number of constraints is bounded by k.

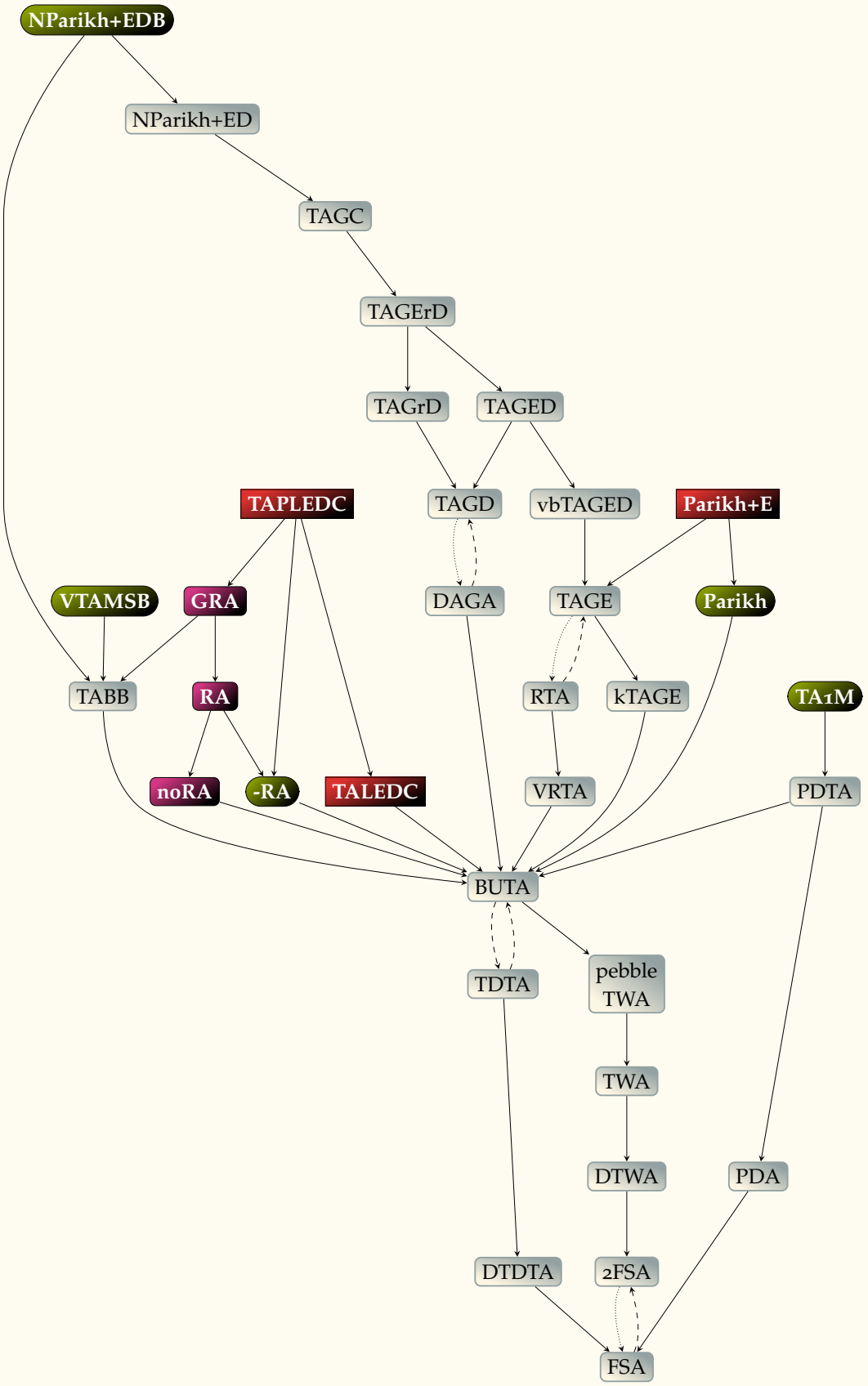


Figure 5.1: A taxonomy of automata, with or without constraints.

Chapter 6
Bounding the Number of Constraints

Contents

—Where economy of constraints does not save that much time...

Tree automata with global equality constraints – or TAGE, or $TA^{=}$ – are a central component of the model-checking approach developed in Part II of this thesis, and have many applications beyond that. Unfortunately, as discussed in the previous chapter on related automata families, the boon of expressive power which constraints provide is always tempered by a commensurate increase in algorithmic complexity, up to and including a loss of decidability. Yet, we have also seen that TAGE admit an equally expressive subclass – RTA, where all constraints are of the form $p \cong p$, cf. section 5.2.2[p113] – for which emptiness is testable in linear time.

In this chapter, we raise a question that is in some sense orthogonal to this observation: if restricting the *kind* of constraints which may be taken can have such a drastic effect on the complexity of some problems – from ExpTime-complete to linear for emptiness – what would be the effect of bounding the *number* of constraints instead? In the same way that some applications only require rigid constraints, it is often the case that one can get by with only a handful of general constraints. Can one therefore hope for efficient decision procedures in those cases? What are the respective parts played by the constraints and the size of the input data in the explosion of the complexity?

To answer this, we consider the classes of $TA^{=}_{k}$, for $k \in \mathbb{N}$. A $TA^{=}_{k}$ is a $TA^{=}$ whose number of constraints is at most k, that is to say, a $TA^{=}$ $\mathcal{A}$ is a $TA^{=}_{k}$ if it satisfies $\mathrm{Card}(\mathcal{A}{:}\cong) \leqslant k$. Assimilating classes of automata to the corresponding sets of automata, we have by definition the following strict chain:

$TA^{=}_{k}$: bounded $TA^{=}$

$$TA = TA^{=}_{0} \subset TA^{=}_{1} \subset \cdots \subset TA^{=}_{k} \subset TA^{=}_{k+1} \subset \cdots \subset TA^{=} = \bigcup_{k \in \mathbb{N}} TA^{=}_{k} \,. \tag{6.1}$$

The primary question is therefore to determine the algorithmic complexity of useful decision problems for each of the different classes in this chain. In the first section, we study the emptiness decision problem, which we show to be solvable in linear time with just one constraint, but ExpTime-complete so soon as at least two constraints are involved. A similar result then follows for finiteness. The second

section finds a different behaviour for the – NP-complete – membership decision problem, which remains in PTIME regardless of how many constraints there are – so long as there is a bound. Those results have been published in [Héam, Hugot & Kouchnarenko, 2012c].

$\mathcal{L}(C)$: *expressive power of* C

Another line of inquiry pertains to the expressive power of those classes: does bounding the number of constraints limit the recognisable languages, and if not, how many constraints does one need a minima? We have yet to define precisely what is meant by *expressive power*, though. If C is a class of automata on a certain alphabet $\mathbb{A}$ – that is to say, a set of automata on $\mathbb{A}$ – then we write $\mathcal{L}(C)$ the class of languages recognised by C, defined as

$$\mathcal{L}(C) = \{\ell \subseteq \mathcal{T}(\mathbb{A}) \mid \exists \mathcal{A} \in C : \mathcal{L}(\mathcal{A}) = \ell\}.$$

For instance, the class of regular languages is $\mathcal{L}(\mathrm{TA})$, and when we write that $\mathrm{TA}^=$, for instance, are strictly more powerful than BUTA, this amounts to stating the strict inclusion $\mathcal{L}(\mathrm{TA}) \subset \mathcal{L}(\mathrm{TA}^=)$. In such terms, by (6.1) we have obviously, for all $k \geqslant 0$, $\mathcal{L}(\mathrm{TA}^=_k) \subseteq \mathcal{L}(\mathrm{TA}^=_{k+1})$, but are all the inclusions strict? Or are they strict up to some rank? Furthermore, is there a $k \in \mathbb{N}$ such that $\mathcal{L}(\mathrm{TA}^=_k) = \mathcal{L}(\mathrm{TA}^=)$? Those are our secondary and tertiary questions, which are answered – positively for the one, and negatively for the other – in the third section of this chapter.

6.1 The Emptiness & Finiteness Problems

The complexity of emptiness and finiteness decision is tied to the number of constraints. We first deal with the case of $\mathrm{TA}^=_1$ which, as we shall see, admit a polynomial transformation into rigid tree automata – unlike the general case. The core of the argument is that the equality constraint can be simulated by an intersection of regular languages, and therefore with a product of tree automata. This holds in the case of one constraint because a single constraint cannot "nest with itself", in a sense which is made clearer by the following lemma:

∀ Lemma 6.1: Incomparable Positions

Let $\mathcal{A}$ be a $\mathrm{TA}^=$ with the constraint $p \cong q$, and ρ an accepting run of $\mathcal{A}$ on a tree t. Assume that both those states are involved in the run: $\{p, q\} \subseteq \operatorname{ran} \rho$; then any two distinct positions $\alpha, \beta \in \rho^{-1}(\{p, q\})$, $\alpha \neq \beta$, are incomparable: $\alpha \wedge \beta$.

Proof. Since $\alpha, \beta \in \rho^{-1}(\{p, q\})$ and $\{p, q\} \subseteq \operatorname{ran} \rho$ and $p \cong q$, we have $t|_\alpha = t|_\beta$ by definition of the satisfaction of the equality constraint $(2.6)_{[p35]}$. Suppose that α and β are comparable; for instance, assume wlog. that $\alpha \lhd \beta$. Then it follows immediately that $t|_\alpha \lhd t|_\beta$; this is absurd since $t|_\beta$ cannot be structurally equal to one of its own strict subterms. Therefore $\alpha \wedge \beta$. ∎

As mentioned before, every $\mathrm{TA}^=$ can be transformed into an equivalent RTA; the general construction of [Filiot, 2008, Lem. 5.3.5] is exponential, however – of the order of $2^{|Q|^2}$. Perhaps a better construction could be found, but let us note that regardless of possible optimisations, it would have to be exponential in the

general case. If there existed a sub-exponential language-preserving construction $\mathfrak{r} : \mathrm{TA}^{=} \to \mathrm{RTA}$, then it would be possible to test emptiness of any $\mathrm{TA}^{=}$ $\mathcal{A}$ in sub-exponential time, by computing $\mathfrak{r}(\mathcal{A})$ and testing its emptiness – which is decidable in linear time for RTA. Since emptiness decision is ExpTime-complete for $\mathrm{TA}^{=}$, such a procedure $\mathfrak{r}$ cannot exist – at least not under the usual assumptions about complexity classes. In any event, it is certain that no such $\mathfrak{r}$ can be polynomial, since the time hierarchy theorem implies that PTime $\subset$ ExpTime. Yet this need not apply when there is only one constraint, as we now show:

Lemma 6.2: Rigidification

For every $\mathrm{TA}^{=}_{1}$ $\mathcal{A}$, there exists an equivalent RTA $\mathcal{B}$ with at most one constraint, whose size is at most quadratic in that of $\mathcal{A}$.

Proof. If $\mathcal{A}$ has no constraints, or a rigid constraint ($p \cong p$), then no transformation is needed: $\mathcal{B} = \mathcal{A}$. Assume that $\mathcal{A}$ has a single constraint of the form $p \cong q$, with $p \neq q$.

Building Blocks. We define the construction in terms of smaller automata obtained by modification of $\mathcal{A}$:

$$\mathcal{B}^{\neg}_{p} = \wr \mathcal{A} \mid Q \setminus \{p\} \int \qquad \mathcal{B}_{p} = \wr \mathcal{B}^{\neg}_{q} \mid F := \{p\}, \Delta := \Delta_{p} \int$$
$$\mathcal{B}^{\neg}_{q} = \wr \mathcal{A} \mid Q \setminus \{q\} \int \qquad \mathcal{B}_{q} = \wr \mathcal{B}^{\neg}_{p} \mid F := \{q\}, \Delta := \Delta_{q} \int$$
$$\mathcal{B}_{pq} = \mathcal{B}_{p} \times \mathcal{B}_{q} \ ,$$

where Δ_p is $\mathcal{B}^{\neg}_{q} : \Delta$ from which all rules where p appears in the left-hand side have been removed, and Δ_q is defined symmetrically to Δ_p. $\mathcal{B}_{pq}$ is built to accept the intersection of the languages of $\mathcal{B}_p$ and $\mathcal{B}_q$ using the standard product algorithm; it has a single final state $q_f = (p, q)$. Note that all those building blocks are vanilla tree automata.

Construction. With this we define the rigid tree automaton

$$\mathcal{B} = \mathcal{B}^{\neg}_{p} \uplus \mathcal{B}^{\neg}_{q} \uplus \wr \mathcal{A} \mid Q', \Delta', q_f \cong q_f \int \ ,$$

with $Q' = (Q \setminus \{p, q\}) \uplus (\mathcal{B}_{pq} : Q)$ and $\Delta' = \Delta^{q_f}_{pq} \uplus (\mathcal{B}_{pq} : \Delta)$, where $\Delta^{q_f}_{pq}$ is $\mathcal{A} : \Delta$ from which all left-hand side occurrences of p or q have been replaced by q_f.

Equivalence. There remains to show that $\mathcal{B}$ is equivalent to $\mathcal{A}$. Let $t \in \mathcal{L}(\mathcal{A})$, accepted through a run ρ; then one of the following is true:

(1) neither p nor q appears in ρ,
(2) p appears, and q does not,
(3) q appears, and p does not,
(4) both p and q appear.

In the three first cases, the constraints are not involved, and t is accepted by: (1) both $\mathcal{B}^{\neg}_{p}$ and $\mathcal{B}^{\neg}_{q}$ (2) $\mathcal{B}^{\neg}_{q}$ (3) $\mathcal{B}^{\neg}_{p}$. In case (4), a subterm evaluating to p will belong to $\mathcal{L}^{p}(\mathcal{A})$ by definition, and also to $\mathcal{L}^{q}(\mathcal{A})$ as it needs to be equal to another extant subterm evaluating to q. Furthermore, p and q can only appear at the root of each subruns, lest $p \cong q$ be trivially violated. Therefore,

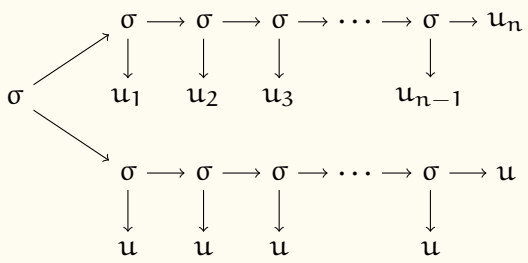


Figure 6.1: Reduction of intersection-emptiness: the language.

a successful run of $\mathcal{B}$ can be constructed by simply substituting all p and q subruns by q_f-runs of $\mathcal{B}_{pq}$. Thus $t \in \mathcal{L}(\mathcal{B})$.

Conversely, let $t \in \mathcal{L}(\mathcal{B})$; it is immediately seen by construction that $\mathcal{L}\left(\mathcal{B}_p^{\neg}\right) \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}\left(\mathcal{B}_q^{\neg}\right) \subseteq \mathcal{L}(\mathcal{A})$. Suppose that t is accepted through a run of the third and last part of $\mathcal{B}$ – namely $\wr\mathcal{A} \mid Q', \Delta', q_f \cong q_f\int$ – then every q_f-subrun can be replaced by either a p-run or a q-run of $\mathcal{A}$. The result of this operation is trivially an accepting run of $\mathrm{ta}(\mathcal{A})$; there remains to observe that it satisfies $p \cong q$, because the corresponding subtrees must be equal given the constraint $(q_f, q_f) \in \mathcal{B}{:}\cong$. Thus $t \in \mathcal{L}(\mathcal{A})$.

SIZE & TIME. All building blocks are of size $O(\|\mathcal{A}\|)$, except $\mathcal{B}_{pq}$, which is of size $O(\|\mathcal{A}\|^2)$. Globally, the size of $\mathcal{B}$ is therefore at most quadratic in that of $\mathcal{A}$. The construction is also straightforwardly done in quadratic time. ■

△ Proposition 6.3: Emptiness

The Emptiness problem is in PTIME for $\mathrm{TA}_1^{=}$, and EXPTIME-complete for $\mathrm{TA}_2^{=}$.

Proof. **$\mathrm{TA}_1^{=}$.** Emptiness is testable in linear time for RTA, therefore the emptiness of $\mathcal{A}$ is testable in quadratic time using the construction of Lemma 6.2.

$\mathrm{TA}_2^{=}$. OVERVIEW. We reduce the test of the emptiness of the intersection of n tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, which is an EXPTIME-complete problem, to the emptiness of a $\mathrm{TA}_2^{=}$ $\mathcal{A}$. This is similar to the arguments of [Filiot et al., 2008, Thm. 1], the major difference being that we can only use two constraints instead of an unbounded number of constraints. The idea is to take advantage of the fact that an explicit equality constraint between two positions effectively enforces an arbitrary number of implicit equality constraints on the subpositions.

ASSUMPTIONS. It is assumed without loss of generality that $n \geqslant 2$ and the sets of states of the $\mathcal{A}_i$ are pairwise disjoint; that is to say, $\forall i, j \in [\![1, n]\!]$, $i \neq j \Rightarrow (\mathcal{A}_i{:}Q) \cap (\mathcal{A}_j{:}Q) = \varnothing$. Furthermore, it can be assumed that each $\mathcal{A}_i$ has exactly one final state q_{fi}. If that is not the case, then $\mathcal{A}_i$ can be modified to be so, which results in its size doubling in the worst case.

LANGUAGE. We define the language ℓ as the set of trees of the form given in Figure 6.1[p120], where σ is a fresh binary symbol and for all i, $u_i \in \mathcal{L}(\mathcal{A}_i)$ and $u = u_i$. Note that this implies that $u \in \bigcap_i \mathcal{L}(\mathcal{A}_i)$, and therefore ℓ is empty if and only if $\bigcap_i \mathcal{L}(\mathcal{A}_i)$ is empty.

Automaton. We build a $\mathrm{TA}^{=}_{2}$ $\mathcal{A}$ that accepts ℓ, by first building a universal tree automaton $\mathcal{U}$, of final state q^u. Then, we let $\mathcal{A} = \langle \Sigma, Q, F, \Delta, \cong \rangle$, where

$$
\begin{aligned}
Q &= (\biguplus_i \mathcal{A}_i{:}Q) \uplus (\mathcal{U}{:}Q) \uplus \{\, q^u_1, \dots, q^u_{n-1}, q^v_1, \dots, q^v_{n-1} \,\} \uplus \{q_f\} \\
F &= \{q_f\} \qquad q^u \cong q^u, \quad q^u_1 \cong q^v_1 \qquad \Sigma = (\bigcup_i \mathcal{A}_i{:}\Sigma) \uplus \{\sigma/_2\} \\
\Delta &= \{\, \sigma(q^v_1, q^u_1) \to q_f \,\} \cup (\bigcup_i \mathcal{A}_i{:}\Delta) \cup (\mathcal{U}{:}\Delta) \cup \\
&\quad \{\, \sigma(q^u, q^u_{k+1}) \to q^u_k \mid k \in [\![1, n-2]\!] \,\} \cup \{\, \sigma(q^u, q^u) \to q^u_{n-1} \,\} \cup \\
&\quad \{\, \sigma(q_{fk}, q^v_{k+1}) \to q^v_k \mid k \in [\![1, n-2]\!] \,\} \cup \{\, \sigma(q_{fn-1}, q_{fn}) \to q^v_{n-1} \,\}\,.
\end{aligned}
$$

With regards to Fig. 6.1[p120], the u are accepted into q^u, and their equality is enforced by the rigid constraint on that state. The entire branch is accepted into q^u_1. As for the other branch, accepted in q^v_1, each u_i is recognised in q_{fi}, and thus $u_i \in \mathcal{L}(\mathcal{A}_i)$, for all i. By the constraint $q^u_1 \cong q^v_1$, both branches are identical, and thus for all i, $u_i = u$. Finally we have by construction $\mathcal{L}(\mathcal{A}) = \ell$, and $\|\mathcal{A}\| = O(\sum_{k=1}^{n} \|\mathcal{A}_i\|)$, which concludes the proof of ExpTime-hardness. Thus emptiness is ExpTime-complete. ∎

With this result, a similar conclusion can be drawn for the finiteness problem:

△ **Proposition 6.4: Finiteness**

The finiteness problem is in PTime for $\mathrm{TA}^{=}_{1}$, and ExpTime-complete for $\mathrm{TA}^{=}_{2}$.

Proof. **$\mathrm{TA}^{=}_{1}$.** Finiteness is testable in polynomial time for RTA – more precisely, in $O(\|A\| \cdot |Q|^2)$ according to the construction of [Filiot et al., 2010] – therefore the finiteness of $\mathcal{A}$ is testable in polynomial time using the transformation of Lemma 6.2. All in all, the above describes a decision procedure in $O(\|A\|^2 \cdot |Q|^4)$ – or $O(\|\mathcal{A}\|^6)$ to simplify – however this complexity can certainly be refined.

$\mathrm{TA}^{=}_{2}$. We reduce the emptiness problem for $\mathrm{TA}^{=}_{2}$ to the finiteness problem. Given a $\mathrm{TA}^{=}_{2}$ $\mathcal{A}$, we build

$$
\begin{aligned}
\mathcal{A}' = \wr\, \mathcal{A} \mid Q \uplus \{p\}, F := \{p\}, \Sigma \uplus \{\sigma/_1\}, \Delta' \int \\
\text{where } \Delta' = \Delta \cup \{\, \sigma(q_f) \to p \mid q_f \in F \,\} \cup \{\, \sigma(p) \to p \,\}\,.
\end{aligned}
$$

$\mathcal{A}'$ is also a $\mathrm{TA}^{=}_{2}$. If $\mathcal{A}$ accepts the empty language, then so does $\mathcal{A}'$. Conversely, if $t \in \mathcal{L}(\mathcal{A})$, then $\sigma^*(t) \subseteq \mathcal{L}(\mathcal{A}')$, and thus $\mathcal{L}(\mathcal{A}')$ is infinite. Consequently, the language of $\mathcal{A}'$ is finite if and only if that of $\mathcal{A}$ is empty. This, combined with Prp. 6.3[p120], shows that $\mathrm{TA}^{=}_{2}$-finiteness is ExpTime-hard; since the general problem for $\mathrm{TA}^{=}$ is ExpTime, $\mathrm{TA}^{=}_{2}$-finiteness is ExpTime-complete. ∎

6.2 The Membership Problem

Let us begin with some general observations and notations. We shall need to reason about the relation $\cong$; unfortunately, it is not an equivalence relation. For instance, given the constraints $p \cong r$ and $r \cong q$ it is syntactically tempting, but in general wrong, to infer $p \cong q$ by transitivity. The crux of the matter here is whether the state r actually appears in the run: if it does, $p \cong q$ is effectively implied, but if

it does not, then both constraints $p \cong r$ and $r \cong q$ are moot. Lemma 6.5 shows that, given the knowledge (or the assumption) of a set $P \subseteq \operatorname{dom} \cong$ of the constrained states which are actually present in runs, the constraints of $\cong$ are interchangeable with an equivalence relation, which we call the *togetherness* relation.

∀ Lemma 6.5: Togetherness

Let $\mathcal{A}$ be a $\mathrm{TA}^{=}$ and $P \subseteq \operatorname{dom} \cong$. Then any run ρ such that $(\operatorname{ran} \rho) \cap (\operatorname{dom} \cong) = P$ is accepting for $\mathcal{A}$ if and only if it is so for

$$\mathcal{A}_P = \wr \mathcal{A} \mid \cong := (\cong \cap P^2)^{\equiv} \int \ ,$$

where the equivalence closure is meant under $\operatorname{dom}(\cong \cap P^2)$.

Proof. Intuitively, this operation first removes all constraints which must be moot – because they involve states not in P – and then adds the constraints which can be deduced assuming all constrained states appear in the run. Formally, let ρ be an accepting run of $\mathrm{ta}(\mathcal{A})$ such that $(\operatorname{ran} \rho) \cap (\operatorname{dom} \mathcal{A}{:}\cong) = P$. Since $\mathcal{A}$ and $\mathcal{A}_P$ share their states and final states, constraints notwithstanding it can be seen as a run of either $\mathrm{TA}^{=}$, and it is accepting for $\mathcal{A}$ if and only if it is so for $\mathcal{A}_P$. Thus we only need to show that the constraints are compatible, that is to say, that if ρ is a run of $\mathcal{A}$ it satisfies $\mathcal{A}_P{:}\cong$, and that if it is a run of $\mathcal{A}_P$, it satisfies $\mathcal{A}{:}\cong$. In keeping with our usual notations we write simply $\cong$ for $\mathcal{A}{:}\cong$.

(1 : " $\Longrightarrow$ ") Assuming that ρ is a run of $\mathcal{A}$, it must satisfy the constraints $\cong$ by definition. We have trivially $(\cong \cap P^2) \subseteq (\cong)$, so a fortiori ρ must satisfy this subset of the constraints. There remains to show that the additional constraints introduced by the equivalence closure are satisfied as well.

SYMMETRY. The definition of the satisfaction (2.6)$_{[p35]}$ of an equality constraint $p \cong q$ is symmetric with respect to p and q, therefore it is trivial that whenever $p \cong q$ holds, then so does $q \cong p$. This does not depend on P and is not specific to this proof – one can assume constraints to be symmetric as a matter of course.

TRANSITIVITY. Suppose that $p \cong r$ and $r \cong q$ are satisfied, where $p, q, r \in P$. By our hypothesis on ρ, $P \subseteq \operatorname{ran} \rho$, and thus there exists in particular a position $\alpha_r \in \rho^{-1}(\{r\})$. For all possible positions $\alpha_p \in \rho^{-1}(\{p\})$ and $\alpha_q \in \rho^{-1}(\{q\})$, we have $t|_{\alpha_p} = t|_{\alpha_r}$ — to satisfy $p \cong r$ — and $t|_{\alpha_q} = t|_{\alpha_r}$ — to satisfy $q \cong r$. Thus for any α_p, α_q we have $t|_{\alpha_p} = t|_{\alpha_r} = t|_{\alpha_q}$, and $p \cong q$ is satisfied as well.

REFLEXIVITY. Suppose that $p \cong q$ holds, for $p, q \in P$. Again, there exists in particular a position $\alpha_q \in \rho^{-1}(\{q\})$. For any two $\alpha_p, \alpha'_p \in \rho^{-1}(\{p\})$, we need to have $t|_{\alpha_p} = t|_{\alpha_q}$ and $t|_{\alpha'_p} = t|_{\alpha_q}$ to satisfy $p \cong q$, and thus $t|_{\alpha_p} = t|_{\alpha'_p}$. Therefore $p \cong p$ holds.

(2 : " $\Longleftarrow$ ") Let us assume ρ to be a run for $\mathcal{A}_P$; again, by definition, it satisfies the constraints of $(\cong \cap P^2)^{\equiv}$. To show that it satisfies $\cong$, it suffices to verify that it complies with any constraint $(p, q) \in \cong \setminus (\cong \cap P^2)$, which is to say, any constraint such that either p or q is not in P. Suppose without loss of generality that $p \notin P$, and recall that $(\operatorname{ran} \rho) \cap (\operatorname{dom} \cong) \subseteq P$. Since $p \in \operatorname{dom} \cong$, we have $p \notin \operatorname{ran} \rho$, and $\rho^{-1}(\{p\}) = \varnothing$; it follows that $p \cong q$ is vacuously satisfied. ∎

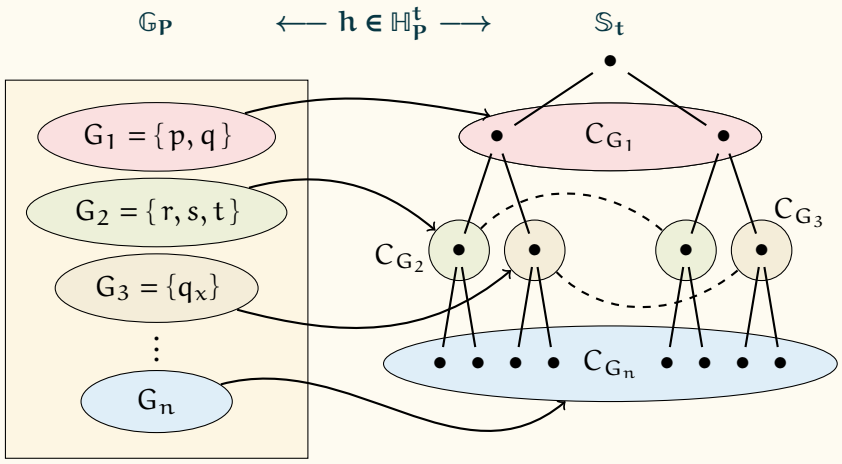


Figure 6.2: Housings: affecting a similarity classes to each group.

Let us take a notation for this equivalence relation: given a set $P \subseteq Q$, we write it

$$\asymp_P = (\cong \cap P^2)^{\equiv}\,, \tag{6.2}$$

and say that q *and* q' *are together with respect to* P if $q \asymp_P q'$. Its equivalence classes over the constrained states are denoted by

$\asymp_P$: *togetherness wrt.* P

$$\mathbb{G}_P = \frac{\mathrm{dom}(\asymp_P)}{\asymp_P} = \frac{\mathrm{dom}(\cong \cap P^2)}{(\cong \cap P^2)^{\equiv}}$$

and called *groups*. Note again that only a subset P of the constrained states which actually appear in the run have any real influence: that subset is $\bigcup \mathbb{G}_P = \mathrm{dom}(\cong \cap P^2)$. The others are part of constraints which are moot given P. If t is a tree, we write $\sim$ (or $\sim_t$ when the tree under consideration is not obvious) for the *similarity relation on* t, defined on $\mathcal{P}(t)^2$ such that $\alpha \sim \beta \iff t|_\alpha = t|_\beta$, and build the quotient set

$\mathbb{G}_P$: *groups of states for* $\asymp_P$

$\sim$: *similarity relation*

$$\mathbb{S}_t = \frac{\mathcal{P}(t)}{\sim}\,,$$

which we call the *similarity classes of* t. With this, we can outline a polynomial algorithm for testing membership, which is developed in the next lemma and proposition. The idea is that given P, and in order to satisfy the constraints, there must be a way to "house" each group of $\mathbb{G}_P$ into the tree t, in the sense that all states of a same group must be affected by the run to positions in the same similarity class. There are finitely many such arrangements, thus we can simply test them all; all that we need is to show that this can be done in polynomial time. To summarise, the approach is in four iterated steps:

$\mathbb{S}_t$: *similarity classes of* t

(1) Choose some $P \subseteq \mathrm{dom} \cong$ – all are eventually chosen.
(2) Given P, turn $\cong$ into the equivalence relation $\asymp_P$.
(3) Try all possible *housings* of $\mathbb{G}_P$ into $\mathbb{S}_t$.
(4) For each such housing, try to build an accepting run around it.

The next lemma begins to describe this notion of housing more precisely:

∀ Lemma 6.6: Housing Groups

Let $\mathcal{A}$ be a TA$^{=}$, $P \subseteq \operatorname{dom} \cong$ and ρ a run of $\mathrm{ta}(\mathcal{A})$ on a tree t, such that $(\operatorname{ran} \rho) \cap (\operatorname{dom} \cong) = P$. Then ρ satisfies the constraints of $\cong$ if and only if each group can be assigned a similarity class, such that all states of that group appear within this class in the run. Formally: $\forall G \in \mathbb{G}_P,\ \exists C_G \in \mathbb{S}_t : \rho^{-1}(G) \subseteq C_G$.

Proof. Let $G \in \mathbb{G}_P$, and ρ as above.

(1 : " $\Longrightarrow$ ") Assume that ρ satisfies $\cong$. Then by Lem. 6.5[p122], it satisfies $\asymp_P$. Let any $p, q \in G$; we have $p \asymp_P q$ by definition of $\mathbb{G}_P$, and thus for all $\alpha_p \in \rho^{-1}(\{p\})$ and $\alpha_q \in \rho^{-1}(\{q\})$, $t|_{\alpha_p} = t|_{\alpha_q}$. Or, using another notation, $\alpha_p \sim \alpha_q$. We let $C_G = [\alpha_p]_\sim = [\alpha_q]_\sim$. Since $\rho^{-1}(G) = \bigcup_{g \in G}(\rho^{-1}(\{g\}))$, any $\alpha \in \rho^{-1}(G)$ is such that $\exists g \in G$ which satisfies $\alpha \in \rho^{-1}(\{g\})$, and $p \asymp_P g$; thus $\alpha \in [\alpha_p]_\sim = C_G$.

(2 : " $\Longleftarrow$ ") Consider any constraint $p \asymp_P q$; the states p and q belong to the same group $G \in \mathbb{G}_P$, and thus by the hypothesis there exists a similarity class $C_G \in \mathbb{S}_t$ such that $\rho^{-1}(\{p, q\}) \subseteq \rho^{-1}(G) \subseteq C_G$. This in turn implies that for all $\alpha_p \in \rho^{-1}(\{p\})$, $\alpha_q \in \rho^{-1}(\{q\})$, $\alpha_p \sim \alpha_q$, or in other words: $t|_{\alpha_p} = t|_{\alpha_q}$. Thus ρ satisfies $p \asymp_P q$; and since the choice of this constraint was arbitrary, it satisfies $\asymp_P$. Therefore, invoking Lem. 6.5 a second time, ρ satisfies $\cong$. ∎

$\mathbb{H}^t_P$: P-*housings on* t

It is such a mapping $G \mapsto C_G$ which we call a *housing*. More generally, any map from $\mathbb{H}^t_P = \mathbb{G}_P \to \mathbb{S}_t$ is a housing, in the sense that it affects groups of constrained states to similarity classes in the tree – cf. Fig. 6.2. However, a housing is only interesting if it is possible to build a run around it. A housing $h \in \mathbb{H}^t_P$ is *compatible* with a run ρ – and vice versa – if the conditions of the previous lemma are satisfied, which is to say:

$$\forall G \in \mathbb{G}_P,\ \rho^{-1}(G) \subseteq h(G)\ .$$

With this in mind, we can now make explicit the algorithm outlined above, while counting the overall number of operations required.

△ Proposition 6.7: Membership

Given an arbitrary but fixed $n \in \mathbb{N}$, the Membership problem for TA$^{=}_n$ is in PTIME — albeit with an overhead exponential in n.

Proof. Let $\mathcal{A}$ be a TA$^{=}_n$, and t a tree. The Housing Lemma (Lem. 6.6) has already established that a run ρ of $\mathcal{A}$ on t satisfies $\cong$ if and only if there exists a housing $h \in \mathbb{H}^t_P$ which is compatible with ρ, where $P = (\operatorname{dom} \cong) \cap (\operatorname{ran} \rho)$ is the set of constrained states which actually appear in the run. Our strategy to check the membership of t is simply to try each possible $P \subseteq \operatorname{dom} \cong$ successively, by attempting, for each possible housing $h \in \mathbb{H}^t_P$, to craft an accepting run ρ of $\mathrm{ta}(\mathcal{A})$ compatible with h. There are at most 2^{2n} possible P, and given a choice of P, there are $|\mathbb{S}_t|^{|\mathbb{G}_P|} \leqslant \|t\|^{2n}$ P-housings on t, which gives at most $4^n \cdot \|t\|^{2n}$ tests in total. Note that since n is a constant, this remains polynomial. There only remains to show that given a choice of P and $h \in \mathbb{H}^t_P$, the existence of a

compatible run can be tested in polynomial time. To do so, we use a variant of the standard reachability algorithm, where the only constrained states which may appear are those of P, and the states of a given group $G \in \mathbb{G}_P$ may only appear at the positions assigned to them by the chosen housing h. Formally, given a choice of P and a housing $h \in \mathbb{H}_P^t$, there exists such a run if and only if $\Phi_t^{P,h}(\varepsilon) \cap F \neq \varnothing$, where

$$\Phi_t^{P,h}(\alpha) = \left\{ q \in Q \;\middle|\; \begin{array}{l} t(\alpha)(p_1, \dots, p_n) \to q \in \Delta \\ \forall i \in [\![1, n]\!],\ p_i \in \Phi_t^{P,h}(\alpha.i) \\ q \in \bigcup \mathbb{G}_P \implies \alpha \in h([q]_{\asymp_P}) \\ q \notin \mathrm{dom}(\cong) \setminus P \end{array} \right\}.$$

The reader will notice that, were the last two conditions removed, $\Phi_t^{P,h}(\alpha)$ would simply be the set of reachable states at position α. The additional two constraints are polynomial operations, thus $\Phi_t^{P,h}(\cdot)$ does run in polynomial time; there only remains to show that our algorithm does what is expected of it. There are two points to this: **(1)** no false negative: every successful run is subsumed by some $\Phi_t^{P,h}(\cdot)$ **(2)** no false positive: every run subsumed by some $\Phi_t^{P,h}(\cdot)$ is successful.

(1) Let ρ be a successful run for $\mathcal{A}$, and $P = (\mathrm{ran}\,\rho) \cap (\mathrm{dom} \cong)$; then by Lem. 6.5 and the Housing Lemma, it satisfies $\asymp_P$, and there exists a housing $h \in \mathbb{H}_P^t$ with which it is compatible. We propose that ρ is subsumed by $\Phi_t^{P,h}(\cdot)$, which is to say that for each position $\alpha \in \mathcal{P}(t)$, we must have $\rho(\alpha) \in \Phi_t^{P,h}(\alpha)$. Indeed, let α be any position, and $q = \rho(\alpha)$; we check that q satisfies all four conditions for belonging to $\Phi_t^{P,h}(\alpha)$. The first condition is trivially satisfied since ρ is a run. The second one will be the hypothesis of our recursion which, quite conveniently, evaluates to true vacuously if α is a leaf. The third condition is taken care of by the Housing Lemma: suppose $q \in \bigcup \mathbb{G}_P$; then there is a group $G \in \mathbb{G}_P$ such that $q \in G$ (in fact $G = [q]_{\asymp_P}$), and $\rho^{-1}(G) \subseteq h(G)$. Thus we have the chain $\alpha \in \rho^{-1}(\{q\}) \subseteq \rho^{-1}(G) \subseteq h(G)$, and in particular $\alpha \in h([q]_{\asymp_P})$. The fourth and last condition is trivial given our choice of P: Assuming its negation $q \in \mathrm{dom}(\cong) \setminus P$, it follows that $q \notin \mathrm{ran}\,\rho$, which is absurd.

(2) Let ρ be a run subsumed by $\Phi_t^{P,h}(\cdot)$, for some P and h. By the fourth condition, $(\mathrm{ran}\,\rho) \cap (\mathrm{dom}(\cong) \setminus P) = \varnothing$, and thus $(\mathrm{ran}\,\rho) \cap (\mathrm{dom} \cong) \subseteq P$. Let $\alpha \in \mathcal{P}(t)$; by the third condition, if $\rho(\alpha) \in G \in \mathbb{G}_P$, then $\alpha \in h(G)$; in other words, $\rho^{-1}(G) \subseteq h(G)$, thus by the Housing Lemma ρ is successful. The watchful reader will notice that a more precise formulation of the lemma is required to assert that, because Lem. 6.6$_{[p124]}$ as written requires $(\mathrm{ran}\,\rho) \cap (\mathrm{dom} \cong) = P$. The inclusion is actually sufficient for the "if" part, as shown by the relevant halves of the proofs of Lem. 6.6 and Lem. 6.5$_{[p122]}$. Alternatively, one could replace P and h by adequate $P' \subseteq P$ and $h' \in \mathbb{H}_P^t$, such that we have equality and preserve subsumption. Either way this is an easy technicality which only comes into play at this point of the proof. ∎

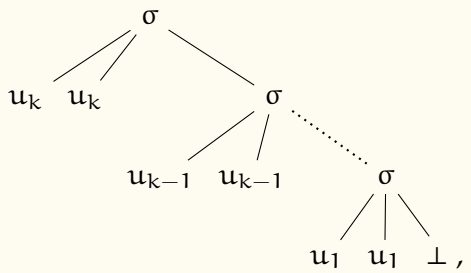
6.3 A Strict Hierarchy

We now turn our attention to our secondary questions regarding the expressive power of $\mathrm{TA}^{=}_{k}$. The simplest approach to solve this is to exhibit a family of languages $L = (\ell_k)_{k\in\mathbb{N}}$ such that encoding ℓ_k requires at least k equality constraints. The intuition which guides us in the search for such a separation language is that, if there are k subterm equalities in terms of the language, and all those equalities are independent from one another, then k distinct constraints will be required, because using a constrained state q to enforce two different equalities means breaking their independence. To capitalise upon this informal idea, we work with the ranked alphabet $\biguplus_{i=1}^{k} \mathbb{A}_i \uplus \{\sigma/_3, \bot/_0\}$, where $\mathbb{A}_i = \{a_i, b_i/_0, f_i, g_i/_2\}$, and define L such that

(1) $\ell_0 = \{\bot\}$

(2) $\ell_k = \{\sigma(u, u, t_{k-1}) \mid u \in \mathcal{T}(\mathbb{A}_k), t_{k-1} \in \ell_{k-1}\}$, for $k \geqslant 1$.

More graphically, ℓ_k is the language of all terms of the general form



(6.3)

with $u_i \in \mathcal{T}(\mathbb{A}_i)$, for all i. We can already note that ℓ_1 is virtually identical to $L_=$, and thus is a non-regular language easily recognisable using one global equality constraint. In other words, we have $\ell_1 \in \mathcal{L}(\mathrm{TA}^{=}_{1}) \setminus \mathcal{L}(\mathrm{TA})$, and there remains to show that the same is true at every rank, which is to say that

$$\ell_k \in \mathcal{L}(\mathrm{TA}^{=}_{k}) \setminus \mathcal{L}\left(\mathrm{TA}^{=}_{k-1}\right), \quad \forall k \geqslant 1\,. \tag{6.4}$$

Proof. The positive part — $\ell_k \in \mathcal{L}(\mathrm{TA}^{=}_{k})$ — is easy to justify, as it suffices to exhibit automata $\mathcal{A}_k \in \mathrm{TA}^{=}_{k}$ such that $\mathcal{L}(\mathcal{A}_k) = \ell_k$. The construction is immediate by generalisation of the TAGE accepting $L_=$ given in section 2.5[p35]. Letting the $\mathcal{U}_i \in \mathrm{TA}$ be universal tree automata such that $\mathcal{U}_i{:}F = \{q^u_i\}$ for all i, $\mathcal{A}_k$ is defined with

$$Q = \{q^v_0\} \uplus \biguplus_{i=1}^{k} \mathcal{U}_i{:}Q \uplus \{q^v_i\} \qquad F = \{q^v_1\} \qquad q^u_i \cong q^u_i,\ \forall i \in [\![1, k]\!]$$
$$\Delta = \left\{ \sigma(q^u_i, q^u_i, q^v_{i-1}) \to q^v_i \;\middle|\; i \in [\![1, k]\!] \right\} \cup \{\bot \to q^v_0\}\,.$$

The negative part — $\ell_k \notin \mathcal{L}\left(\mathrm{TA}^{=}_{k-1}\right)$ — requires a bit more work. Let us take the notation $\mathrm{acs}\,\rho$ for the *active constrained states* of a run ρ, defined as

$\mathrm{acs}\,\rho$: *active constrained states*

$$\mathrm{acs}\,\rho = \{\rho(\alpha) \mid \alpha \in \mathcal{P}(\rho), \exists \beta \in \mathcal{P}(\rho) \setminus \{\alpha\} : \rho(\alpha) \cong \rho(\beta)\}\,. \tag{6.5}$$

That is to say, a state is considered active if it is a constrained state which not only appears in the run, but actually involves the constraint, because there appears in

the run at least one instance of its partner state – possibly another instance of itself. For example, even if $p \cong p$, one lone instance of p in the run is not enough for the constraint to actually do anything. One needs appearances of p at two distinct positions before it is considered active.

Let us assume for a moment that there is an automaton $\mathcal{A} \in \mathrm{TA}^{=}_{k-1}$ that recognises ℓ_k. We first make the observation that there is no possible execution ρ of $\mathcal{A}$ such that any active constrained state appears on the spine of the term. Formally, for any execution ρ, there are no distinct positions α and β such that $\alpha \in 3^*$ and $\rho(\alpha) \cong \rho(\beta)$. Indeed, assuming that to be the case, given $\alpha \in 3^*$, either β is also on the spine 3^* – in which case α and β are comparable, and Lem. 6.1[p118] is violated – or it is not on the spine, in which case the subterms under α and β cannot be equal, because one is rooted in σ and the other cannot be.

We are using the standard notations for regular expressions as a shorthand for sets of positions. e.g. $3^*(1+2) = \{1, 2, 31, 32, 331, 332, \dots\}$.

This remark will come in handy in a short time. Meanwhile, it holds in particular that $\mathcal{A}$ accepts a term $t \in \ell_k$ such that, in the terms of the general form (6.3), $|u_i| > |Q|$, for all i. By this we mean more precisely that $\left|t|_\alpha\right| > |Q|$, for all $\alpha \in 3^*(1+2)$. Suppose now that there exists an accepting run ρ of $\mathcal{A}$ on t such that at least one of the u_i – either a first-child or a second-child instance – is accepted without ever using any active constrained state. That is to say, there exists a position $\alpha \in 3^*(1+2)$ such that $\operatorname{ran} \rho|_\alpha \cap \operatorname{acs} \rho = \varnothing$. Since, by the above remark, there cannot be any active constrained state on the spine either, there is overall no position in the subterm u_i involved by ρ in any equality test, whether directly or indirectly as a consequence of an ancestor's involvement in such. Thus, as far as $t|_\alpha$ is concerned, $\mathcal{A}$ behaves exactly as a run-of-the-mill BUTA, and this means that the pumping lemma applies as usual. Since we have conveniently chosen t such that $\left|t|_\alpha\right| > |Q|$, that means we may pump ρ under α – it doesn't matter in which direction – to obtain a new run ρ'. Since ρ is final, so is ρ', and the constraints are still satisfied, as none of the states involved in the pumping are active. Through ρ', $\mathcal{A}$ recognises a new term $t' \neq t$, identical to t except under α. Suppose without loss of generality that $\alpha = \beta.1$, for some $\beta \in 3^*$; then $t'|_{\beta.1} \neq t'|_{\beta.2}$. Thus $t' \notin \ell_k$, and $t' \in \mathcal{L}(\mathcal{A})$, which is of course a contradiction. From this we conclude that all accepting runs of $\mathcal{A}$ on t must involve at least one active constrained state under each of the $t|_\alpha$, with $\alpha \in 3^*(1+2)$.

This observation, combined with a counting argument, clinches the proof. Indeed, consider an accepting run ρ of $\mathcal{A}$ on t and – using (6.3) – subterms u_i and u_j of t, with $i \neq j$. It does not matter whether one considers the first-child or second-child instances. By the previous paragraph, there must be active constrained states p_i and p_j, appearing in the subruns on u_i and u_j, respectively. Their partner states q_i and q_j must also appear somewhere in ρ, by dint of them being active. Suppose that q_i appears in the subrun on u_j. Then there exist $s_i \trianglelefteq u_i$ and $s_j \trianglelefteq u_j$ such that $s_i = s_j$. But $u_i \in \mathcal{T}(\mathbb{A}_i)$ and $u_j \in \mathcal{T}(\mathbb{A}_j)$, thus $s_i \in \mathcal{T}(\mathbb{A}_i)$ and $s_j \in \mathcal{T}(\mathbb{A}_j)$, and since the alphabets are disjoint by definition, $\mathcal{T}(\mathbb{A}_i) \cap \mathcal{T}(\mathbb{A}_j) = \varnothing$. Thus $s_i = s_j \in \varnothing$, which is absurd. We must conclude that q_i may only appear under u_i itself, or under its brother, but not at a different "level". In a slightly more precise language, if $\rho(\alpha) \cong \rho(\beta)$, then there exist $\gamma \in 3^*$ such that $\alpha \trianglelefteq \gamma.1$ and $\beta \trianglelefteq \gamma.2$. So, whenever a constrained state is used in a level, neither it nor its partner state may be used in any other levels. And, as was shown above, each level uses at least one constrained state. There are k levels by definition of ℓ_k, and only $k-1$ constraints, by definition

of $\mathcal{A}$. Therefore ρ cannot exist, and $\mathcal{A}$ cannot accept ℓ_k. This concludes the proof of (6.4). ■

△ Proposition 6.8: Strict Hierarchy

The $\mathrm{TA}^{=}_{k}$ form a strict hierarchy of expressive powers:

$$\mathcal{L}(\mathrm{TA}^{=}_{0}) \subset \mathcal{L}(\mathrm{TA}^{=}_{1}) \subset \cdots \subset \mathcal{L}(\mathrm{TA}^{=}_{k}) \subset \mathcal{L}(\mathrm{TA}^{=}_{k+1}) \subset \cdots \subset \mathcal{L}(\mathrm{TA}^{=}) \,.$$

Proof. All the groundwork for this proof has been done above. Let $k \geqslant 0$. By (6.1) we have $\mathcal{L}(\mathrm{TA}^{=}_{k}) \subseteq \mathcal{L}(\mathrm{TA}^{=}_{k+1})$, and by (6.4) the inclusions are strict. ■

6.4 Summary and Conclusions

In the case of emptiness and finiteness testing we have shown that, perhaps somewhat counter-intuitively, and despite the loss of expressive power incurred by bounding the number of constraints, the full complexity of the general, unbounded problem comes into play as soon as two constraints are involved. While this is unfortunate, there are a number of interesting cases which can be handled using only one constraint – even if one may need to break down a problem in several independent cases, each expressible with $\mathrm{TA}^{=}_{1}$, and deal with them separately. This can be the case, for instance, if no nesting of constraints is required to encode the property under consideration. More practically, one may want to define a class of $\mathrm{TA}^{=}$ with several constraints, but where constraints are not allowed to nest in a run, and such that every class of the togetherness relation $(6.2)_{[p123]}$ on the active constrained states $(6.5)_{[p126]}$ of any run is of a cardinality bounded by some integer m. This would allow for polynomial time emptiness decision, while being enough for some purposes such as – possibly – one-step rewriting. We discuss that idea in a bit more detail in the general perspectives, Part V. In the general case, generating rigid constraints inasmuch as possible and transforming into rigid tree automata before testing appears to be the most viable strategy, since the exponential cost is unavoidable either way.

This stands in contrast to the behaviour of the membership problem which, while NP-complete in general, becomes polynomial once the number of constraints is bounded by a constant, regardless of the size of that constant – though admittedly "polynomial" is in that case quite unlikely to mean "efficient" for anything but the smallest constants. Nevertheless, this suggests a potentially more scalable alternative to the existing general SAT encoding approach of [Héam, Hugot & Kouchnarenko, 2010b], which we present briefly in the next chapter.

Acknowledgements. The anonymous reviewers of the original version of [Héam, Hugot & Kouchnarenko, 2012c], which covered sections 6.1 and 6.2, provided many useful suggestions, in particular regarding the $\mathrm{TA}^{=}_{1}$ parts of Propositions 6.3 and 6.4, which became Lemma 6.2 in the final version of the paper.

Chapter 7
SAT Encodings for TAGED Membership

Contents

—Where somebody else does all the hard work.

The uniform membership problem for TAGED is NP-complete. We have already been reminded of the lower bound at the end of section 2.5[p35], by the encoding (2.9)[p37] of formulæ of propositional logic, which shows that deciding whether a formula is satisfiable reduces to testing TAGED membership – hence the NP-hardness. As for the upper bound, it is easy to see that a run can be guessed nondeterministically, and tested to be accepting in polynomial time. As an NP-complete problem, the TAGED membership test is subject to the reverse operation, in that it can in turn be reduced, in polynomial time and space, to the boolean satisfiability problem. For the sake of self-containedness, let us state that problem explicitly and say a few words of its importance in computer science.

boolean satisfiability problem

The *boolean satisfiability problem*, or *SAT problem*, consists in determining whether, for a given formula φ, there exists a *valuation* ν – also called an *interpretation* – which satisfies it, i.e. such that the formula evaluates to true. This is written $\nu \models \varphi$. It is the first known NP-complete decision problem. Before it was proven to be so by Cook in 1971, the notion of NP-completeness did not even exist. Since then, a tremendous amount of research has gone into crafting highly optimised heuristics for solving this problem, and into implementing them efficiently in specialised tools, aptly called *SAT solvers*. Let us just mention two among them, which we shall meet again in the experimental part of the present chapter: PicoSAT [Biere, 2008] and MiniSAT2 [Eén & Sörensson, 2003]. Those efforts were successful enough that modern SAT solvers are generally capable of dealing expediently with huge formulæ, ranging in the hundred thousands of free variables, and even in the millions.

SAT solvers
PicoSAT
MiniSAT2

NP-complete decision problems used to be generally considered intractable in practice, but while it is true that naive approaches are unlikely to scale, the tremendous practical prowess of modern SAT solvers challenges that conception to a degree. Since any NP-complete decision problem can, by definition, be polynomially reduced to an instance of the SAT problem, encoding a new NP-complete problem

into SAT, and then solving it using a SAT solver, has arisen as a general and viable vector of attack. Instead of spending much time determining, validating and implementing specific heuristics for each new NP-complete problem, this method takes advantage of all the hard work and sophisticated optimisations that went into SAT solvers in almost forty years of active research. This modus operandi was first introduced in [Clarke, Biere, Raimi & Zhu, 2001], where it is applied to bounded model checking – already mentioned at the end of section 1.1[p10] – which proved highly successful.

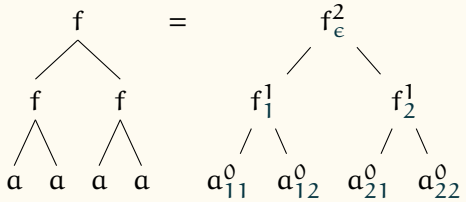
In this chapter, we present a SAT encoding of the membership problem for TAGED. Of course, the overarching goal is to rely on the performance of SAT solvers to decide membership efficiently. Experiments were conducted, using small examples and the two solvers mentioned above. The main results of this chapter have been published in [Héam, Hugot & Kouchnarenko, 2010b], though the presentation is more thorough in this thesis.

7.1 Propositional Encoding

This section presents our propositional encoding of the membership problem, which is justified step by step. We shall also illustrate our sub-formulæ as we go along by instantiating them on a small example. For this purpose we use the TAGED $\mathcal{A}$, such that $\mathbb{A} = \{ a/_0, f/_2 \}$, $Q = \{ q, \hat{q}, q_f \}$, $F = \{ q_f \}$, with the constraints $\hat{q} \cong \hat{q}$ and $\hat{q} \ncong q_f$, and the transitions

$$\Delta = \{ f(\hat{q}, \hat{q}) \to q_f,\ f(q, q) \to q,\ f(q, q) \to \hat{q},\ a \to q,\ a \to \hat{q} \}\,. \tag{7.1}$$

The reader will notice that this is almost the same automaton as that accepting $L_=$, given in section 2.5, the only difference being the addition of the disequality constraint $\hat{q} \ncong q_f$, which is of course redundant and moot, and used purely for illustrative purposes. Thus we have $\mathcal{L}(\mathcal{A}) = L_=$. We shall also make use of the following annotated term:

$t =$  . (7.2)

In the annotated term, the subscripts are of course the positions and the superscripts are unique references to the structure of the subterms. On this example, we have the following mapping:

$$0 \mapsto a \quad 1 \mapsto f(a, a) \quad 2 \mapsto f(f(a, a), f(a, a)) = t\,. \tag{7.3}$$

This mapping will come in useful later on, when the need arises to speak about the structural equality or difference of subterms.

The principle of the encoding is to translate the definition of an accepting run for TAGED in terms of propositional logic. Let us summarise the conditions which need to be satisfied in order for some term t to be accepted by a TAGED $\mathcal{A}$ through a run ρ, and break them down in sub-conditions until we can encode them:

(1) The run ρ is *successful* for the underlying tree automaton $\mathcal{A}' = \langle \mathbb{A}, Q, F, \Delta \rangle$.
  a. The run ρ is a function mapping positions of t to states of $\mathcal{A}$:
    i. $\rho \subseteq \mathcal{P}(t) \times Q$,
    ii. $\forall \alpha \in \mathcal{P}(t), p, q \in Q;\ (\alpha, p) \in \rho \wedge (\alpha, q) \in \rho \implies p = q$,
    iii. $\forall \alpha \in \mathcal{P}(t), \exists q \in Q;\ (\alpha, q) \in \rho$.
  b. The run ρ must be compatible with the rules of Δ: $(2.3)_{[p33]}$.
  c. The run ρ must be *accepting*, i.e. $\rho(\varepsilon) \in F$.
(2) It must satisfy the global equality constraints in $\cong$: $(2.6)_{[p35]}$.
(3) It must satisfy the global disequality constraints in $\ncong$: (2.7) or (2.8).

The first important point, expressed by condition (**1(a)i**), is that a run – as any function – is a relation, in this case between positions and states. This suggests to choose the building blocks of our formula as variables of the form X_q^α, taken from the set of propositional variables $\mathbb{X}$ and imbued with the intuitive meaning that at a position $\alpha \in \mathcal{P}(t)$, the run takes us into the state $q \in Q$. Thus, intuitively,

X_q^α: *variable:* $(\alpha, q) \in \rho$

$$X_q^\alpha \equiv \text{“}\rho \text{ exists and } \rho(\alpha) = q\text{”} .$$

This can be made more precise, by introducing a specific correspondence between valuations and relations. We define the higher-order mapping

$$\varpi(\cdot) : \left| \begin{array}{ccc} \mathbb{X} \to \mathbb{B} & \longrightarrow & \wp(\mathcal{P}(t) \times Q) \\ \nu & \longmapsto & \left\{ \alpha \mapsto q \;\middle|\; \nu(X_q^\alpha) = \top \right\} \end{array} \right. ,$$

where $\mathbb{B}$ is of course the set of boolean values. By this definition, if ν is a valuation, then $\rho = \varpi(\nu)$ is the corresponding relation between positions of t and the states. We extend this notation to formulæ; if φ is a formula, then $\varpi(\varphi)$ is defined as

$$\varpi(\varphi) = \{\, \varpi(\nu) \mid \nu \models \varphi \,\} .$$

That is to say, $\varpi(\varphi)$ is the set of relations which are compatible with, or described by, the formula φ. The aim of the game is to come up with a formula $\Theta_{\mathcal{A}}(t)$ such that $\varpi(\Theta_{\mathcal{A}}(t))$ is exactly the set of all the accepting runs of $\mathcal{A}$ on t. We start with the least restrictive formula possible, $\top$, such that $\varpi(\top) = \wp(\mathcal{P}(t) \times Q)$, which is to say that everything is possible, and we shall progressively sculpt this undistinguished block into the accepting runs, by knocking off everything that violates any of the conditions listed above. Each blow of our chisel will take the form of an additional conjunctive clause.

By the above, we have already coded condition (**1(a)i**), as $\top$ describes the set of all relations. Now we must implement the necessary restrictions to confine ourselves to the set of functions. Condition (**1(a)ii**) encodes the fact that ρ is a functional relation, i.e. a partial function. However, it is not expressed in a way which maps nicely to our choice of variables; we need something expressible in terms of positive and negative literals; fortunately, (**1(a)ii**) can equivalently be reformulated as

$$\forall \alpha \in \mathcal{P}(t), p \neq q \in Q;\ (\alpha, p) \in \rho \implies (\alpha, q) \notin \rho , \tag{7.4}$$

and, with another optional step, into our preferred form

$$\forall \alpha \in \mathcal{P}(t), q \in Q;\ (\alpha, q) \in \rho \implies \forall p \in Q \setminus q,\ (\alpha, p) \notin \rho , \tag{7.5}$$

which translates nicely into the *partial function constraint* $\Omega_{\nrightarrow}$:

$\Omega_{\nrightarrow}$: *partial function constraint*

$$\Omega_{\twoheadrightarrow} = \bigwedge_{\substack{\alpha\in\mathcal{P}(t)\\ q\in Q}} \left[X_q^\alpha \implies \bigwedge_{\substack{p\in Q\\ p\neq q}} \neg X_p^\alpha \right].$$

All instantiations of the formulæ on our running example are elided versions of the LaTeX 2_ε output of our implementation. See the experimental section, for instance Fig. 7.4[p140].

Applied to our minimalist example (7.1) & (7.2), this yields

$$\{X_q^\epsilon \Rightarrow [\neg X_{\hat{q}}^\epsilon \wedge \neg X_{q_f}^\epsilon]\} \wedge \{X_{\hat{q}}^\epsilon \Rightarrow [\neg X_q^\epsilon \wedge \neg X_{q_f}^\epsilon]\} \wedge \cdots \wedge \{X_{q_f}^{22} \Rightarrow [\neg X_q^{22} \wedge \neg X_{\hat{q}}^{22}]\}.$$

It should be clear at this point that $\varpi(\Omega_{\twoheadrightarrow}) = \mathcal{P}(t) \twoheadrightarrow Q$: we have encoded the set of partial functions. One would expect the next move to be encoding totality, as per (**1(a)iii**), and we could indeed do so easily with

$$\bigwedge_{\alpha\in\mathcal{P}(t)} \bigvee_{q\in Q} X_q^\alpha, \tag{7.6}$$

however this condition would actually become redundant with our encoding of (**1b**), as we will see, so we move on directly to it. The object is to enforce the compatibility of ρ with the transition rules of $\mathcal{A}$. Let us then translate the fact that a given transition rule applies at some position α by the *rule application constraint* $\Psi^\alpha(\cdot)$, which takes a rule $r \in \Delta$ as its argument: for any $\alpha \in \mathcal{P}(t)$, and any transition rule $f(q_1,\ldots,q_n) \to q \in \Delta$, we let

$\Psi^\alpha(\cdot)$: *rule application constraint*

$$\Psi^\alpha\big(f(p_1,\ldots,p_n) \to q\big) = X_q^\alpha \wedge \bigwedge_{k=1}^{n} X_{p_k}^{\alpha.k}.$$

This is fairly straightforward: we are stating that the rule $f(p_1,\ldots,p_n) \to q \in \Delta$ applies at position α. By (2.3), this amounts to the statement:

$$\rho(\alpha) = q \ \wedge\ \rho(\alpha.1) = p_1 \wedge \ \ldots\ \wedge \rho(\alpha.n) = p_n\ .$$

Now, in order to express the notion of compatibility with the transition rules, and to finally encode a run of the underlying BUTA $\mathcal{A}'$, there remains to assert that, at each position in the term, a transition rule applies. Only those rules with the right symbol can apply at any given position, so let us define $\Delta_\sigma \subseteq \Delta$ as the subset of rules which are rooted in the symbol $\sigma \in \mathbb{A}$: $\Delta_\sigma = \{\, \sigma(\ldots) \to \cdots \in \Delta \,\}$. With this, we can write the *rules compatibility constraint* Ω_Δ:

Δ_σ: *transitions rooted in* σ.

Ω_Δ: *rules compatibility constraint*

$$\Omega_\Delta = \bigwedge_{\alpha\in\mathcal{P}(t)} \left[\bigvee_{r\in\Delta_{t(\alpha)}} \Psi^\alpha(r) \right].$$

When instantiated on our small running example, this yields

$$([X_{q_f}^\epsilon \wedge X_{\hat{q}}^1 \wedge X_{\hat{q}}^2] \vee [X_q^\epsilon \wedge X_q^1 \wedge X_q^2] \vee [X_{\hat{q}}^\epsilon \wedge X_q^1 \wedge X_q^2]) \wedge \cdots \wedge (X_q^{22} \vee X_{\hat{q}}^{22}).$$

Note that Ω_Δ subsumes (7.6), and thus takes care of the totality condition (**1(a)iii**) on top of (**1b**), since at every position $\alpha \in \mathcal{P}(t)$, we must be in some state q resulting from the application of some transition rule, by the X_q^α component of $\Psi^\alpha(r)$. Recall that this clause is meant to be added conjunctively to what we already have: if both $\Omega_{\twoheadrightarrow}$ and Ω_Δ are satisfied simultaneously, then at most one rule applies at each position, so that in the end, exactly one rule applies. Thus we have so far

$$\varpi(\Omega_{\twoheadrightarrow} \wedge \Omega_\Delta) = \left\{\, \rho \;\middle|\; \rho \text{ is a run of } \mathcal{A}' \text{ on } t \,\right\}.$$

The last thing which is required to encode an accepting run for the underlying tree automaton $\mathcal{A}'$ is that the run must end up in a final state at the root of the term, satisfying condition (**1c**). This is directly translated into $\bigvee_{q\in F} X^{\varepsilon}_q$, and thus we have

$$\varpi\Big(\Omega_{\twoheadrightarrow} \wedge \Omega_{\Delta} \wedge \bigvee_{q\in F} X^{\varepsilon}_q\Big) = \big\{\, \rho \;\big|\; \rho \text{ is an accepting run of } \mathcal{A}' \text{ on } t \,\big\}\,.$$

Now we must add further restrictions to ensure compatibility with the global equality and disequality constraints, following conditions (**2** and **3**). The variables which we have already defined are not sufficient to translate statements of the form "such structural subtree does (or does not) evaluate to such state". The keyword here is *structural*, that is to say, considering only the tree $t|_\alpha$ itself, and forgetting the position α. Note that access to positions is not enough to discuss equality of subterms, as $t|_\alpha = t|_\beta \not\Rightarrow \alpha = \beta$, and a same subtree u may evaluate to different states, in different positions. Therefore we need to introduce new variables to link states and structural subterms by a relation. Let us use T^u_q to denote "the subterm u evaluates to q", for any $u \trianglelefteq t$ and $q \in Q$: intuitively

$$T^u_q \equiv \text{“}u \in \mathcal{L}^q(\mathcal{A})\text{”}\,.$$

Semantics & Notations

Although α appears in the *notation* $t|_\alpha$, the information α is not included in the *object* $t|_\alpha$. Refer to the definition of subtrees in section 2.2[p24]. This has to do with our convention that a subtree is a normal tree: its root is ε, and not α. Other authors sometimes take the second convention – which is occasionally convenient.

T^u_q: *variable "$u \in \mathcal{L}^q(\mathcal{A})$"*

Of course, in order for that meaning to hold we need to "glue" these new variables to the old ones: if we are in a certain state q at a position α, then it follows that the subterm $t|_\alpha$ evaluates to q: this is straightforwardly translated into the *structural glue* formula $\Omega_{\leftrightarrows}$:

$\Omega_{\leftrightarrows}$: *structural glue*

$$\Omega_{\leftrightarrows} = \bigwedge_{\substack{\alpha\in\mathcal{P}(t)\\ q\in Q}} \Big[X^{\alpha}_q \implies T^{t|\alpha}_q\Big]\,.$$

On our running example, this yields

$$\{X^{\epsilon}_q \Rightarrow T^2_q\} \wedge \{X^{\epsilon}_{\hat{q}} \Rightarrow T^2_{\hat{q}}\} \wedge \{X^{\epsilon}_{q_f} \Rightarrow T^2_{q_f}\} \wedge \cdots \wedge \{X^{22}_{q_f} \Rightarrow T^0_{q_f}\}\,,$$

where the superscript 2 of T^2_q designates the subtree $f\big(f(a,a), f(a,a)\big)$, 0 designates a, and so forth, as given in the annotation (7.2) of t and the mapping (7.3). Note that so far, this formula does not influence satisfiability at all, and we have

$$\varpi\Big(\Omega_{\twoheadrightarrow} \wedge \Omega_{\Delta} \wedge \bigvee_{q\in F} X^{\varepsilon}_q \wedge \Omega_{\leftrightarrows}\Big) = \varpi\Big(\Omega_{\twoheadrightarrow} \wedge \Omega_{\Delta} \wedge \bigvee_{q\in F} X^{\varepsilon}_q\Big)\,.$$

This will of course change so soon as negated versions of the T^u_q variables are added into the mix. Now that the different kinds of variables are linked, we can move on and encode the equality constraint, as per condition (**2**). To do so, let us rephrase statement (2.6)[p35] a bit; the idea is the same as in (7.5), that is, to transform a positive statement into one expressible in terms of negative variables. The following is equivalent to (2.6):

$$\forall \alpha \in \mathcal{P}(t),\ \rho(\alpha) = q \wedge p \cong q \implies \forall u \trianglelefteq t, u \neq t|_\alpha,\ u \notin \mathcal{L}^p(\mathcal{A})\,.$$

Intuitively, supposing that $\rho(\alpha) = q$, for the run to be compatible with the equality constraint, it must be such that no subterm different from $t|_\alpha$ may evaluate to p, where $p \cong q$. This reformulation translates straightforwardly into the constraint of *compatibility with* $\cong$, $\Omega_{\cong}$:

$\Omega_{\cong}$: *compatibility with* $\cong$

$$\Omega_{\cong} = \bigwedge_{\substack{\alpha\in\mathcal{P}(t)\\ q\in Q}} \left[X^{\alpha}_q \implies \bigwedge_{\substack{p\in Q\\ p\cong q}} \bigwedge_{\substack{u\trianglelefteq t\\ u\neq t|_\alpha}} \neg T^u_p\right]\,.$$

On our running example, we obtain the formula

$$\{X^{\epsilon}_{\hat{q}} \Rightarrow [\neg T^{1}_{\hat{q}} \wedge \neg T^{0}_{\hat{q}}]\} \wedge \{X^{11}_{\hat{q}} \Rightarrow [\neg T^{2}_{\hat{q}} \wedge \neg T^{1}_{\hat{q}}]\} \wedge \cdots \wedge \{X^{22}_{\hat{q}} \Rightarrow [\neg T^{2}_{\hat{q}} \wedge \neg T^{1}_{\hat{q}}]\}\,.$$

There remains to encode the compatibility with the disequality constraint. Let us deal with the case where either $\not\approx$ is assumed to be irreflexive – as in [Filiot et al., 2008] and (2.7)$_{[p36]}$ – or the states involved are different. Suppose that we are at position α, and that $\rho(\alpha) = q$; then we cannot have any subterm identical to $t|_\alpha$ evaluate to any p, when $p \not\approx q$. The translation of (2.7) is therefore immediate, and we have the *compatibility with* $\not\approx$ *(for* $p \neq q$*)* formula $\Omega^{\neq}_{\not\approx}$:

$\Omega^{\neq}_{\not\approx}$: *compatibility with irr.* $\not\approx$

$$\Omega^{\neq}_{\not\approx} = \bigwedge_{\substack{\alpha\in\mathcal{P}(t)\\ q\in Q}} \left[X^{\alpha}_{q} \implies \bigwedge_{\substack{p\in Q\\ p\not\approx q\\ p\neq q}} \neg T^{t|\alpha}_{p} \right].$$

On our running example, this yields

$$\{X^{\epsilon}_{\hat{q}} \Rightarrow \neg T^{2}_{q_f}\} \wedge \{X^{\epsilon}_{q_f} \Rightarrow \neg T^{2}_{\hat{q}}\} \wedge \cdots \wedge \{X^{22}_{q_f} \Rightarrow \neg T^{0}_{\hat{q}}\}\,.$$

However, the current definition (2.8) of $\not\approx$ does not assume irreflexivity [Filiot et al., 2010], an aspect which, as has already been pointed out at the end of section 5.2.1$_{[p112]}$, is known to increase expressive power. With the current definition, one is able to write statements such as $q \not\approx q$, with the meaning that no two *distinct* subtrees which evaluate to q may be structurally identical. Here we hit a little snag, since this distinction is made with respect to the positions in which the subtrees are rooted. This is obviously not respected by $\Omega^{\neq}_{\not\approx}$, because, if the irreflexivity condition is removed, the formula will not and cannot differentiate between two distinct subterms and the same subterm, taken twice. To clarify that, suppose that $\rho(\alpha) = q$. Without the condition $p \neq q$ in $\Omega^{\neq}_{\not\approx}$, $\neg T^{t|\alpha}_{q}$, and yet, by $\Omega_{\leftrightarrows}$ we have $T^{t|\alpha}_{q}$, yielding an immediate contradiction – which mirrors the behaviour of (2.7).

This is why the case where $q \not\approx q$ must be dealt with separately. The comparison of positions which appears in (2.8) cannot be encoded yet, as we do not have any means of linking subterms with positions. A new kind of variables is therefore required, which we take of the form S^{α}_{u}, encoding the intuitive statement "the subterm u is rooted in α". The above property is then encoded using this variable, in the *compatibility with* $\not\approx$ *(for* $q \not\approx q$*)* formula $\Omega^{=}_{\not\approx}$:

S^{α}_{u}: *variable "*u *rooted in* α*"*

$\Omega^{\neq}_{\not\approx}$: *compatibility with refl.* $\not\approx$

$$\Omega^{=}_{\not\approx} = \bigwedge_{\alpha\in\mathcal{P}(t)} S^{\alpha}_{t|_\alpha} \wedge \bigwedge_{\substack{\alpha\neq\beta\in\mathcal{P}(t)\\ q\not\approx q}} \left[X^{\alpha}_{q} \wedge X^{\beta}_{q} \implies \neg S^{\alpha}_{t|_\beta} \right].$$

It should be noted that $\Omega^{=}_{\not\approx}$ deals exclusively with constraints of the form $q \not\approx q$, and is therefore only useful as a conjunct of $\Omega^{\neq}_{\not\approx}$. We can now state our main result, and define the overall encoding formula $\Theta_{\mathcal{A}}(t)$:

$\Theta_{\mathcal{A}}(t)$: *membership formula*

$$\Theta_{\mathcal{A}}(t) = \Omega_{\twoheadrightarrow} \wedge \Omega_{\Delta} \wedge \bigvee_{q\in F} X^{\epsilon}_{q} \wedge \Omega_{\leftrightarrows} \wedge \Omega_{\cong} \wedge \Omega^{\neq}_{\not\approx} \wedge \Omega^{=}_{\not\approx}\,,$$

such that, by all the above, we have

$$\varpi(\Theta_{\mathcal{A}}(t)) = \{\, \rho \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ on } t \,\}\,.$$

Equivalently, this can be stated as:

Theorem 7.1: $\mathrm{TA}^{=}_{\neq}$ membership, correctness and soundness

There exists a successful run ρ of the $\mathrm{TA}^{=}_{\neq}$ $\mathcal{A}$ on a term t if and only if $\Theta_{\mathcal{A}}(t)$ is satisfiable. Moreover, if $\nu \models \Theta_{\mathcal{A}}(t)$, then for any $\alpha \in \mathcal{P}(t)$ we have $\rho(\alpha) = q \iff \nu \models X^{\alpha}_{q}$.

The above encoding has been simplified, implemented and tested. This is the subject matter of the next sections.

7.2 Complexity and Optimisations

The encoding proposed above is straightforward, but in the interest of keeping the size of the formula to a minimum, we quickly go over some ways in which it can be lightened through some relatively simple observations.

Although the encoding is sizeable, it remains polynomial in the size of our input automaton $\mathcal{A}$ and the term t: the size of $\Theta_{\mathcal{A}}(t)$ – as number of literals – is visibly $O(|t|^2|Q|^2)$. In practice however, this can often be pared down considerably. Let ρ be a successful run of the underlying tree automaton $\mathcal{A}$ on t, and consider for instance the structural glue:

$$\Omega_{\leftrightarrows} = \bigwedge_{\substack{\alpha \in \mathcal{P}(t) \\ q \in Q}} \left[X^{\alpha}_{q} \implies T^{t|\alpha}_{q} \right].$$

The formula considers all possible couples (α, q), but in general this is unnecessary because not all states are obtainable at any given position. In order to ever have X^{α}_{q}, that is to say, $\rho(\alpha) = q$, there must be some transition rule of the form $t(\alpha)(\dots) \to q$ in Δ, at least. Thus we let $\delta(\alpha)$ be the set of *possibly obtainable states at position* α:

$$\delta(\alpha) = \{\, q \in Q \mid \exists t(\alpha)(\dots) \to q \in \Delta \,\},$$

and, given a position α, we only need to deal with $q \in \delta(\alpha)$. Another observation which can be made a priori is that the only occurrences of negations of the form $\neg T^{u}_{q}$ appear in $\Omega_{\cong}$ and $\Omega^{\neq}_{\not\cong}$, and then only when q is in the domain of either $\not\cong$ or $\cong$. It follows that literals of the form T^{u}_{q} can only alter the satisfiability of $\Theta_{\mathcal{A}}(t)$ when q is in $\mathrm{dom}(\not\cong) \cup \mathrm{dom}(\cong)$. Thus, writing

$$\delta'(\alpha) = \delta(\alpha) \cap (\mathrm{dom}(\not\cong) \cup \mathrm{dom}(\cong)),$$

we can reduce the formula to

$$\Omega_{\leftrightarrows} = \bigwedge_{\substack{\alpha \in \mathcal{P}(t) \\ q \in \delta'(\alpha)}} \left[X^{\alpha}_{q} \implies T^{t|\alpha}_{q} \right].$$

Similar observations can be made for $\Omega^{\neq}_{\not\cong}$, $\Omega^{=}_{\not\cong}$ and $\Omega_{\cong}$. Staying with variables of the form T^{u}_{q}, looking at $\Omega_{\cong}$, one can argue that in the subformula

$$\bigwedge_{\substack{u \trianglelefteq t \\ u \neq t|\alpha}} \neg T^{u}_{p},$$

it is unnecessary to write $\neg T_p^u$ when we know that the subtree u cannot possibly evaluate to the state p. This is clearly the case if the root symbol $u(\varepsilon)$ is not used in any transition rule leading to p. Thus we let

$$\tau(q) = \{\, \sigma \in \mathbb{A} \mid \exists \sigma(\dots) \to q \in \Delta \,\}$$

be the set of symbols which a subterm may be rooted in, given that it evaluates to the state q, and we lighten the above-mentioned subformula, yielding

$$\Omega_{\cong} = \bigwedge_{\substack{\alpha \in \mathcal{P}(t) \\ q \in Q}} \left[X_q^\alpha \implies \bigwedge_{\substack{p \in Q \\ p \cong q}} \bigwedge_{\substack{u \trianglelefteq t, u(\varepsilon) \in \tau(p) \\ u \neq t|_\alpha}} \neg T_p^u \right].$$

Lastly, in the compatibility formula $\Omega_{\not\cong}^{=}$, it is clear that the variables $S_{t|_\alpha}^\alpha$ serve no purpose whatsoever when the subtree in α cannot evaluate to a state q such that $q \not\cong q$. Thus we let

$$\mu(q) = \{\, \alpha \in \mathcal{P}(t) \mid\ t(\alpha) \in \tau(q) \,\}$$

be the set of positions at which the subtree may evaluate to the state q, and reduce the first part of $\Omega_{\not\cong}^{=}$ to

$$\bigwedge_{\alpha \in \bigcup_{q \not\cong q} \mu(q)} S_{t|_\alpha}^\alpha .$$

In its second part, we arbitrarily order positions and regroup couples of implications with the same premises, and thus the condition becomes:

$$\Omega_{\not\cong}^{=} = \bigwedge_{\alpha \in \bigcup_{q \not\cong q} \mu(q)} S_{t|_\alpha}^\alpha \wedge \bigwedge_{\substack{\alpha < \beta \in \mu(q) \\ q \not\cong q}} \left[X_q^\alpha \wedge X_q^\beta \implies \neg S_{t|_\beta}^\alpha \wedge \neg S_{t|_\alpha}^\beta \right].$$

Note that reducing $\Omega_{\twoheadrightarrow}$ is much more problematic, but it is possible to simply do away with this part of the formula altogether if one replaces $\bigvee_{q \in F} X_q^\varepsilon$ by $\bigwedge_{q \notin F} \neg X_q^\varepsilon$, provided that the term is accepted by the underlying tree automaton. This can be checked separately by other, less expensive means, since the membership problem for tree automata is polynomial. Of course in that case the second result of Theorem 7.1 does not hold anymore.

While computationally inexpensive, these simplifications can yield significant savings on TAGED with low density and where few states are involved in the global constraints, which are fairly reasonable assumptions in the context of XML documents processing. Note that one could find more drastic simplifications by examining the tree automaton more closely; for instance one could remove, at each position, any state which cannot appear in a successful run. Simplifications of this kind would certainly yield better results on sizeable and complex TAGED, but it is not certain that the overhead of implementing and computing them would be compensated by the SAT-solving performance gains.

7.3 Implementation and Experiments

conjunctive normal form

In the remainder of this chapter, we shall often be referring to the *conjunctive normal*

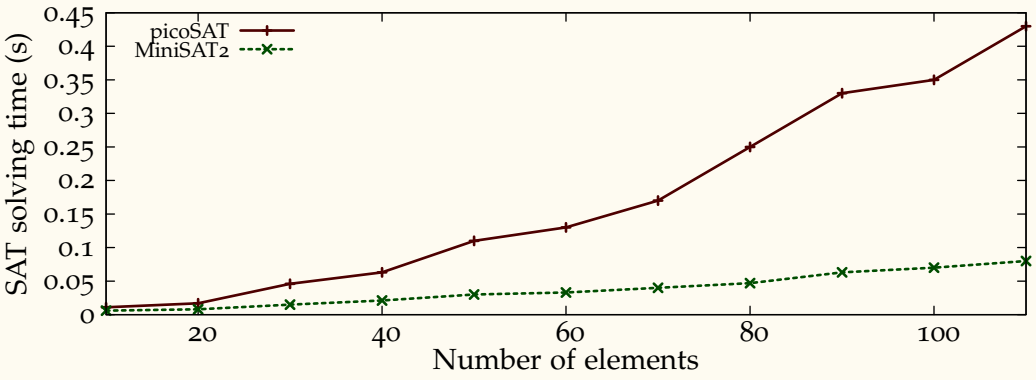


Figure 7.1: CNF solving time, laboratory example.

form (*CNF*) of propositional formulæ, which has yet to be defined. A formula is in CNF if it is a conjunction of disjunction of literals, and any formula can be put in CNF through various methods, generally by applications of De Morgan's laws to push the negations inside the formula and switch between $\wedge$ and $\vee$, as well as the other usual equivalences to get rid of implications and other unwanted operators. This is a fairly important form, especially in this context, as SAT solvers generally require their inputs to be in CNF.

CNF

One of our test cases refers to the example of an XML database modelling a laboratory in a university, its teams, and its members. We do not give the detail of this test case, as it is extremely similar to our "starship" running example; cf. section 1.3[p16].

7.3.1 Experimental Results

For the tests, we implemented the static simplifications described in section 7.2, which divided the size of the generated formula by 36 in the case of the laboratory example automaton. The testing tool which we developed, implemented in the OCaml programming language, takes as input a TAGED expressed in a syntax close to that of Timbuk [Feuillade et al., 2004] and a term, from which it generates the corresponding formula $\Theta_{\mathcal{A}}(t)$. However, most modern SAT solvers take input in the DIMACS CNF format, and a naive conversion to Conjunctive Normal Form (using De Morgan's laws, distributivity and removal of double negations) could lead to an explosion of the size of the formula.

DIMACS CNF

Example input for $(X \vee \neg Z) \wedge (Y \vee Z \vee \neg X)$:

```
c DIMACS CNF for φ
p cnf 3 2
1 -3 0
2 3 -1 0
```

In order to avoid running into this problem we used an existing tool to handle linear-size conversion to CNF and generation of DIMACS CNF files: the *bit-level analysis tool* (*BAT*), version 0.2 [Manolios, Srinivasan & Vroon, 2007], which is a prototype implementation of an efficient CNF conversion algorithm [Chambers, Manolios & Vroon, 2009]. Experiments were run on an 2.53 GHz Intel Core2 Duo machine with 2Gb of RAM under the Linux kernel. It should be noted that this was done in late 2010, and that the SAT solvers may have evolved – presumably and hopefully improved – since that time. The BAT is still in version 0.2 at the time of writing, though.

bit-level analysis tool

BAT

Figure 7.1 shows the respective running times of the two SAT solvers PicoSAT and MiniSAT2 on an implementation of our laboratory example. Accepted trees of

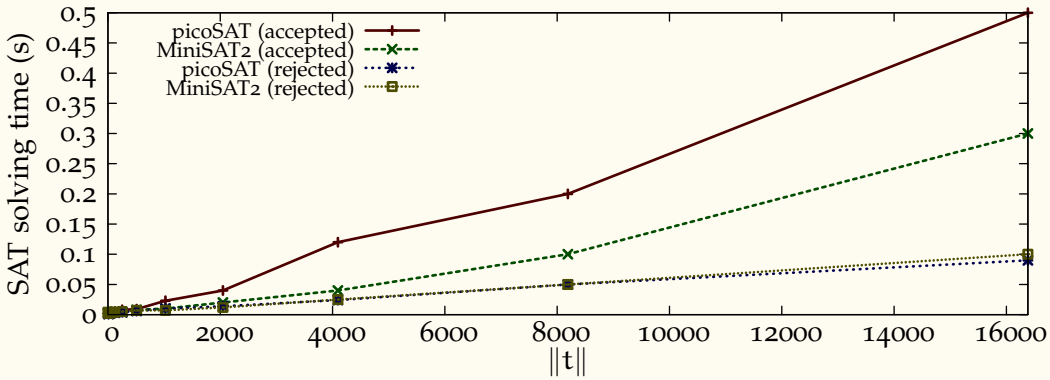


Figure 7.2: CNF solving time, $L_{=}$, for accepted and rejected terms.

varying sizes have been generated with random member names of random length. In the figure, the size of the generated trees is given in terms of the number of teams in the university; the size in terms of the number of nodes is proportional to these data. The test shows that while both solvers perform very well on this query, MiniSAT2 tends to outperform PicoSAT as the terms grow, which suggests that the heuristic used for SAT solving may significantly affect the overall efficiency of our queries.

Figure 7.2 shows the same experiment, this time with the small TAGED accepting $L_{=}$ given in $(7.1)_{[p130]}$, and for both accepted and rejected terms. The size of the terms designates the number of nodes of the tree, as usual. Both solvers display similar performances for this experiment, with MiniSAT2 being about twice as fast as PicoSAT on accepted terms. On rejected terms however both solvers show roughly the same performances, and take less time than on accepted terms, by a factor of 3 (PicoSAT) and 5 (MiniSAT2) on large terms.

It would have been interesting to increase the size of our terms until both solvers timed out, but we were unfortunately limited by the software we used. Our own tool is not optimised for speed, and CNF conversion with BAT took about 4.5 times as much time as formula generation. Moreover, BAT fails with a stack overflow when the input formula becomes too large. Despite these practical setbacks, the results remain fairly encouraging, as the current bottleneck lies on the least computationally expensive parts of the process: both the generation of the formula and the conversion to CNF are quadratic in the worst case. On the other hand, SAT solving proves quite efficient, even on fairly large formulæ: the order of magnitude of the largest tested formulæ is of approximately 70 000 variables, 120 000 clauses and 250 000 literals (in CNF), for a solving time well under one second.

7.3.2 The Tool: Inputs and Outputs

The testing prototype has been implemented in the OCaml programming language. SAT solving is the hard part of the process, not formula generation and conversion, which are both polynomial, or, more precisely, quadratic in the worst case. Experimentally, with our static optimisations, the formula can grow linearly, as we have observed in the case of $L_{=}$, for instance. For this reason, we focused on SAT solving time in our experiments; including our unoptimised tool in the benchmarks would not be pertinent.

```
(* TAGED for L=, cf. (7.1)[p130] *)
Taged fxxA Alphabet f a b
States q qq qf Final qf
Rules f qq qq : qf
 f q q : q  f q q : qq
 a:q  a:qq  b:q  b:qq
Equal qq qq Different qq qf
```

```
f(f(a,a), f(a,a))
// input term t
// output: cf. Fig. 7.4[p140]
```

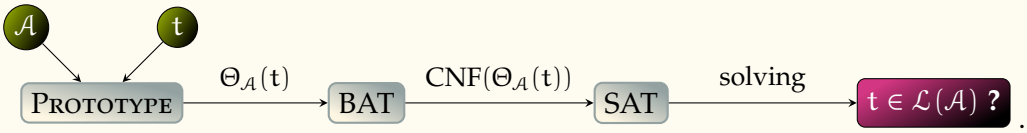
Figure 7.3: Input syntax of the membership tool: automaton and term.

The tool takes as input a TAGED in a syntax close to that of Timbuk [Feuillade et al., 2004] and a term, and generates the corresponding formula in the BAT's input format. Example inputs of our tools are given in Figure 7.3.

The default mode of operation is of course to generate BAT input and to run BAT and a SAT solver immediately. But there is also the possibility of outputting a LaTeX 2ε file detailing, in a user-friendly format, the input automaton, the annotated input term, and the corresponding generated formula. Figure 7.4[p140] gives the LaTeX output corresponding to the input of Fig. 7.3 – although the disequality constraints are disabled in that example. Note that the tool indexes both subterms and positions, in an effort to keep the formula somewhat readable even on large trees. This is the same convention as that taken for the annotations (7.1) of our running example.

7.4 Conclusions

Though experiments were limited by extraneous factors – namely CNF generation – the results are encouraging. Indeed, despite the respectable size of the generated formulæ, SAT solving remains surprisingly fast. Indeed, CNF generation – an easy, polynomial task in theory – was the bottleneck of our experiments. To recapitulate the process schematically:

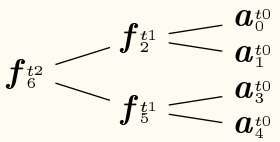


For a practical implementation, it would probably be best to renounce the dependency upon an external tool for conversion into CNF, especially as the tools are not mature. It would certainly be much better to generate the CNF formula on-the-fly, while interfacing with a SAT solver. The detection of conflicting clauses could then be done in parallel to the generation itself. With luck, terms may be rejected before the generation of the formula has even ended.

Also note that the formula $\Theta_{\mathcal{A}}(t)$, such as we have defined it, is already mostly in CNF, the exception being the subformula Ω_{Δ}. If this could be recoded directly in – reasonably sized – CNF, this would obviate the need for any supplementary conversion step.

Automaton: **fxxA**
Alphabet: $\{\boldsymbol{f}, \boldsymbol{a}, \boldsymbol{b}\}$
States: $\{\mathsf{q}, \mathsf{qq}, \mathsf{qf}\}$
Final States: $\{\mathsf{qf}\}$
Transition Rules: {
$\boldsymbol{f}(\mathsf{qq}, \mathsf{qq}) \to \mathsf{qf}$,
$\boldsymbol{f}(\mathsf{q}, \mathsf{q}) \to \mathsf{q}$,
$\boldsymbol{f}(\mathsf{q}, \mathsf{q}) \to \mathsf{qq}$,
$\boldsymbol{a} \to \mathsf{q}$,
$\boldsymbol{a} \to \mathsf{qq}$,
$\boldsymbol{b} \to \mathsf{q}$,
$\boldsymbol{b} \to \mathsf{qq}$
}
Global State Equality: $\{\mathsf{qq} = \mathsf{qq}\}$
Global State Disequality: $\{\mathsf{qq} \neq \mathsf{qf}\}$
End Automaton.

Term as expression: $\boldsymbol{f}\{\boldsymbol{f}[\boldsymbol{a}, \boldsymbol{a}], \boldsymbol{f}[\boldsymbol{a}, \boldsymbol{a}]\}$
Term as tree:



Membership formula $= [[(X^3_{\mathsf{q}} \vee X^3_{\mathsf{qq}}) \wedge ([X^5_{\mathsf{qf}} \wedge X^3_{\mathsf{qq}} \wedge X^4_{\mathsf{qq}}] \vee [X^5_{\mathsf{q}} \wedge X^3_{\mathsf{q}} \wedge X^4_{\mathsf{q}}] \vee [X^5_{\mathsf{qq}} \wedge X^3_{\mathsf{q}} \wedge X^4_{\mathsf{q}}]) \wedge ([X^2_{\mathsf{qf}} \wedge X^0_{\mathsf{qq}} \wedge X^1_{\mathsf{qq}}] \vee [X^2_{\mathsf{q}} \wedge X^0_{\mathsf{q}} \wedge X^1_{\mathsf{q}}] \vee [X^2_{\mathsf{qq}} \wedge X^0_{\mathsf{q}} \wedge X^1_{\mathsf{q}}]) \wedge (X^1_{\mathsf{q}} \vee X^1_{\mathsf{qq}}) \wedge ([X^6_{\mathsf{qf}} \wedge X^2_{\mathsf{qq}} \wedge X^5_{\mathsf{qq}}] \vee [X^6_{\mathsf{q}} \wedge X^2_{\mathsf{q}} \wedge X^5_{\mathsf{q}}] \vee [X^6_{\mathsf{qq}} \wedge X^2_{\mathsf{q}} \wedge X^5_{\mathsf{q}}]) \wedge (X^0_{\mathsf{q}} \vee X^0_{\mathsf{qq}}) \wedge (X^4_{\mathsf{q}} \vee X^4_{\mathsf{qq}})] \wedge [\neg X^6_{\mathsf{q}} \wedge \neg X^6_{\mathsf{qq}}] \wedge [\{X^3_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_0\} \wedge \{X^5_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_1\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_1\} \wedge \{X^1_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_0\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_2\} \wedge \{X^0_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_0\} \wedge \{X^4_{\mathsf{qq}} \Rightarrow T^{\mathsf{qq}}_0\}] \wedge [\{X^5_{\mathsf{qq}} \Rightarrow \neg T^{\mathsf{qf}}_1\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow \neg T^{\mathsf{qf}}_1\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow \neg T^{\mathsf{qf}}_2\}] \wedge [\{X^3_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_1]\} \wedge \{X^5_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_0]\} \wedge \{X^2_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_0]\} \wedge \{X^1_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_1]\} \wedge \{X^6_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_1 \wedge \neg T^{\mathsf{qq}}_0]\} \wedge \{X^0_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_1]\} \wedge \{X^4_{\mathsf{qq}} \Rightarrow [\neg T^{\mathsf{qq}}_2 \wedge \neg T^{\mathsf{qq}}_1]\}]]$

1

Figure 7.4: Example LaTeX output of the tool – cf. Fig. 7.3[p139].

— Part IV —

Decision Problems for Tree-Walking Automata

Chapter 8
Tree Automata for XML

Contents

—Where we binarise trees, walk all over them, and then throw pebbles at them.

Up until now, this thesis has concerned itself almost exclusively with varieties of automata extending the standard bottom-up model, within domains of inquiry chiefly tied to model-checking of programs. This chapter and the next depart from that focus, as their objects include the more stateful classes of automata, such as tree-walking automata (TWA), which are defined in the first section, and the motivations have more to do with verification and queries in the context of semi-structured documents and databases than with programs and circuits – an aspect already mentioned in section 1.3[p16], which we shall canvass in more detailed fashion in the present introductory chapter.

A point which must also be made clear before getting to our contributions is that of the relevance of the ranked-tree model to that kind of questions; a model which we have, to that point, assumed without justification – and none was required until now. However, semi-structured documents are more readily seen as unranked trees. To clarify that, the relationship between ranked and unranked trees is discussed in the first section. Although this chapter draws on multiple references, [Comon et al., 2008, Chap. 8] provides – yet again – a very comprehensive and detailed survey of those topics.

The section on queries is also indebted to the thesis [Filiot, 2008], which focuses on queries and XML, and provides exhaustive surveys in those domains. The sections on tree-walking automata, caterpillar expressions and their variants draw on [Bojańczyk, 2008] and [Hosoya, 2010] as well, the latter also being a good general survey of some other topics presented here.

The definition of TWA in Sec. 8.1[p144] is required reading. However, while the remainder of this chapter serves to put our contributions into due context, its contents are not strictly required to understand them. The reader anxious to read our own results may therefore, after reading Sec. 8.1, skim or skip this material and proceed forthwith to the next chapter, page 165.

8.1 Tree-Walking Automata

Both the top-down and the bottom-up strains are sometimes referred to as branching tree automata, because the way in which they operate can be thought of as having a moving head on each branch of the tree. A top-down automaton branches out, as a head separates towards each child of the current node, while a bottom-up automaton branches in, the heads on the children fusing onto the father. This is in contrast to finite-state automata, which can be seen as a single head, moving left-to-right on a word. Less restrictively, the head can actually be allowed to move right-to-left as well, choosing which depending on its current state, the symbol being read, and whether its current position is at the beginning, the middle, or the end of the word. The purpose of this last datum is obviously to prevent the head from inadvertently falling off the word. The class of FSA with a bidirectional head is called *two-way automata* (*2FSA*) and, somewhat surprisingly, has exactly the same expressive power as baseline, unidirectional FSA. Bidirectionality is nevertheless occasionally convenient, as some languages can be represented much more succinctly with it than without – up to an exponential decrease in the number of states.

two-way automata
2FSA

We have seen that the generalisation from FSA to BUTA is natural when looking at the transitions rules: from $\sigma(p) \to q$ to $\sigma(p_1, \ldots, p_n) \to q$. But thinking of a FSA as a moving head on a word, it is also natural to imagine a tree automaton as being a head moving on a tree. This seems of dubious usefulness if the head can only move in one way – it is obviously impossible to visit a tree without ever doubling back – but by analogy with two-way automata, it may be allowed to move from father to children and vice-versa. As in the case of words, there is a need for some additional positional information to prevent the head from jumping off the tree. This is not a problem at the leaves, because the information that there are no children to move down to is encoded into the arity of the symbols at the leaves: they are constants. On the other hand, the root of a tree has no such guardrail, therefore the head must know whether it is at the root, lest it tries to move up – although in Part IV[p143], we will *make* automata jump off the root of trees as a matter of course. There are many variants on the matter of accepting conditions; one can choose to have accepting or rejecting transitions or commands, or to switch to a final state. In this thesis we shall use final states. While this is enough to define a working class of tree automata, it is not a terribly powerful class, as the head gets lost in the tree very quickly [Kamimura & Slutzki, 1981; Bojańczyk, 2008]. To mitigate this effect, it is reasonable that, besides knowing whether it is at the root, the head should know whether it is on a first child, a second child, etcetera.

The combination of those ideas defines the class of *tree-walking automata* (*TWA*).

tree-walking automata
TWA

For simplicity of exposition, in this document they will be defined only on binary trees, which is not a fundamental restriction, as trees and automata can be *binarised*; this aspect is discussed at some length in section 8.2.3[p152]. A *binary alphabet* is a ranked alphabet $\mathbb{A}$ such that $\mathbb{A} = \mathbb{A}_0 \cup \mathbb{A}_2$, and binary trees are trees formed over a binary alphabet. In this context, we shall use positions defined over $\{0,1\}^*$, whereas in other contexts they are taken over $\{1,2,\dots\}^*$; it seems to be a common choice in the literature on TWA, which this document mirrors. To summarise the intuitive ideas developed above, a tree-walking automaton can be thought of as a head moving in the tree from father to son and from son to father. The head chooses its next move based on its internal state, the symbol at its current position, and whether its current position is the root of the tree, a left son, or a right son. A TWA accepts a tree if, starting from the root in an initial state, its head can move back to the root in a final state.

binarisation

binary alphabet

▽ Definition 8.1: Tree-Walking Automata

A tree-walking automaton $\mathcal{A}$ is a tuple $\langle \mathbb{A}, Q, I, F, \Delta \rangle$, where

- $\mathbb{A}$ is a binary alphabet,
- Q is a finite set of *states*,
- I is the set of *initial states*,
- F is the set of *final states*,
- Δ is the set of *transitions*.

As usual, states are fresh nullary symbols, and initial as well as final states are found in Q; in short, $Q \cap \mathbb{A} = \varnothing$ and $I, F \subseteq Q$. The transitions use special symbols $\mathbb{T}$ and $\mathbb{M}$ to denote styles of positions in the tree and the direction of movement, respectively:

$$\Delta \subseteq \mathbb{A} \times Q \times \underbrace{\{\star, \mathbf{0}, \mathbf{1}\}}_{\mathbb{T}\,:\,\text{types}} \times \underbrace{\{\uparrow, \circlearrowleft, \swarrow, \searrow\}}_{\mathbb{M}\,:\,\text{moves}} \times Q\,.$$

$\mathbb{T}$: *TWA node types*

$\mathbb{M}$: *TWA moves*

In order to formally explain how a tree-walking automaton activates its transitions, let us introduce a few necessary notions and notations. Each node α of a tree t has a *type* in $\mathbb{T}$, denoted by $\mathrm{ty}\,\alpha$, such that $\mathrm{ty}\,\varepsilon = \star$ (root), $\mathrm{ty}\,(\beta.0) = \mathbf{0}$ (left son), $\mathrm{ty}\,(\beta.1) = \mathbf{1}$ (right son). In practice, it is useful to have a special notation $\mathbb{S}$ for the *sons*: $\mathbb{S} = \{\mathbf{0}, \mathbf{1}\} \subset \mathbb{T}$. We shall also put in relation types and moves through the function $\chi(\cdot) : \mathbb{S} \to \{\swarrow, \searrow\}$ such that $\chi(\mathbf{0}) = \swarrow$ and $\chi(\mathbf{1}) = \searrow$. It is convenient to take the special notation $\langle f, p, \tau \to \mu, q \rangle$ for the tuple $(f, p, \tau, \mu, q) \in \Delta$. With this notation, some of the parameters can be replaced by sets, with the obvious meaning that we consider the set of all transitions thus described. For instance

ty α: *TWA type of* α

$\mathbb{S}$: *TWA "son" types*

$\chi(\cdot)$: *TWA get move from type*

$$\langle \mathbb{A}_2, p, \mathbb{T} \to \circlearrowleft, q \rangle = \{\, (\sigma, p, \tau, \circlearrowleft, q) \mid \sigma \in \mathbb{A}_2, \tau \in \mathbb{T} \,\}\,.$$

Furthermore, when building up composite states using information such as node types and symbols, which will be done in our second example below, it is useful to name some of the values in the sets on the left-hand side, so as to repeat them on the right-hand side. For instance, if $\mathbb{A}_0 = \{x, y\}$, and f is some function,

$$\langle v \in \{x, y\}, s, \mathbf{0} \to \uparrow, f(v) \rangle = \{\, \langle x, s, \mathbf{0} \to \uparrow, f(x) \rangle, \langle y, s, \mathbf{0} \to \uparrow, f(y) \rangle \,\}\,.$$

All the transitions from the set $\langle \mathbb{A}_0, Q, \mathbb{T} \to \{\swarrow, \searrow\}, Q \rangle \cup \langle \mathbb{A}, Q, \star \to \uparrow, Q \rangle$, corresponding respectively to moving down to the children of a leaf, and to moving

up to the father of the root, are considered invalid and should not appear in any well-formed automaton.

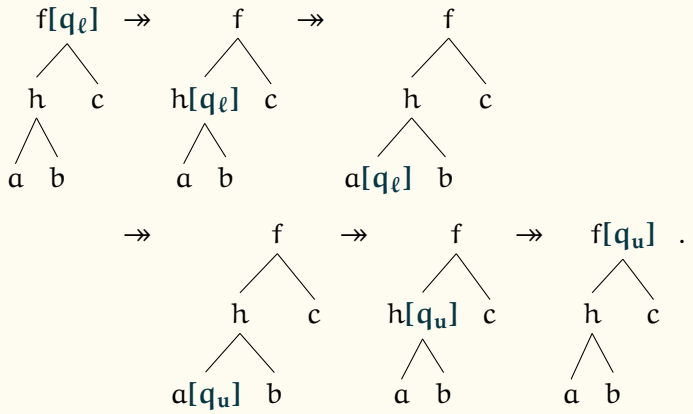
Let $\mathcal{A}$ be a tree-walking automaton. A *configuration* of $\mathcal{A}$ on a tree t is a pair $c = (\beta, q) \in \mathcal{P}(t) \times Q$; it is *initial* if $c \in \{\varepsilon\} \times I$ and *final* (or *accepting*) if $c \in \{\varepsilon\} \times F$. It is a *successor* of a configuration (α, p) if $\langle t(\alpha), p, \mathrm{ty}\,\alpha \to \mu, q\rangle \in \Delta$, where μ is $\uparrow$ if $\beta = \mathsf{parent}(\alpha)$, $\circlearrowleft$ if $\beta = \alpha$, $\swarrow$ if $\beta = \alpha.0$ and $\searrow$ if $\beta = \alpha.1$. We write $c_1 \twoheadrightarrow_{\mathcal{A}} c_2$ (or simply $c_1 \twoheadrightarrow c_2$ whenever $\mathcal{A}$ is clear from the context) if the configuration c_2 is a successor of c_1. A *run* is a sequence of successive configurations $c_1 \twoheadrightarrow c_2 \twoheadrightarrow \cdots c_n \twoheadrightarrow \cdots$. A run is *accepting* (or *successful*) if it starts with an initial configuration and reaches a final configuration. As usual, a tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t.

run (TWA)

Our first example is the TWA $\mathcal{X}$, which will also serve as the running example of Chapter 9[p165]. $\mathcal{X}$ is defined such that

$$\mathbb{A} = \{a, b, c/_0, f, g, h/_2\}, \quad Q = \{q_\ell, q_u\}, \quad I = \{q_\ell\}, \quad F = \{q_u\},$$
$$\Delta = \langle \mathbb{A}_2, q_\ell, \{\star, \mathbf{0}\} \to \swarrow, q_\ell\rangle \cup \langle a, q_\ell, \{\star, \mathbf{0}\} \to \circlearrowleft, q_u\rangle \cup \langle \mathbb{A}, q_u, \mathbf{0} \to \uparrow, q_u\rangle\,.$$

In total, $\mathcal{X}$ has two states and fourteen rules. Let us consider the tree $f(h(a, b), c)$; we have the following execution:



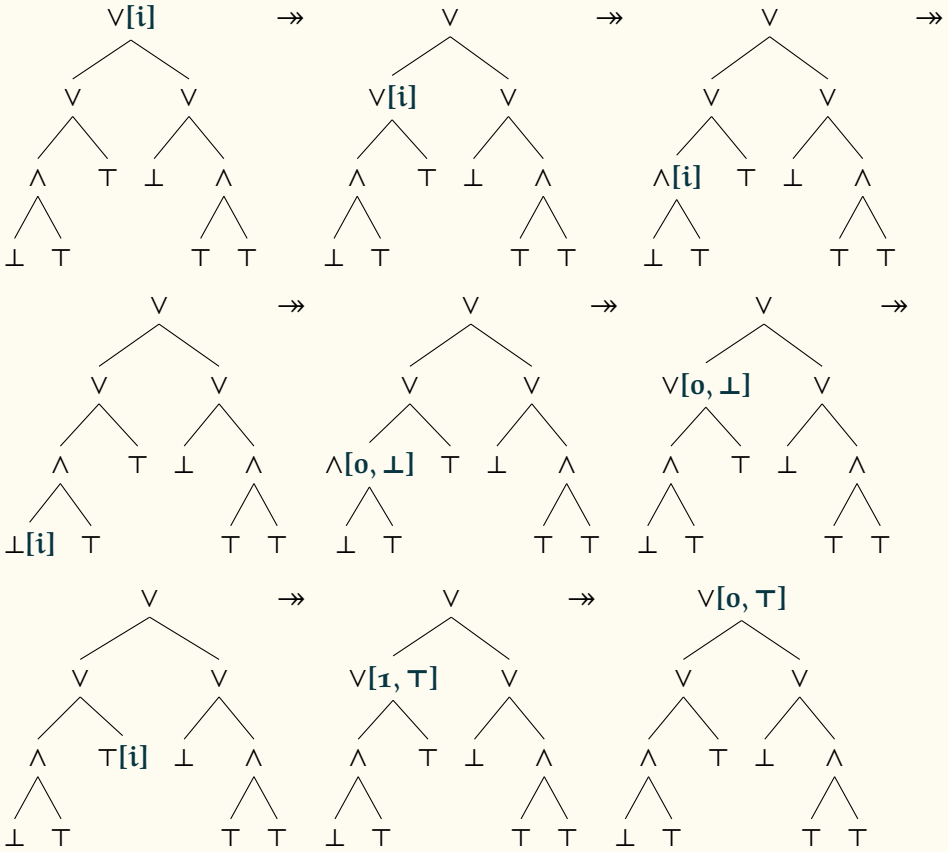
This run makes use of the three subsets of rules in the order in which they have been given in the definition. First, $\langle \mathbb{A}_2, q_\ell, \{\star, \mathbf{0}\} \to \swarrow, q_\ell\rangle$ takes effect: the initial state q_ℓ causes the head to go down and left, all the way to the leaves. At the leaves, those rules cannot apply, since they are only defined on binary nodes. Then, by $\langle a, q_\ell, \{\star, \mathbf{0}\} \to \circlearrowleft, q_u\rangle$, the head uses a stationary move, changing its state to q_u while staying in place. This is only possible because the leaf is labelled by a, otherwise the head would be stuck. Lastly, $\langle \mathbb{A}, q_u, \mathbf{0} \to \uparrow, q_u\rangle$ takes over, and the head goes all the way up to the root, staying in state q_u, which is final, and thus accepting the term as soon as the root is reached. It is easy to see that $\mathcal{X}$ recognises exactly all trees whose left-most leaf is labelled by a, including the trivial tree a, accepted by the immediate run $a[q_\ell] \twoheadrightarrow a[q_u]$ thanks to the rule $\langle a, q_\ell, \star \to \circlearrowleft, q_u\rangle$.

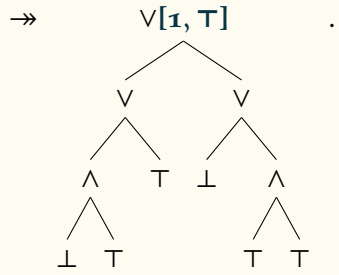
This is not a very interesting language, however. Let us see how a TWA can recognise the language of true variable-free propositional logic formulæ, so as to contrast it with its BUTA counterpart seen in the previous section. The idea is, when confronted with an internal node, to first evaluate one of the subtrees, e.g. the left subtree. For instance, schematically denoting the position of the head by $*$, let us say we have the following configuration: $\wedge(\vee_*(\top, u), v)$, where the

head has already explored the left subtree $\top$, and knows that it evaluates to true. Then in that case there is no need to explore the right subtree u, and the head can immediately move up, reporting the entire subtree $\vee(\top, u)$ as evaluating to true. This brings it into the configuration $\wedge_*(\vee(\top, u), v)$; the head knows that the left subtree is true, and the current symbol is $\wedge$, which requires both its children to be true. Thus, the head must visit the right child: $\wedge(\vee(\top, u), v_*)$. Whether v evaluates to true or false, the entire tree will evaluate to v; thus, when the head goes back up on $\wedge$ with the result of the evaluation of the right subtree v, it can immediately carry the information further up, without any change. This idea is implemented in the TWA below, using i for the initial state (visiting a tree for the first time), and (t, v) for the other states, storing the type of the last visited subtree in t, and the value of that subtree in v.

$$
\begin{aligned}
&\mathbb{A} = \{\wedge, \vee/_2, \top, \bot/_0\}, \quad Q = i \cup (\mathbb{S} \times \{\top, \bot\}), \quad I = \{i\}, \quad F = \{(\mathbf{1}, \top)\},\\
&\Delta = \langle v \in \mathbb{A}_0, i, \star \to \circlearrowleft, (\mathbf{1}, v)\rangle \cup \langle v \in \mathbb{A}_0, i, t \in \mathbb{S} \to \uparrow, (t, v)\rangle \cup \langle \mathbb{A}_2, i, \mathbb{T} \to \swarrow, i\rangle\\
&\quad \cup \langle \wedge, (\mathbf{0}, \bot), \star \to \circlearrowleft, (\mathbf{1}, \bot)\rangle \cup \langle \wedge, (\mathbf{0}, \bot), t \in \mathbb{S} \to \uparrow, (t, \bot)\rangle\\
&\quad \cup \langle \vee, (\mathbf{0}, \top), \star \to \circlearrowleft, (\mathbf{1}, \top)\rangle \cup \langle \vee, (\mathbf{0}, \top), t \in \mathbb{S} \to \uparrow, (t, \top)\rangle\\
&\quad \cup \langle \wedge, (\mathbf{0}, \top), \mathbb{T} \to \searrow, i\rangle \cup \langle \vee, (\mathbf{0}, \bot), \mathbb{T} \to \searrow, i\rangle\\
&\quad \cup \langle \mathbb{A}_2, (\mathbf{1}, \top), t \in \mathbb{S} \to \uparrow, (t, \top)\rangle \cup \langle \mathbb{A}_2, (\mathbf{1}, \bot), t \in \mathbb{S} \to \uparrow, (t, \bot)\rangle
\end{aligned}
$$

Note that the alphabet does not include $\neg$, simply because it is a unary symbol while we are working with a binary alphabet. It would be easy to implement a binary symbol $\neg/_2$, which simply ignores its left subtree and negates the value of its right. Extending the example to do precisely that is left as an exercise for the reader. In the meantime, the following execution on a tree with only $\wedge$ and $\vee$ is sufficient to bring most of the transitions into play:





In that case, the BUTA version was much more compact; on the other hand, the TWA version is lazy, in that it does not actually need to explore all the tree to evaluate it. This is a useful quality of TWA in general, which makes them attractive, and often more convenient than BUTA in circumstances when the aim is to locate a specific subtree, and the context does not really matter. In particular, they constitute a straightforward model of XML path expressions, which describe a navigation along the nodes of the tree. This, and other connexions with XML, have greatly contributed to the current research interest in TWA and their variants. However, their expressive power is strictly less than that of BUTA: they still tend to get lost in trees. More is said about TWA in section 2.6[p37] and in Part IV. Meanwhile, the reader is invited to consult [Bojańczyk, 2008] for a survey of TWA and variants, – especially from the viewpoint of expressive power – and [Hosoya, 2010, Chap. 12] for an overview of path expressions, XPath, Caterpillar expressions, TWA, and their mutual relationships. The next sections survey such material.

8.2 Abstracting Away Unranked Trees

It has been casually remarked in section 8.1[p144] that it could be assumed, without loss of generality, that one was dealing with ranked, even binary trees. Indeed, other kinds of trees, whether non-binary ranked trees or unranked trees, can be transformed into binary trees, and this transformation can be mirrored into the corresponding tree acceptors. It therefore suffices to study binary trees, and any result thus obtained can automatically be transferred to more general models. Since this choice makes for considerably smoother exposition and shorter proofs, a large proportion of the literature on semi-structured documents – objects which are most naturally described by unranked trees – is written under the assumption of a binary model. From what we have seen at least, it is a quasi-pervasive convention when it comes to the literature on tree-walking automata, and an extremely convenient shortcut of which we shall avail ourselves as well in our own contributions.

Nevertheless, this description of the – undeniable – ubiquity and convenience of the binary approach must be tempered by a few caveats regarding the exact sense in which results are "transferred" from one model to the other. A short presentation of the classical binarisation processes is therefore in order.

8.2.1 Unranked Trees and Their Automata

We begin by a quick definition of unranked trees, which we have only mentioned in passing so far. Although they have appeared as early as in the nineteen-sixties

and -seventies, within works of Pair and Quere, Thatcher, and Takahashi, it is only recently – late nineties, early 2000s – that they have attracted much research interest, a resurrection which owes much to their numerous and immediate XML-related applications. The next section serves to illustrate that by taking the example of DTD, an essential component of day-to-day activity with XML documents, which are direct applications of unranked tree automata.

As one could surmise, the nub of the unranked model is simply to disregard the arity of all symbols. Any position of an unranked tree, regardless of the symbol which it holds, may therefore admit any number of children. Not all structure is abandoned, however, as the children must remain finite in numbers, and ordered. One can therefore still characterise the children using ordinals – first child, second child, etcetera – and refer to the next or previous sibling and so forth. A *hedge* being defined as a finite, possibly empty sequence of unranked trees – a terminology first introduced in [Bruggemann-Klein, Murata & Wood, 2001] – an *unranked tree* is a hedge, coiffed with a functional symbol. Syntactically, we write

hedge

unranked tree

$$u := \sigma(h) \qquad h := \varnothing \mid u : h \qquad \sigma \in \mathbb{A} \,,$$

where $\varnothing$ symbolises the empty sequence and $\mathbb{A}$ is an ordinary alphabet, i.e. not a ranked alphabet. For the sake of simplicity, the unranked tree $a(\varnothing)$ is routinely denoted by $a()$ or a and a sequence $u_1 : \cdots : u_n : \varnothing$ by $u_1, \ldots, u_n$. For instance $f(a, b, c)$ is a shortcut to the unwieldy $f(a(\varnothing) : b(\varnothing) : c(\varnothing) : \varnothing) : \varnothing$ – but it must be kept in mind that this not the same object as the ranked tree $f(a, b, c)$. The context will always make clear whether we are dealing with ranked or unranked trees.

It is of course trivial to take any ranked alphabet $(\mathbb{A}, \mathsf{arity})$ and simply discard arity, and in that sense every ranked alphabet and every ranked tree are also unranked. The other direction is obviously a tad more thought-provoking, and is the object of section 8.2.3. We write $\mathcal{U}(\mathbb{A})$ for the set of unranked trees over $\mathbb{A}$, and $\mathcal{U}^*(\mathbb{A})$ for the set of hedges over $\mathbb{A}$. Furthermore, we assimilate a singleton hedge with the sole unranked tree which it contains.

There remains to define a suitable notion of acceptors for unranked trees. Recall the definition of bottom-up tree automata given in section 2.4[p30]. Disregarding the connexions with term-rewriting systems, this definition could as well have been given directly in terms of runs, and the transitions $\sigma(p_1, \ldots, p_n) \to q$, instead of being rewriting rules, could simply be seen as tuples $(\sigma, p_1, \ldots, p_n, q)$, so that

$$\Delta \subseteq \biguplus_{k \in \mathbb{N}} \mathbb{A}_k Q^{k+1} \equiv \biguplus_{k \in \mathbb{N}} \mathbb{A}_k \times Q^k \times Q \,,$$

isolating the target state in the last position. With unranked trees, there is no predicting how many children a given symbol may take, so the generalisation to unranked trees is of the form

$$\Delta \subseteq \mathbb{A} \times Q^* \times Q \,.$$

While the semantics of such transitions is intuitively clear by analogy to the ranked case, this definition leaves the door open to rather questionable choices, such as

$$\Delta = \{\, (f, m, q) \mid m \text{ encodes a terminating Turing machine} \,\} \,.$$

Thus it is clear that the overall complexity of decision problems for unranked tree automata is contingent upon the difficulty of representing and testing the

transitions. The usual solution is to limit the transitions to regular languages, so as to control the overall memory footprint of the unranked automaton. Even then, there is room for choosing different concrete representations: finite-state automata come to mind most readily, but two-way automata, alternating automata (*AFA*) and two-way alternating automata (*2AFA*, [Ladner, Lipton & Stockmeyer, 1984]), as well as all their deterministic variants, without forgetting regular grammars, regular expressions, weak monadic second-order logic over the successor relation, and many more, are all equally valid candidates, with varying degrees of efficiency and conciseness depending on the specifics of the languages and tasks at hand. Given the choice of such a class $\mathcal{C}$ of word automata – or another representation of regular languages – we define *unranked tree automata with* $\mathcal{C}$ (*UTA/*$\mathcal{C}$), also often called *hedge automata*, as the variant of BUTA such that

AFA

2AFA

unranked tree automata with $\mathcal{C}$

UTA/$\mathcal{C}$

$$\Delta \subseteq \mathbb{A} \times \mathcal{L}(\mathcal{C}) \times Q \,,$$

the alphabet underlying the class $\mathcal{C}$ being understood as the set of states Q. The unranked languages accepted by UTA are termed regular, like their ranked counterparts. Let us go back – one last time – to our perennial example of variable-free formulæ of propositional logic; instead of having fixed binary operators, as in section 2.4[p30], we can now handle variadic conjunction and disjunction operators. Taking the unranked alphabet $\mathbb{A} = \{\wedge, \vee, \neg, \top, \bot\}$, we define the states $Q = \{0, 1\}$, $F = \{1\}$, and the transitions are expressed by an extension of the usual $\to$ notation, and using regular expressions as $\mathcal{C}$:

$$\begin{array}{llll} \vee\big((0+1)^*1(0+1)^*\big) \to 1 & \vee(0^*) \to 0 & \neg(0) \to 1 & \top \to 1 \\ \wedge\big((0+1)^*0(0+1)^*\big) \to 0 & \wedge(1^*) \to 1 & \neg(1) \to 0 & \bot \to 0 \,. \end{array}$$

Note that a rule like $\top \to 1$ is shorthand for $(\top, \varepsilon, 1)$. Of course, the rules do not cover every possible tree that may be formed on the alphabet; the automaton will simply not run on a malformed tree – for instance if it contains $\neg(x, y)$ – and therefore such trees will be rejected regardless of the final state. On the other hand, trees such as $\wedge()$ and $\vee()$ will naturally be evaluated – to 1 and 0, respectively – which is the expected behaviour in logic.

Decision problems have been studied for various choices of $\mathcal{C}$; it is known in particular [Martens & Neven, 2003; Neven, 2002; Comon et al., 2008] that

(1) membership is testable in
$O(\|t\| \cdot \|B\|^2)$ for UTA/FSA, and it is
NP-complete for UTA/AFA.

(2) emptiness is
PTIME-complete for UTA/FSA and
PSPACE-complete for UTA/AFA or 2AFA,

(3) containment is EXPTIME-complete for UTA/2AFA,

(4) equivalence is EXPTIME-complete for UTA/2AFA.

Besides the problem of the choice of $\mathcal{C}$, it is worth noting that certain properties which are taken for granted with ranked automata do not carry over to their unranked cousins. For instance, it is known and relied upon that deterministic BUTA admit a unique – up to isomorphism – minimal automaton. With the usual

definition of determinism for UTA, stating that

$$\forall (\sigma, \ell, q), (\sigma', \ell', q') \in \Delta;\ q \neq q' \implies \ell \cap \ell' = \varnothing\ ,$$

and even assuming UTA/DFA, not only is there no unique minimal automaton, but the minimisation problem is even NP-hard [Martens & Niehren, 2005]. However, many results on ranked trees do carry over, as we shall see in section 8.2.3. Before that, let us say a few brief words about an important and direct application of UTA to semi-structured documents: DTD.

8.2.2 Document Type Definitions (DTD)

Document Type Definitions
DTD

Recall the example XML document illustrated by Fig. 1.1[p18], in section 1.3. *Document Type Definitions* (*DTD*), and more generally *Schema languages*, specify the general structure that a document must follow in order to be considered correct. For instance, this is a possible DTD for our rather frivolous "Star Trek"-flavoured running example:

```
<!DOCTYPE crew [
  <!ELEMENT crew (team*)>
  <!ELEMENT team (member+,starship)>
  <!ATTLIST team name CDATA>
  <!ELEMENT member (#PCDATA)>
  <!ELEMENT starship (#PCDATA)>
]>
```

Handling XML Attributes

As mentioned before, handling the unordered and non-duplicable aspects of attributes is tricky, and requires specialised tools outside the scope of this thesis. See [Hosoya, 2010, Chap. 15] for a presentation of this topic.

Attributes are generally abstracted – at least in first approximation – when reasoning about XML from a theoretical point of view; in Fig. 1.1, we have simply represented the attribute as just another node, so the actual tree is better described by the addition of a text node `name`, and the modification of `team` so that this node appears among its children:

```
  <!ELEMENT team (name,member+,starship)>
  <!ELEMENT name (#PCDATA)>
```

With this detail out of the way, what information does a DTD actually provide? The `DOCTYPE` instruction specifies the starting point, or outermost node, of the document, while each `ELEMENT` is a statement of the general form "in order to obtain a valid subtree of type `x`, the children need to follow some regular expression on the types". This is strikingly similar to the automaton model seen in the previous section. Indeed, let us transform the above DTD into an unranked tree automaton which accepts trees of the right structure, abstracting away the contents of the data nodes. This is always possible, as DTD are – strictly – less powerful than tree automata. We take the alphabet $\mathbb{A} = \{\texttt{crew}, \texttt{team}, \texttt{name}, \texttt{member}, \texttt{starship}\}$ and, to avoid the multiplication of notations, the states are simply defined as $Q = \mathbb{A}$; this will not introduce any ambiguity. Since the data value of the nodes are discarded, text elements will simply be considered as leaf nodes, thus a statement `<!ELEMENT x (#PCDATA)>` is translated by the rule $\texttt{x}(\varepsilon) \to \texttt{x}$, or simply $\texttt{x} \to \texttt{x}$. This takes care of `name`, `member` and `starship`, which are all data nodes. The last two

rules are

$$\texttt{team}(\texttt{name}\ \texttt{member}^{+}\texttt{starship}) \to \texttt{team}$$
$$\texttt{crew}(\texttt{team}^{*}) \to \texttt{crew}\,,$$

and the outermost element dictates the choice of the final state $F = \{\texttt{crew}\}$. This illustrate how natural the unranked model is for semi-structured documents. In the next section, we show how most results on ranked trees automatically carry over to the unranked case.

8.2.3 Binarisation of Trees and Automata

The crux of the matter is to adopt a systematic, ranked representation of unranked trees, and adapt unranked tree acceptors to work over this representation. For the sake of brevity, this section only presents the tree encodings themselves; the reader is referred to the literature for the full details.

first-child next-sibling
FCNS

There are many different kinds of encodings, the most common of which are the *first-child next-sibling* encoding (*FCNS*), and *tree currying* (*TC*). The FCNS encoding is probably the best established: it appears in [Knuth, 1997], and although we quote the third edition, it was already present in the 1968 edition, as referenced in [Takahashi, 1975, Thm. 4.3.1], and is often attributed to [Rabin, 1969]. It also appears in [Hosoya, 2010, Sec. 4.2] and [Neven, 2002, Sec. 4.2]. FCNS corresponds closely to a linked-list representation of data: it relies on the introduction of a fresh nullary symbol which can be seen as playing the role of a null pointer. It should be noted – and we shall come back on this point when discussing TC – that it actually deals with hedges, and not isolated unranked trees. The FCNS encoding and decoding functions are typed, for $\mathbb{A}$ an unranked alphabet and its corresponding binarised ranked alphabet $\mathbb{A}^{\#} = \{\#/_0\} \uplus \{\,\sigma/_2 \mid \sigma \in \mathbb{A}\,\}$, as

$$[\cdot]_{\#} : \mathcal{U}^{*}(\mathbb{A}) \to \mathcal{T}(\mathbb{A}^{\#}) \quad \text{and} \quad [\cdot]_{\#}^{-1} : \mathcal{T}(\mathbb{A}^{\#}) \to \mathcal{U}^{*}(\mathbb{A})\,.$$

The intuition behind the encoding is this: any first child remains a first child, but the encodings of its siblings, taken in order, become its right descendants. A leaf node takes # for its left child. A node with no next sibling takes # for its right child. Formally, we have

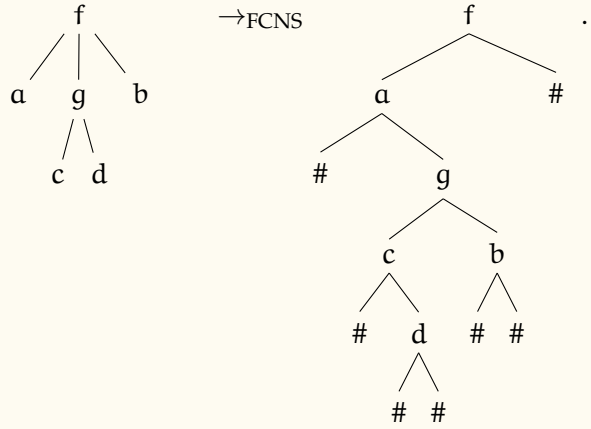
$$\left[\sigma(h) : h'\right]_{\#} = \sigma([h]_{\#}, [h']_{\#}) \quad \text{and} \quad [\varnothing]_{\#} = \#\,. \tag{8.1}$$

Let us take a simple example: the binarisation of the unranked term $f(a, b, c)$, or equivalently, of the singleton hedge $f(a(\varnothing) : b(\varnothing) : c(\varnothing) : \varnothing) : \varnothing$, which is what really is under the simplified notation:

$$\begin{aligned}
[f(a,b,c)]_{\#} &= [f(a(\varnothing) : b(\varnothing) : c(\varnothing) : \varnothing) : \varnothing]_{\#} \\
&= f([a(\varnothing) : b(\varnothing) : c(\varnothing) : \varnothing]_{\#}, [\varnothing]_{\#}) \\
&= f(a([\varnothing]_{\#}, [b(\varnothing) : c(\varnothing) : \varnothing]_{\#}), \#) \\
&= f(a(\#, b([\varnothing]_{\#}, [c(\varnothing) : \varnothing]_{\#})), \#) \\
&= f(a(\#, b(\#, [c(\varnothing) : \varnothing]_{\#})), \#) \\
&= f(a(\#, b(\#, c([\varnothing]_{\#}, [\varnothing]_{\#}))), \#) \\
&= f(a(\#, b(\#, c(\#, \#))), \#)\,.
\end{aligned}$$

to be folklore. Note that in [Neven, 2002], for instance, the example abbreviates $a(\#,\#)$ into a. This is a notational shortcut, and not a different encoding.

Graphically, and on a more complex example, we have the transformation:



(8.2)

It is easy to see that this transformation is a bijection, whose inverse can be straightforwardly defined as

$$\big[\sigma(b, b')\big]_{\#}^{-1} = \sigma([b]_{\#}^{-1}) : [b']_{\#}^{-1} \quad \text{and} \quad [\#]_{\#}^{-1} = \varnothing\,. \tag{8.3}$$

We can immediately check that $[[h]_{\#}]_{\#}^{-1} = h$, for any hedge $h \in \mathcal{U}^*(\mathbb{A})$. It has been shown that this encoding preserves recognisability of languages, and furthermore, the constructions are polynomial. Specifically, it is known [Neven, 2002] that

(1) for every UTA/FSA $\mathcal{A}$, there exists a BUTA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = [\mathcal{L}(\mathcal{A})]_{\#}$ and $\|\mathcal{B}\| = O(\|\mathcal{A}\|^n)$, for some $n \in \mathbb{N}$,

(2) for every BUTA $\mathcal{B}$, there exists a UTA/FSA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = [\mathcal{L}(\mathcal{B})]_{\#}^{-1}$ and $\|\mathcal{A}\| = O(\|\mathcal{B}\|^n)$, for some $n \in \mathbb{N}$.

Through this, all closure properties of binary tree automata carry over UTA: to recapitulate, they are closed by union, intersection and complementation. The above results were probably most clearly proven in [Suciu, 2001], although this paper does not use the exact FCNS encoding such as defined in (8.1) and (8.3). Whereas we only introduce one fresh nullary symbol #, Suciu's construction involves an additional fresh binary symbol serving as a backbone of sorts for the binarised tree – or as the cells of a linked list. Apart from that, the idea is pretty much the same.

Furthermore, all decidability results carry over as well, and since the encoding and decoding functions can be expressed in weak monadic second-order logic with child and next-sibling relations (*WMSO*), the regular unranked tree languages are characterised by WMSO, thus extending the results for WSkS previously mentioned for BUTA [Neven, 2002, Sec. 4.3]. However, it should be noted that complexity results do not carry over directly, as they depend in fine upon the choice of the class $\mathcal{C}$ in the unranked representation, as discussed at the end of section 8.2.1.

WMSO

It was briefly noted above that deterministic unranked tree automata are not as well-behaved as their ranked counterparts. Deterministic automata may become exponentially larger after the encoding, if the target is to be deterministic as well. And then, minimisation is difficult, in great part because any reasonable definition of the size of an UTA must account for that of the $\mathcal{C}$ representations. Indeed, the transitions may be split while leaving the language unchanged, like so:

$$\{\,\sigma(\ell) \to q\,\} \equiv \{\,\sigma(\ell_1) \to q, \ldots, \sigma(\ell_n) \to q\,\} \qquad \text{with } \bigcup_{i=1}^{n} \ell_i = \ell\,.$$

While this operation augments the number of rules, it does not follow that the overall size of the unranked automaton follows suit and increases. This is quite unlike the behaviour of ranked automata, and is understood by looking at the definition of the "size" of an unranked automaton:

$$\|\langle \mathbb{A}, Q, F, \Delta \rangle\| = |Q| + \sum_{(\sigma, \ell, q) \in \Delta} \big(2 + \|\ell\|\big) \,,$$

where $\|\ell\|$ is of course the size of the $\mathcal{C}$-representation of the language, and not the cardinal of the language itself. Thus, if splitting a rule in a certain way allows for much more compact representations of some of the sub-languages ℓ_i, the global size may actually decrease despite the automaton having a higher number of individual rules. Another related problem is the lack of a Myhill-Nerode theorem for UTA.

tree currying *TC*

We shall now see a second binary encoding, the *tree currying* (*TC*) encoding, which not only allows a transfer of closure and decidability properties, but also provides good properties with respect to determinism and minimisation. This encoding was presented and studied in [Carme, Niehren & Tommasi, 2004; Martens & Niehren, 2005], although the idea of the extension operator that underlies it was already present three decades before in [Takahashi, 1975, Def. 4.2.1]'s **l**-operator – albeit in reversed form.

The extension operator $@ : \mathcal{U}(\mathbb{A}) \to \mathcal{U}(\mathbb{A})$ simply inserts its second argument as the last sibling of its first:

$$\sigma(u_1, \ldots, u_n) \mathbin{@} t = \sigma(u_1, \ldots, u_n, t) \,. \tag{8.4}$$

currying

The intuition is to see a term as a λ-term, or function application, and to apply the well-known *currying* operation, through which multi-ary functions are naturally transformed into unary functions and vice versa, a process related to partial application. For instance, the following are the signatures of a binary function f and its curried counterpart f_c:

$$f : D_1 \times D_2 \to C \quad \equiv \quad f_c : D_1 \to (D_2 \to C) \,,$$

Note that we use the $f\,x$ notation for function application, as is common in programming languages rooted in λ-calculus. Since curried functions are the default view, the parentheses would be cumbersome in such contexts. Nevertheless, we do not abbreviate $(f\,x)\,y$ into $f\,x\,y$ – as is customary – in this discussion.

and both functions are equivalent in the sense that $f(x, y) = (f_c\,x)\,y$, for all $x \in D_1$ and $y \in D_2$. Since trees can naturally be interpreted as describing function applications, one can apply this reasoning to them as well, and thus

$$f(a, b, c) \equiv ((f\,a)\,b)\,c \,.$$

Now, using an explicit binary operator @ for function application – that is to say, writing $f \mathbin{@} x$ instead of $f\,x$ – this becomes

$$f(a, b, c) \equiv ((f \mathbin{@} a) \mathbin{@} b) \mathbin{@} c \,,$$

and thus we see that definition 8.4 actually translates function application, as its execution to the above yields back the original term $f(a, b, c)$. Since function application – and therefore the extension operator – is by definition a binary operation, this suggests a new binary encoding, targeting the binary alphabet $\mathbb{A}^{@} = \{\, \sigma/_0 \mid \sigma \in \mathbb{A} \,\} \uplus \{@/_2\}$. We take

$$[\cdot]_{@} : \mathcal{U}(\mathbb{A}) \to \mathcal{T}(\mathbb{A}^{@}) \quad \text{and} \quad [\cdot]_{@}^{-1} : \mathcal{T}(\mathbb{A}^{@}) \to \mathcal{U}(\mathbb{A}) \,,$$

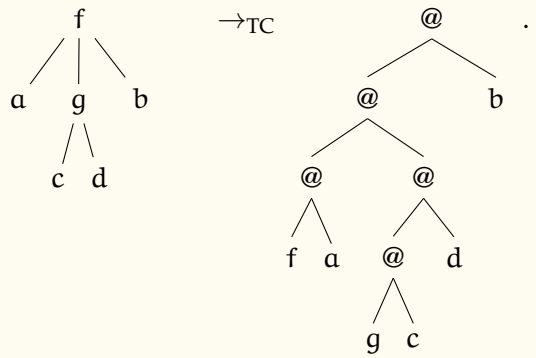
and the transformation is defined as

$$[\sigma(u_1, \ldots, u_n)]_@ = @(\sigma(u_1, \ldots, u_{n-1}), u_n) \quad \text{and} \quad [a]_@ = a \,,$$

its inverse being

$$\left[@(t, t')\right]_@^{-1} = t \mathbin{@} t' \quad \text{and} \quad [a]_@^{-1} = a \,.$$

On the same example term as earlier, we have the new encoding



(8.5)

Again it is easy to see that $[[u]_@]_@^{-1} = u, \ \forall u \in \mathcal{U}(\mathbb{A})$, and therefore this establishes a bijection between unranked trees and their curried binary encoding. Note that this is in contrast to FCNS, which only provided a one-to-one correspondence between the binary encoding and hedges; more precisely, any FCNS encoding e such that $e(2) \neq \#$ does not correspond to an unranked tree. There is no such problem with TC.

A *stepwise automaton* (*SA*) is simply a BUTA running on a TC-encoding alphabet $\mathbb{A}^@$ – though there are other equivalent characterisations. By dint of the above, the same automaton can also be considered to run on the curried version of unranked trees, and in that sense it accepts two languages, one ranked, and the other unranked. This establishes once again the closure and decision properties of unranked tree languages. Furthermore, SA directly inherit the nice minimisation properties and the Myhill-Nerode theorem of ranked tree automata.

stepwise automaton
SA

We shall see in section 8.4.1 that binarisation works very well for tree-walking automata as well.

8.3 Queries, Path Expressions, and Their Automata

The problems which we have seen so far have been rather global in scope: an entire document – a tree – or a DTD – an automaton – is validated or manipulated in some way. In this section, we look into a slightly different kind of operations which, instead of yielding a clear-cut answer to a polar question, selects nodes or subtrees. Such an operation is called a *query*. More precisely, a query q is a mapping from a tree t to a subset of its nodes, or more generally to a set of tuples of nodes. That is to say

query

$$q : \mathcal{T}(\mathbb{A}) \to \wp(\mathcal{P}(t)^n) \,,$$

If $n = 0$, one can assimilate the singleton containing the empty tuple with true, and the empty set with false…

for some $n \in \mathbb{N}_1$. The usual vocabulary of adicity applies to queries: for instance if $n = 1$ we speak of a unary or monadic query – and we shall see this is the most commonly used kind. There are of course many ways to define queries; we start off with declarative, logic-based queries, and move on progressively to more procedural, automata-focused methods.

8.3.1 Logic-based Queries

Recall the discussion of predicate logic formulæ as word and tree acceptors, at the end of section 2.4[p30]. The example formula φ given there is a sentence – it has no free variables – and this is why, given a tree, it has a fixed truth value, and therefore defines an acceptor. Consider now the formula

Reminder: $S(\alpha, \beta)$ holds if the position β is the immediate successor of α, and $a(\alpha)$ if $w(\alpha) = a$.

$$\psi = a(\alpha) \implies \exists \beta : S(\alpha, \beta) \wedge b(\beta) \,,$$

which, unlike φ, has a free variable α. In order to obtain a fixed truth value for ψ, one needs to interpret it both in reference to a tree, and according to an assignment of its free variable. Thus, if the word or tree t is fixed, the formula can be interpreted as defining a subset $\psi(t)$ of nodes of t such that the formula is satisfied:

$$\psi(t) = \{\, \alpha \in \mathcal{P}(t) \mid \ \models \psi[\alpha] \,\} \,.$$

For instance, considering the word $w = \mathtt{abracadabra}$, and numbering the positions from 1 to 11, we have $\psi(w) = \{1, 8\}$. Of course, there may be more than one free variable, in which case we let

$$\psi(t) = \{\, (\alpha_1, \ldots, \alpha_n) \in \mathcal{P}(t)^n \mid \ \models \psi[\alpha_1, \ldots, \alpha_n] \,\} \,.$$

Thus predicate logic is a natural tool to define queries; unfortunately it is almost too good a tool for its own sake, as the model-checking problem is known to be PSPACE-complete for both first-order and monadic second-order logic, and of course, actually exhibiting a variable assignment that satisfies the formula is at least as hard as determining whether it is satisfied by a given assignment. This in itself does not not necessarily entail intractability, however. For instance LTL model-checking – equivalently, model-checking of monadic first-order logic of order on words, by Kamp's theorem – is PSPACE-complete as well, and yet it is has proven itself invaluable for practical applications.

In order to understand how that can be, one must look at the parametrised complexity instead of the combined complexity. In the case of LTL, it is roughly $O(2^k \cdot n)$, where n is the size of the structure and k that of the formula. Therefore, if the formula is fixed and small enough, the evaluation will be tractable, and scale linearly with the size of the input. Typically, it can be assumed that a query is relatively small, and that the document or database on which it runs may be very large. The question of whether a logic defines tractable queries is therefore generally considered equivalent to asking whether the complexity of the model-checking problem can be expressed as $O(f(k) \cdot p(n))$, where p is polynomial and f is some not-too-explosive computable function. This question was investigated in [Frick & Grohe, 2002], which found the answer to be negative with f elementary, for both first- and monadic second-order logics on words, under reasonable and widely held

complexity-theoretic assumptions, such as P ≠ NP. Thus the combined complexity cannot even be bounded by something of the form

$$2^{2^{\cdots^{2^k}}} \cdot p(n) \, .$$

This can be somewhat mitigated by bounding the width of the trees, as the complexity becomes linear in the size of the structure, i.e. p is linear; this was shown on graphs in [Courcelle, 1990]. Nevertheless, the non-elementary overhead is generally considered prohibitive. Thus, while the predicate logics are often used as yardsticks for the expressive powers of other query languages, they are deemed a smidgen too powerful to be used directly in practical settings. One approach to alleviate that problem has been explored in [Neven & Schwentick, 2000], where a fragment of monadic second-order logic appropriately dubbed *efficient tree logic* (ETL) is introduced. ETL has the same expressive power as monadic second-order logic for unary queries, but its model-checking problem is in $O(2^{k^2} \cdot n)$, and even $O(2^k \cdot n)$ for another equally expressive fragment. Similarly, μ-calculus – on ranked trees – and monadic datalog share the same expressive power but sport good complexities. The latter admits model-checking tests in time linear in both the size of the query and that of the tree, provided a suitable representation of the tree. Furthermore, most practical queries are succinctly expressed in this language, so that the lesser conciseness – with regards to monadic second-order logic – is not too serious a drawback [Gottlob & Koch, 2004].

8.3.2 (Core) XPath: a Navigational Language

Another approach to query languages is *navigation*, that is to say the specification of the path which one must follow and the tests which must be taken in order to reach the nodes of interest. Incidentally, this should remind the reader of the modus operandi of the tree-walking automata introduced in section 8.1[p144].

The most ubiquitous navigational language is certainly the W3C standard XPath [Consortium, 1999, 2010], which is used as the node-selecting sub-language of a number of other highly successful W3C tools, such as the XSLT transformation language, the XQuery query language and its update facility, the XPointer addressing language, and the standard schema language XML Schema. We shall not present the full syntax of XPath, but instead offer a simple example. The path expression

```
.//starship[/captain/species/human]/crew
```
(8.6)

selects all `crew` nodes which are descendants (`//`) of the current node (`.`) and sons of `starship` nodes which are descendants of the current node as well, and which have a `captain` child (`/`) itself with a `species` child with a `human` child (`[ ]` is a test). In more prosaic terms, this query yields the set of all crews of all starships defined below the current node whose captain is human.

Note that this is a unary query if the starting point is assumed to be the root node, but if not, it defines a binary relation between starting nodes and end nodes, and is therefore a binary query. Hence the term "navigational language" as opposed to the more restrictive "selection language".

The full specification of XPath is quite large – about 30 pages for XPath 1.0 [Consortium, 1999] and 90 pages for XPath 2.0 [Consortium, 2010] – and it contains

a lot of features which add to the overall difficulty of evaluation, such as an arithmetic component. This complexity renders it difficult to study, which is why cleaner fragments of XPath have been isolated and examined independently of the whole specification. In particular the navigational core of XPath 1.0, singled out in [Gottlob, Koch & Pichler, 2002, 2005], only manipulates sets of nodes, and is referred to as *Core XPath*, or *Navigational XPath*. As its names suggest, it captures the navigational capabilities of full XPath, and discards the other features. It was shown in [Gottlob et al., 2005] that the combined complexity of query evaluation is linear for this fragment, while it is polynomial for the full language – and empirically exponential in most popular XSLT engines (Apache's XALAN, XT) and implementations within web browsers (Microsoft's Internet Explorer 6).

Core XPath 1.0

Core XPath 1.0 is a two-sorted language, where we distinguish path expressions π and node expressions ν. These are defined by the following grammar:

$$\begin{aligned}\pi &:= . \mid \uparrow \mid \downarrow \mid \rightarrow \mid \leftarrow \mid \uparrow^+ \mid \downarrow^+ \mid \rightarrow^+ \mid \leftarrow^+ \mid \pi/\pi \mid \pi \cup \pi \mid \pi[\nu] \\ \nu &:= \sigma \mid \langle\pi\rangle \mid \neg\nu \mid \nu \wedge \nu \mid \nu \vee \nu \qquad\qquad \sigma \in \mathbb{A}\,.\end{aligned}$$

Caveat lector: the semantics of $\rightarrow$ and $\leftarrow$ are inverted compared to that of [ten Cate & Segoufin, 2010].

The semantics are interpreted over a tree t, for which a path expression π defines a binary relation $[\![\pi]\!]_t \subseteq \mathcal{P}(t)^2$, and a node expression ν a set of nodes $[\![\nu]\!]_t \subseteq \mathcal{P}(t)$. First come the axes:

$$\begin{aligned}[\![.]\!]_t &= \{\,(\alpha,\alpha) \mid \alpha \in \mathcal{P}(t)\,\} && \text{self}\\ [\![\uparrow]\!]_t &= \{\,(\alpha.k,\alpha) \mid \alpha.k \in \mathcal{P}(t)\,\} && \text{parent}\\ [\![\downarrow]\!]_t &= \{\,(\alpha,\alpha.k) \mid \alpha.k \in \mathcal{P}(t)\,\} && \text{child}\\ [\![\rightarrow]\!]_t &= \{\,(\alpha.k,\alpha.(k+1)) \mid \alpha.(k+1) \in \mathcal{P}(t)\,\} && \text{next-sibling}\\ [\![\leftarrow]\!]_t &= \{\,(\alpha.(k+1),\alpha.k) \mid \alpha.(k+1) \in \mathcal{P}(t)\,\} && \text{previous-sibling}\end{aligned}$$

The remaining axes are defined as the transitive closure of the above, and are called ancestor, descendant, following-sibling and preceding-sibling.

$$[\![\leftrightarrow^+]\!]_t = [\![\leftrightarrow]\!]_t^+ \qquad\qquad \leftrightarrow \in \{\uparrow,\downarrow,\rightarrow,\leftarrow\}$$

Then we have the composite path expressions:

$$\begin{aligned}[\![\pi_1/\pi_2]\!]_t &= \{\,(\alpha,\gamma) \mid \exists\beta \in \mathcal{P}(t) : (\alpha,\beta) \in [\![\pi_1]\!]_t \wedge (\beta,\gamma) \in [\![\pi_2]\!]_t\,\}\\ [\![\pi[\nu]]\!]_t &= \{\,(\alpha,\beta) \in [\![\pi]\!]_t \mid \beta \in [\![\nu]\!]_t\,\}\\ [\![\pi_1 \cup \pi_2]\!]_t &= [\![\pi_1]\!]_t \cup [\![\pi_2]\!]_t\end{aligned}$$

And finally, the node expressions:

$$\begin{aligned}[\![\sigma]\!]_t &= \{\,\alpha \in \mathcal{P}(t) \mid t(\alpha) = \sigma\,\}\\ [\![\langle\pi\rangle]\!]_t &= \{\,\alpha \in \mathcal{P}(t) \mid \exists\beta \in \mathcal{P}(t) : (\alpha,\beta) \in [\![\pi]\!]_t\,\}\\ [\![\neg\nu]\!]_t &= \mathcal{P}(t) \setminus [\![\nu]\!]_t\\ [\![\nu_1 \wedge \nu_2]\!]_t &= [\![\nu_1]\!]_t \cap [\![\nu_2]\!]_t\\ [\![\nu_1 \vee \nu_2]\!]_t &= [\![\nu_1]\!]_t \cup [\![\nu_2]\!]_t\,.\end{aligned}$$

For instance, $\neg\langle\downarrow\rangle$ selects the leaves, $\neg\langle\uparrow\rangle$ the root, $\neg\langle\leftarrow\rangle$ all the first children, and $\neg\langle\rightarrow\rangle$ selects all the last children; as for the crew query (8.6), it is expressed by

$$(\,. \cup \downarrow^+)\big[\texttt{starship} \wedge \langle\downarrow[\texttt{captain}]/\downarrow[\texttt{species}]/\downarrow[\texttt{human}]\rangle\big]/\downarrow[\texttt{crew}]\,.$$

Beyond the results of the original paper, the definition of Core XPath 1.0 has been a very successful endeavour, which kindled interest in the theoretical study of XPath. A similar approach was taken in [ten Cate & Marx, 2007, 2009] for the second version of XPath which, unlike the first, was designed to be a full-fledged n-ary query language, expressively complete for first-order queries with descendant and following-sibling relations. This implies that model-checking is PSpace-complete for XPath 2.0, and thus a polynomial time evaluation algorithm is unlikely, as its existence is contingent on P = PSpace, which is widely assumed not to hold.

Among the interesting results stemming from the clean semantics of the Core XPaths, let us mention the complete axiomatisation of query equivalence for both versions 1.0 [ten Cate, Litak & Marx, 2007] and 2.0 [ten Cate & Marx, 2007, 2009]. The Core XPaths have also been compared with a large amount of fragments of predicate logics. For instance [Marx & de Rijke, 2005] shows that Core XPath 1.0 is equivalent to first-order logic with two variables and equipped with child, descendant and following-sibling relations. Later, [ten Cate & Segoufin, 2010] develops the natural extension to Regular Core XPath with subtree relativisation (*RXPathW*), obtained simply by adding the productions

RXPathW

$$\pi := \pi^* \qquad \nu := \mathsf{W}\nu \ ,$$

with the semantics

$$[\![\pi^*]\!]_t = [\![\pi]\!]_t^* \quad \text{and} \quad [\![\mathsf{W}\nu]\!]_t = \left\{ \alpha \in \mathcal{P}(t) \;\middle|\; \varepsilon \in [\![\nu]\!]_{t|\alpha} \right\} .$$

Caveat lector: the semantics in [ten Cate & Segoufin, 2010] is written $[\![\mathsf{W}\nu]\!]_t = \{ \alpha \in \mathcal{P}(t) \mid \alpha \in [\![\nu]\!]_{t|\alpha} \}$. Their notion of subtree appears to be defined as rooted in α, while our subtrees are rooted in ε.

The relativisation operator W permits to focus on a specific subtree for the purpose of evaluating a sub-query; it is unknown whether it actually increases the expressive power of Regular Core XPath. Moreover, the authors show this extension to be equivalent to first-order logic with monadic transitive closure (*FOT*), a logic strictly more expressive than basic first-order, but at most as powerful as MSO. FOT is then characterised by nested tree-walking automata – see section 8.4.2 below – and is thus shown to actually be strictly less expressive than MSO.

FOT

As we shall come back to this logic in the next section, it is worth defining more precisely what it is; FOT is first-order logic extended with a $+$ operator for taking the transitive closure of any first-order–definable binary relation. More specifically, if $\varphi(x, y)$ is a first-order formula, potentially with other free variables besides x and y, the transitive closure of φ with respect to x and y is the formula $+_{x,y}(\varphi)$, defined as being equivalent to the infinitary disjunction

$$\bigvee_{k \in \mathbb{N}_2} \exists z_1, \ldots, z_k : (x, y) = (z_1, z_k) \wedge \forall i \in [\![1, k-1]\!],\ \varphi(z_i, z_{i+1}) \ ,$$

which is not definable as a bare first-order formula [Fagin, 1975] – but would be in the second order. Transitive closure allows, for instance, to define the descendant relation, given the child relation.

A reader interested in a very comprehensive survey of XPath fragments and their characterisations will find that in [Benedikt & Koch, 2008], at least for results up to 2005.

8.3.3 Caterpillar Expressions

caterpillar expressions
CE

XPath is not the only approach to navigation and queries. Regular path queries [Abiteboul, Buneman & Suciu, 1999] and *caterpillar expressions* (*CE*) [Brüggemann-Klein & Wood, 2000] are other takes on that problem, independently developed although equivalent on unranked trees. They also coincide exactly with the expressive power of tree-walking automata [Bojańczyk, 2008; Salomaa, Yu & Zan, 2007, 2009, Thms. 11 & 3.1]. While [Brüggemann-Klein & Wood, 2000] introduced caterpillar expressions directly on unranked trees, we give a definition directly on binary trees, and more precisely, on the binary FCNS encoding of some unranked tree, in the style of [Hosoya, 2010, Sec. 12.1.2]. We therefore consider $\mathbb{A}^{\#}$ as our working alphabet, and use words of $\{0,1\}^*$ for positions.

A biologist's caterpillar is of course a colourful and hairy lepidopteran tree-crawler; it inches along from leaf to leaf, performing simple tests along the way to ascertain that it is not going to fall off. Back in computer science, a caterpillar expression captures similar sequences of actions. More specifically, it is a regular expression over a set of *caterpillar atoms*, consisting of moves and tests which are precisely the same as those introduced for TWA in section 8.1[p144], and for which we shall use the same notations, as well as tests for determining the label of the current node. The set C of caterpillar atoms is therefore given by

$$C = \{\uparrow, \swarrow, \searrow, \star, \mathbf{0}, \mathbf{1}\} \uplus \mathbb{A}^{\#} .$$

A finite word $c_1 \cdots c_n \in C^*$ is called a *caterpillar path*, and is said to *describe* a sequence of nodes $s = \alpha_1 \cdots \alpha_n \in \mathcal{P}(t)^*$ on a tree $t \in \mathcal{T}(\mathbb{A}^{\#})$ if the following holds for all $k \in [\![1, n-1]\!]$:

$$\begin{aligned}
c_k = \uparrow \quad &\Rightarrow \exists i \in \{0,1\} : \alpha_{k+1}.i = \alpha_k \\
c_k = \swarrow \quad &\Rightarrow \alpha_{k+1} = \alpha_k.0 \\
c_k = \searrow \quad &\Rightarrow \alpha_{k+1} = \alpha_k.1 \\
c_k = \sigma \in \mathbb{A}^{\#} \quad &\Rightarrow \alpha_{k+1} = \alpha \ \wedge \ t(\alpha) = \sigma \\
c_k = \star \quad &\Rightarrow \alpha_{k+1} = \alpha_k = \varepsilon \\
c_k = \mathbf{0} \quad &\Rightarrow \exists \beta \in \mathcal{P}(t) : \alpha_{k+1} = \alpha_k = \beta.0 \\
c_k = \mathbf{1} \quad &\Rightarrow \exists \beta \in \mathcal{P}(t) : \alpha_{k+1} = \alpha_k = \beta.1 .
\end{aligned}$$

A sequence of nodes s is described by a caterpillar expression e if there exists a caterpillar path $c \in \mathcal{L}(e)$ such that c describes s. Furthermore, e encodes a binary query in the sense that it selects all couples of nodes (α, β) such that some sequence $\alpha = \gamma_1 \cdots \gamma_n = \beta$ is described by e. We write $[\![e]\!]_t$ the set of couples selected by e. It should be noted that one could say exactly the same of tree-walking automata, and define them as selecting couples of nodes, although in that case one would prefer the alternative – and equivalent – definition of final states as accepting the current term immediately, without needing to go back to the root.

While the expressive powers of XPath and caterpillar expressions are incomparable, some XPath expressions can be expressed as CE. Recall that we are working on FCNS binarised trees; keeping in mind that the right child is the next sibling, we

can define expressions equivalent to all the standard XPath axes, as follows:

$$\uparrow = (\mathbf{1}\uparrow)^*\mathbf{0}\uparrow \qquad \downarrow = \swarrow\searrow^* \qquad \rightarrow = \searrow \qquad \leftarrow = \mathbf{1}\uparrow$$
$$\uparrow^+ = ((\mathbf{1}\uparrow)^*\mathbf{0}\uparrow)^+ \qquad \downarrow^+ = (\swarrow\searrow^*)^+ \qquad \rightarrow^+ = \searrow^+ \qquad \leftarrow^+ = (\mathbf{1}\uparrow)^+ .$$

Note that, in $\uparrow = (\mathbf{1}\uparrow)^*\mathbf{0}\uparrow$ for instance, the $\uparrow$ on the left is an XPath axis, while the $\uparrow$ on the right is a caterpillar atom.

To quickly see how that works, it is best to run those expressions on an example binarised tree, for instance (8.2)[p153]. With $\uparrow$, the head of the automaton – or the metaphorical caterpillar – goes up so long as it is on a right child, which corresponds to moving leftwards from sibling to previous sibling in the unranked tree; then, when it is a first child, it moves up, which also translates into moving up in the unranked tree. The caterpillar has therefore moved from some child to its parent. This is inverted with $\downarrow$, which, in the unranked tree, moves nondeterministically to any child of the current position. The remaining expressions are simpler, and should be self-explanatory at this point. Another classic example – borrowed from [Hosoya, 2010] – is a caterpillar expression which, starting at the root, explores all the nodes of the binary trees, starting with the left-most leaf:

$$(\swarrow^* \# (\mathbf{1}\uparrow)^* \mathbf{0} \uparrow \searrow)^* \# (\mathbf{1} \uparrow)^* . \tag{8.7}$$

It has been mentioned that tree-walking automata, and therefore caterpillar expressions, are strictly less powerful than branching automata, or equivalently, monadic second order logic. How do they compare to lesser – but still powerful – yardsticks of expressive power, such as XPath, first-order logic, and its transitive closure extension? Queries are given in [Goris & Marx, 2005, Prp. 2.6], which separate Core XPath 1.0 from caterpillar expressions and vice versa: the two are incomparable. The proof of [Bojańczyk & Colcombet, 2005] that $\mathcal{L}(\text{TWA}) \subset \mathcal{L}(\text{BUTA})$, as it turns out, also provides half the proof that first-order logic and caterpillar expressions have incomparable expressive powers, as they show the separation language which they exhibit to be definable in first-order logic. The other direction is much easier [Bojańczyk, 2008, Thm. 13], as some languages easily recognised by TWA are not expressible in first-order logic, such as the language of trees whose left-most path is of even length, or the language of true boolean expressions, for which we have explicitly built a TWA at the end of section 8.1[p144]. That these languages are not expressible in first-order logic is shown by *Ehrenfeucht–Fraïsse games*, which are a well-know technique for proving that kind of negative results. In particular, it is folklore that first-order logic, although sufficient to define any finite structure, can only express local properties, and that even expressing simple global notions such as "the domain has even cardinality" is beyond it.

Ehrenfeucht–Fraïsse games

This leaves caterpillar expressions in about the same place as Core XPath 1.0. On the one hand, they are sufficiently expressive for many applications; indeed [Brüggemann-Klein & Wood, 2000] points out that caterpillar automata – i.e. tree-walking automata – are strictly more expressive than needed to express *tree-local tree languages* [Takahashi, 1975], that is to say, the tree languages which are the set of derivation trees of some (extended) context-free grammar. This is quite useful, since many document grammar mechanisms define tree-local languages. On the other hand, not supporting all of first-order logic is a bit problematic, as that is often considered to be the least common denominator of respectable query languages; hence the various extensions to caterpillar expressions, of which we shall now present a few.

tree-local tree languages

cutting caterpillars

Following [Bojańczyk, 2008], we define *cutting caterpillars* as caterpillar expressions with three additional caterpillar atoms. The first and second are the positive and negative *nesting tests* $\langle e \rangle$ and $\langle \neg e \rangle$, where e is some caterpillar expression:

$$c_k = \langle e \rangle \quad \Rightarrow \alpha_{k+1} = \alpha_k \ \wedge \ \exists \beta : (\alpha_k, \beta) \in [\![e]\!]_t$$
$$c_k = \langle \neg e \rangle \quad \Rightarrow \alpha_{k+1} = \alpha_k \ \wedge \ \nexists \beta : (\alpha_k, \beta) \in [\![e]\!]_t \ .$$

The third is the *cutting command*, or subtree relativisation command, which we shall denote by W by analogy to RXPathW; it is semi-formally defined as:

$$c_k = \mathsf{W} \quad \Rightarrow \alpha_{k+1} = \alpha_k \ \wedge \ \alpha_k \text{ is root}, i \geqslant k \Rightarrow \alpha_i \trianglelefteq \alpha_k \ .$$

To clarify, the cutting command applies only within the scope of the current nesting level, and by "is root", we mean that within that level, the $\star$ test is redefined to report α_k as the root instead of ε. In essence, it causes all nodes beyond the subtree under α_k to disappear. The similarity of purpose to [ten Cate & Segoufin, 2010]'s relativisation operator is not coincidental, as cutting caterpillars are exactly as expressive as first-order logic with monadic transitive closure, itself exactly as powerful as Regular XPath with subtree relativisation, and nested tree-walking automata.

Positive Cutting

Caveat lector: [Bojańczyk, 2008, Thm. 12] defines only the first nesting test, and states that the positive fragment forbids "nesting commands under the scope of a negation", while there is no negation in CE. We have contacted the author to clarify that, and the definition given in this page is based upon his answer.

Another interesting subclass is that of the slightly less powerful *positive cutting caterpillars*, which are cutting caterpillars forbidden from using negative nesting commands. They capture exactly the expressive power of first-order logic with positive monadic transitive closure (*FOT+*) [Bojańczyk, 2008, Thms. 12 & 14], that is to say FOT where the transitive closure operator may not appear under the scope of a negation, and are thus equivalent to pebble automata, which we shall examine briefly in section 11.2.1[p196], in the appendix to this thesis. They are also equivalent to [Goris & Marx, 2005, Sec. 4]'s caterpillar expressions with variable binders, which are described as syntactic analogues of pebbles.

FOT+

looping caterpillars

Lastly, let us mention the *looping caterpillars* of [Goris & Marx, 2005, Sec. 2.3], which extend basic caterpillar expressions with a looping operation $[e]$ defined as

$$c_k = [e] \quad \Rightarrow \alpha_{k+1} = \alpha_k \ \wedge \ (\alpha_k, \alpha_k) \in [\![e]\!]_t \ .$$

This simple extension suffices for looping caterpillars to capture at least the expressive power of first-order logic – thereby making them more powerful than Core XPath 1.0 – while still keeping a polynomial-time combined complexity for query evaluation. The same paper also proposes an extension of caterpillar expressions with monadic datalog tests, which characterises MSO-definable binary queries.

8.4 The Families of Tree-Walking Automata

While discussing the expressive capabilities of the various formalisms presented above, a lot has been said already about various kinds of tree-walking automata, which we shall not repeat in this short section. Instead the focus is on providing short definitions and historical and bibliographic references for the various models. We also tersely summarise the relationships between the main query languages, logics and automata appearing this chapter.

8.4.1 Basic Tree-Walking Automata

The technical definition of this model, as well as a few examples, have already been given in section 8.1[p144]. Tree-walking automata have originally been introduced more than four decades ago, in [Aho & Ullman, 1969]. However, it should be noted that the original definition is not quite the same as that which we use in this thesis and which appears in the majority of the literature since then. As [Hosoya, 2010, Sec. 12.3] remarks, the 1969 definition does not include tests for the kind of the current node, which results in a much weaker model, incapable for instance of visiting all the nodes of a tree; a feat which comes easily to a model where such tests are available, as demonstrated by the caterpillar expression (8.7). It was shown in [Kamimura & Slutzki, 1981] that the weaker model was properly less powerful than BUTA. It should be said that the automata of [Aho & Ullman, 1969] were working in a specific context, as they were modelling syntax-directed string rewriting, and to do so they were provided with an underlying context-free grammar and an output tape; the reason why they did not test the kind of the nodes was that it would have been redundant for them to do so, as this information could already be encoded into the non-terminals of the grammar. The question for the stronger – and unarguably, standard – model remained open for quite some time, although the consensus was that the stronger model was probably still strictly weaker than BUTA; this was finally proven in [Bojańczyk & Colcombet, 2005]. The same authors went on the next year to close another long-standing open question, previously approached in a weaker context [Bojańczyk, 2003]: TWA cannot be determinised [Bojańczyk & Colcombet, 2006].

The initial context of research for TWA was tree transformations and attribute grammars [Aho & Ullman, 1969; Deransart, Jourdan & Lorho, 1988; Bloem & Engelfriet, 2000]. Currently, they owe a great deal of the renewed research interest directed towards them to the ever-growing popularity of XML. The first tangible connexion between TWA and XML was probably the development of caterpillar expressions – see the previous section – originally introduced in [Brüggemann-Klein & Wood, 2000], along with *caterpillar automata* which are actually almost exactly the same as – and exactly as expressive as – tree-walking automata. They are then used in the context of the validation of streaming XML documents, [Segoufin & Vianu, 2002], and in [Milo, Suciu & Vianu, 2003], a model extended with pebbles is central to the study of the decidability of type-checking for some XML transformation languages, including XML-QL and a fragment of XSLT.

The expressive power of TWA is incomparable with that of first-order logic, which we have already argued above, about the equivalent caterpillar expressions. They are also at least expressive enough to capture tree-local tree languages. It is shown in [Engelfriet & Hoogeboom, 1999, Sec. 3] that they capture all tree languages definable in locally first-order logic, meaning that the formulæ may speak about the parent-child relation, but not the more general ancestor relation.

Although deterministic TWA are not terribly powerful, they have the interesting property that, unlike the general, non-deterministic model, they are closed under complementation, which follows from the – potentially unintuitive – fact that every DTWA can be simulated by another DTWA that halts on all inputs [Sipser, 1978; Muscholl, Samuelides & Segoufin, 2006].

Membership is testable in polynomial time – linear for deterministic TWA – while the emptiness, containment and equivalence problems for tree-walking automata are shown to be ExpTime-complete in [Neven, 1999, Sec. 5], in the context of unranked trees. This carries over to ranked trees; note that we have, by [Neven, 2002, Prp. 4], an even better correspondence to the FCNS binary encoding than for branching automata:

(1) for every unranked (D)TWA $\mathcal{A}_u$, there exists a ranked (D)TWA $\mathcal{A}_r$ such that $\mathcal{L}(\mathcal{A}_r) = [\mathcal{L}(\mathcal{A}_u)]_\#$ and $\|\mathcal{A}_r\| = O(\|\mathcal{A}_u\|)$,

(2) for every ranked (D)TWA $\mathcal{A}_r$, there exists an unranked (D)TWA $\mathcal{A}_u$ such that $\mathcal{L}(\mathcal{A}_u) = [\mathcal{L}(\mathcal{A}_r)]_\#^{-1}$ and $\|\mathcal{A}_u\| = O(\|\mathcal{A}_r\|)$.

Let us note a critical difference between the ranked and unranked models, however: a ranked TWA can go up to check the label of the parent, and go back where it was. An unranked TWA cannot, because there is no bound on its current position – it may be at the third child, or the 3^{333}-rd – and that position can therefore not be remembered with finite memory. This is even clearer when considering a binary TWA running on a FCNS encoding: the head would get lost. This can be solved with just one pebble, using the terminology of section 11.2.1[p196].

TWA can be converted into branching automata, with an exponential blowup in the number of states. We conduct a detailed study of this transformation in our own contributions, presented in the next chapter.

8.4.2 Nested Tree-Walking Automata

Quick Tips

Back-references to each citation are included at the end of the thesis. PDF: follow the hyperlinks.

Nested tree-walking automata have been introduced in [ten Cate & Segoufin, 2010], the results of which have already been discussed multiple times in this chapter. In particular, they characterise exactly first-order logic with monadic transitive closure, and RXPathW, as well as cutting caterpillars.

For the sake of self-containedness, we just give a brief informal definition of nested TWA. Basic TWA – using the variant where a run is accepting as soon as a final state is used – are considered to be 0-nested. For $k \geqslant 1$, a k-nested TWA $\mathcal{A}$ is a TWA that contains a finite collection of $(k-1)$-nested TWA $\mathcal{B}_i$, and such that each transition of $\mathcal{A}$ may be contingent upon a number of conditions on the $\mathcal{B}_i$, each of which may be required to have (resp. not to have) an accepting run starting in the current node, without restrictions (resp. contained in the subtree rooted in the current node).

This model does not seem to have been used outside of [ten Cate & Segoufin, 2010], at the time of writing.

Chapter 9
Loops and Overloops: Effects on Complexity

Contents

—Where tree-walking automata save time by jumping off trees.

As we have seen in the previous chapter, tree-walking automata (TWA) and their many relatives have lately been the object of renewed research interest, thanks to their tight connexions to XML. In this chapter, we focus on an important algorithm on TWA: the transformation into an equivalent branching automaton – more specifically, into a bottom-up tree automaton (BUTA) – which is classically based on the somewhat folklore notion of "tree loop".

We give a formal treatment of tree loops, introduce the closely related notion of tree overloops, and investigate the use of both for the following common operations on TWA: deciding membership efficiently, building equivalent BUTA, and deciding or positively approximating emptiness. Notably, we argue that the transformation into a BUTA is slightly less straightforward than was previously assumed in the literature, show that using overloops yields much smaller BUTA in the deterministic case, and provide a polynomial over-approximation of this construction, capable of detecting emptiness with surprising accuracy – given that emptiness is an ExpTime-complete problem – against randomly generated TWA.

The results appearing in this chapter have been published in [Héam, Hugot & Kouchnarenko, 2011, 2012b].

If this was not already done, the reader is invited to consult section 8.1, and in particular definition 8.1[p145] and what follows, where the technical and notational prerequisites for this chapter are introduced.

9.1 Introduction

In light of the applications of TWA to XML, it becomes crucial to have reasonably efficient algorithms for essential operations on TWA such as deciding membership and emptiness, as well as the transformation into a BUTA. Until now, research has been mainly focused on closing fundamental open problems concerning the expressive power of TWA, in particular their relationship with regular languages, whether they are determinisable, etcetera; refer to section 8.4.1[p163] for a survey of such work. While algorithms for the above operations are known, they appear in print mostly as proof sketches, and there has been no focus on finding tighter complexity bounds. In contrast, this chapter provides explicit algorithms for these tasks and deals with complexity issues. The common thread of our contributions is the notion of *tree loop*, which is pervasive to the algorithms we give. This notion is closely related to Knuth's construction for testing circularity of attribute grammars [Knuth, 1968], and is a generalisation to trees of a similar construction for two-way word automata [Shepherdson, 1959]. The contributions are organised as follows:

- Section 9.2.1 gives a thorough introduction to tree loops – the basic idea of which is more or less folklore – and lays the groundwork for a new notion of *tree overloop* which we then introduce in Sec. 9.2.3[p172]. Simple algorithms for testing membership follow naturally from this work; beyond the immediate application of the recursive definitions of loops and overloops, a more efficient method based on a boolean matrix encoding of loops is given in Sec. 9.2.2[p170]. To the best of our knowledge, no such algorithm existed in the literature.

- Section 9.3[p174] deals with the transformation from TWA to BUTA, based on the proof sketches in [Bojańczyk, 2008] and [Samuelides, 2007, p143]. Two variants are given in Sec. 9.3.1: one using loops and another using overloops. Section 9.3.2 proceeds to show that, in the deterministic case, the overloops-based construction admits a much smaller upper bound on the number of generated states.

- The emptiness problem is known to be ExpTime-complete for TWA, and is traditionally tested by first transforming the TWA into a BUTA, and then invoking the usual linear emptiness test on the latter. Section 9.4[p179] provides a polynomial-time algorithm which computes an "over-approximation" of this BUTA, and thus may decide emptiness positively. Should it prove inefficient against some families of TWA, then the approximation can be refined as much as needed.

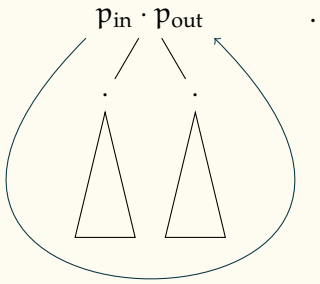
- Section 9.5[p181] presents random experiments performed to confirm our theoretical results. They involve both an *ad-hoc* random generation scheme for non-deterministic TWA, and a more interesting one, based on the results of [Héam, Nicaud & Schmitz, 2009], that yields complete and deterministic TWA according to the uniform probability distribution – which imparts statistical significance to our results. The dependability of the approximation method developed in Sec. 9.4 is tested in Sec. 9.5.1 – it is shown to be astonishingly accurate against both schemes. Section 9.5.2 compares the respective sizes of the BUTA obtained from the loops and overloops-based transformations, and

shows that overloops yield much smaller BUTA than loops *in average*. It is also shown that this size gain is independent of – and cumulates with – post-processing cleanup (cf. [Héam, Hugot & Kouchnarenko, 2010a]) of the BUTA. The ideas of these tests are illustrated on our running example, the small TWA $\mathcal{X}$ given in Sec. 8.1[p144], then validated against the above-mentioned uniform random generation scheme.

9.2 Loops, Overloops and the Membership Problem

9.2.1 Defining, Classifying and Computing Loops

The notion of *loop* turned out to be very useful to deal with TWA. Informally, loops arise naturally as a generalisation of the definition of an accepting run, where the automaton enters the root in a given initial state p_{in}, moves along the tree, and then comes back to the root in a certain final state p_{out}. In practice, the details of the moves which form the loop itself are largely irrelevant and are discarded: the most useful information is the pair of states (p_{in}, p_{out}).



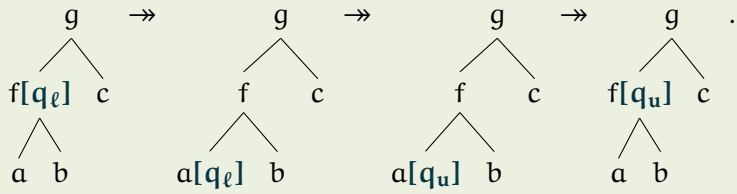
▽ Definition 9.1: Tree Loops

Let $\mathcal{A}$ be a TWA, t a tree and $\alpha \in \mathcal{P}(t)$. A pair of states $(p, q) \in Q^2$ is a *loop* of $\mathcal{A}$ on the subtree $t|_\alpha$ if there exist $n \geqslant 0$ and a run *tree loop*

$$(\alpha, p), (\beta_1, s_1), \ldots, (\beta_n, s_n), (\alpha, q)$$

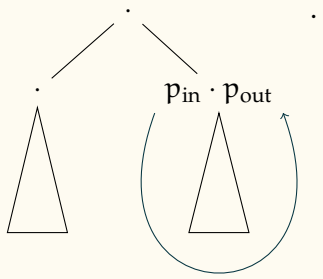
such that $\beta_k \trianglelefteq \alpha$ for all $k \in [\![1, n]\!]$. Such a run is a *looping run*, and we say that it *forms* the loop (p, q).

Example: The looping run $(0, q_\ell), (0.0, q_\ell), (0.0, q_u), (0, q_u)$ of $\mathcal{X}$ on the subtree $g(f(a, b), c)|_0 = f(a, b)$ forms the loop (q_ℓ, q_u):



◊

Notice that loops are not only defined on whole trees, but on subtrees as well, with the restriction that the automaton cannot leave the subtree during the looping run.



It is in fact this restriction which grants loops their usefulness. TWA, unlike their branching cousins, whose runs are defined inductively, do not naturally lend themselves to inductive reasoning; and yet, thanks to the above restriction, loops are easily computed by induction. Thus loops and their variants can be thought of as convenient devices which hide the sequential, stateful aspect of TWA runs beneath a much more "user-friendly" layer of induction.

In the next few paragraphs we compute the loops of a TWA $\mathcal{A}$ on a subtree $t|_\alpha$.

▽ Definition 9.2: Kinds of Loops

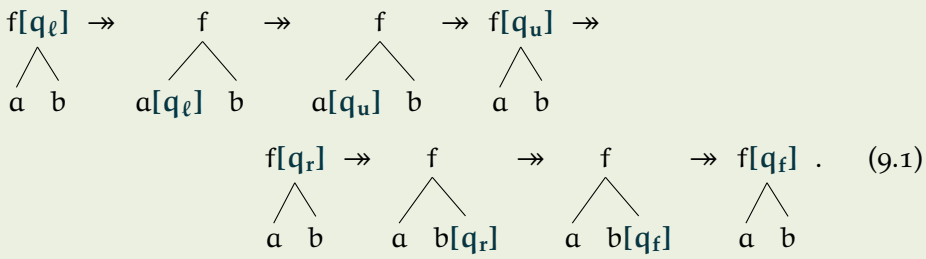
Clearly for all $p \in Q$, (p, p) is a loop; we call such loops *trivial*. A looping run of $\mathcal{A}$ on $t|_\alpha$ is *simple* if it reaches α exactly twice, which is to say that there are two configurations in the run that are at position α. It is *non-trivial* if it reaches α at least twice. A loop is *simple* (resp. *non-trivial*) if there exists a *simple* (resp. *non-trivial*) looping run forming it.

trivial loop

simple loop

non-trivial loop

Example: The loop (q_ℓ, q_u) in the above example is simple, because $(0, q_\ell)$, $(0.0, q_\ell)$, $(0.0, q_u)$, $(0, q_u)$ only reaches $\alpha = 0$ twice, on the first and last configuration. The TWA $\mathcal{X}$ forms only trivial and simple loops, but suppose that we alter and extend it so that it also checks that the *right-most* leaf is b. During an accepting run it would go down and left in q_ℓ, back up to the root in q_u, down and right in q_r, and back up to the root again, in a final state q_f. Thus all accepting runs would be non-trivial and non-simple, reaching – or staying at – the root at least three times, and exactly four if we use the same style of stationary transitions as before:



(9.1)

◇

Fortunately, we only ever need to compute simple loops, as all other loops can be computed from them, thanks to the next lemma. It should be noted that, in this chapter, we depart from the conventions for closures made explicit in section 2.1[p23], which otherwise globally apply in this thesis. More specifically, in the context of

sets of loops, which can be seen as binary relations on Q, reflexive closures are always implicitly taken on Q. Thus, if $L \subseteq Q^2$ is a set of loops, $L^* \supseteq \{(p,p) \mid p \in Q\}$, which does not hold in general for ordinary binary relations. This provides a simple shortcut to include all trivial loops in one fell swoop.

Nitpicking on Relations

There are two ways to define a n-ary relation R on sets $S_1, \dots, S_n$: **(1)** as a subset $R \subseteq \prod_{k=1}^{n} S_k$, or **(2)** as a structure $\langle S_1, \dots, S_n, R' \rangle$, with $R' \subseteq \prod_{k=1}^{n} S_k$. We take the first viewpoint throughout this thesis, except for loops.

Lemma 9.3: Loop Decomposition

If $S \subseteq Q^2$ is the set of all simple loops of $\mathcal{A}$ on a given subtree $u = t|_\alpha$, then the closure S^* is the set of all loops of $\mathcal{A}$ on u.

Proof. Every looping run is either trivial or non-trivial. All trivial loops are in S^* by our conventions regarding the reflexive closure of sets of loops. Furthermore, every non-trivial looping run can easily be decomposed into one or more simple runs. Indeed, any non-trivial looping run ℓ has the following general form:

$$\ell = (\alpha, p^0), \left[(\beta_1^k, s_1^k), \dots, (\beta_{n_k}^k, s_{n_k}^k), (\alpha, p^k)\right]^{k \in [\![1,m]\!]},$$

where $\beta_i^k \lhd \alpha$ for all k, i, and the notation $[x_k]^{k \in [\![1,m]\!]}$ designates the run obtained by concatenating the runs $x_1, \dots, x_m$. This is the composition of m simple looping runs ℓ_k, for $k \in [\![1,m]\!]$, forming the simple loops (p^{k-1}, p^k). The remaining loops are obtained by transitive closure:

$$\left\{(p^{k-1}, p^k) \mid k \in [\![1,m]\!]\right\}^+ = \left\{(p^{k-1}, p^l) \mid k, l \in [\![1,m]\!],\ k \leqslant l\right\}. \qquad \blacksquare$$

Let us denote by $\mho^\tau(u)$ the set of all loops of $\mathcal{A}$ on a subtree u, where τ is the type of the root of u; if u is the subtree $t|_\alpha$ then $\tau = \mathrm{ty}\,\alpha$. Note that thanks to the above-mentioned restriction in the definition of loops, the type of the subtree's root is the only information which is actually needed from the context.

$\mho^\tau(u)$: *loops on* u, *type* τ

Let $a \in \mathbb{A}_0$ be a leaf of type τ. We compute the loops on a. By definition of a looping run, $\mathcal{A}$ cannot move up; nor can it move down since leaves have no children. So the only transitions which can be activated are $\circlearrowleft$-transitions. As we are solely interested in *simple* loops, we can only activate one of these transitions *once*, thus creating runs of the form $(\alpha, p) \twoheadrightarrow (\alpha, q)$, and the corresponding loops (p, q). Let us have a general notation for this:

Definition 9.4: Simple Here-Loops

$$\mathcal{H}_\sigma^\tau = \{(p,q) \mid \langle \sigma, p, \tau \to \circlearrowleft, q\rangle \in \Delta\}.$$

$\mathcal{H}_\sigma^\tau$: *simple here-loops*

Thus the simple loops on a are $\mathcal{H}_a^\tau$. By Lemma 9.3 we have $\mho^\tau(a) = (\mathcal{H}_a^\tau)^*$. We now deal with inner nodes. Let $f \in \mathbb{A}_2$, and $u = f(u_0, u_1)$; again, τ denotes the type of the root of u. Clearly the elements of $\mathcal{H}_f^\tau$ are loops on u, as above, but this time $\mathcal{A}$ can move down as well. It cannot move up on the first move – that would mean leaving the subtree – but it will obviously need to move up to rejoin the root if it ever moves down.

To clarify all that, let us reason on what the first move of a simple looping run can be. It cannot be $\uparrow$ and all simple loops whose first move is $\circlearrowleft$ are already computed in $\mathcal{H}_f^\tau$. Say the first move is $\swarrow$: then the run can do whatever it wants in the left subtree u_0, after which it has to move back up to the root to complete the loop.

Again, we only consider simple loops, so no move can be made past this point, as the root has been reached twice already. Thus the general form of such a run is

$$(\varepsilon, p), (0, p_0), (\beta_1, s_1), \ldots, (\beta_n, s_n), (0, q_0), (\varepsilon, q) \,,$$

with all $\beta_k \trianglelefteq 0$. But by definition, this means that (p_0, q_0) is a loop on u_0, i.e. $(p_0, q_0) \in \mathfrak{O}^0(u_0)$. Needless to say, the same applies (with $\mathbf{1}$ instead of $\mathbf{0}$) if the first move is $\searrow$. It follows that to determine whether (p, q) forms a simple loop on u, we need only check three things:

(1) $\mathcal{A}$ can move down (left or right) from state p into a state p_θ,

(2) there is a loop (p_θ, q_θ) on this subtree, and

(3) in state q_θ, $\mathcal{A}$ can move up from this subtree and into the state q.

Then there only remains to take the transitive and reflexive closure to obtain all loops. Formally, this describes the following computation:

$$\mathfrak{O}^\tau(u) = \left(\mathcal{H}_f^\tau \cup \left\{ (p, q) \;\middle|\; \begin{array}{c} \exists \theta \in \mathbb{S} : \\ \exists (p_\theta, q_\theta) \in \mathfrak{O}^\theta(u_\theta) \end{array} : \begin{array}{c} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \\ \langle u_\theta(\varepsilon), q_\theta, \theta \to \uparrow, q \rangle \in \Delta \end{array} \right\} \right)^* .$$

Theorem 9.5: Loops

Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\mathbb{A})$. Then for all $\alpha \in \mathcal{P}(t)$, $\mathfrak{O}^{\mathrm{ty}\,\alpha}(t|_\alpha)$, as defined above, is the set of all loops of $\mathcal{A}$ on $t|_\alpha$.

Example: For the TWA $\mathcal{X}$, $\mathfrak{O}^0(a) = \{(q_\ell, q_u)\}^* = \{(q_\ell, q_\ell), (q_u, q_u), (q_\ell, q_u)\}$, and $\mathfrak{O}^\star(f(a, b)) = (\varnothing \cup \{(q_\ell, q_u)\})^*$ (no simple here-loop, and one loop built on the left child). On the other hand, $\mathfrak{O}^\star(f(b, a)) = \varnothing^*$, because $\mathfrak{O}^1(a) = \mathfrak{O}^0(b) = \varnothing^*$. ♢

9.2.2 A Direct Application of Loops to Membership Testing

Note that a reasonably efficient algorithm for testing membership is straightforwardly derived from the above computation of loops:

Corollary 9.6: TWA Membership

Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\mathbb{A})$. Then we have $t \in \mathcal{L}(\mathcal{A})$ if and only if $\mathfrak{O}^\star(t) \cap (I \times F) \neq \varnothing$.

Proof. There is a loop $(q_i, q_f) \in I \times F$ of $\mathcal{A}$ on t if and only if there is a run of the form $(\varepsilon, q_i), \ldots, (\varepsilon, q_f)$. The first configuration is initial, and the last is final. Therefore it is an accepting run, and $t \in \mathcal{L}(\mathcal{A})$. ■

Corollary 9.7

The complexity of TWA membership is $O\big(|\Delta| + \|t\| \cdot |Q|^3\big)$.

Proof. A naïve computation of $\mathfrak{O}^\star(t)$ would be done in

$$O\big(\|t\| \cdot (|Q|^3 + |Q|^2 \cdot |\Delta|)\big) .$$

The following algorithm, while still simple, runs in $O\big(|\Delta| + \|t\| \cdot |Q|^3\big)$, at the cost of a $O(\|t\| \cdot |Q|^2)$ space complexity.

Preliminaries. Transitions and loops will be represented by relations from Q to Q, coded as matrices of $\mathcal{M}_{|Q|}(\mathbb{B})$ within the classical boolean algebra $(\mathbb{B}, +, \cdot)$. The states of Q are numbered and assimilated to their indices $[\![1, n]\!]$ for the sake of denotational simplicity. A relation $R \subseteq Q^2$ is represented by the matrix $\mathcal{M}[R] = (\mathcal{M}[R]_{ij})$, such that

$$\mathcal{M}[R]_{ij} = 1 \iff jRi .$$

The sum and product of matrices are defined as usual. With those conventions we have the expected result regarding composition: let $R, R' \subseteq Q^2$ and $P = \mathcal{M}[R'] \times \mathcal{M}[R]$; then

$$P_{ij} = \sum_{k=1}^{n} \mathcal{M}[R']_{ik} \mathcal{M}[R]_{kj} .$$

Thus $P_{ij} = 1$ if and only if there exists k such that jRk and $kR'i$, that is to say, $j(R' \circ R)i$. In other words $\mathcal{M}[R' \circ R] = \mathcal{M}[R'] \times \mathcal{M}[R]$.

Input & Variables. A TWA $\mathcal{A}$ and a tree t form the input. The core of the algorithm is the sub-function f, which takes as input α (a position in $\mathcal{P}(t)$). Its call defines a matrix L^α, representing the loops at position α.

Algorithm.

Initialisation.
For each $\sigma \in \mathbb{A}$, $\tau \in \mathbb{T}$, $\mu \in \mathbb{M}$, a matrix $T^{\sigma,\tau,\mu}$ is built such that $T^{\sigma,\tau,\mu}_{qp} = 1$ if and only if $\langle \sigma, p, \tau \to \mu, q \rangle \in \Delta$. The positions of $\mathcal{P}(t)$ are topologically ordered with respect to the partial order $\trianglelefteq$, resulting in the sequence $\alpha_1, \dots, \alpha_m = \varepsilon$.

Body.
For $k = 1$ to m, $f(\alpha_k)$ is called. Then L^ε is returned. On a call to $f(\alpha)$:

(1) Populate the matrix

$$L^\alpha = T^{t(\alpha),\mathrm{ty}\,\alpha,\cup} + \sum_{\theta \in \mathbb{S}} \left[T^{t(\alpha.\theta),\mathrm{ty}(\alpha.\theta),\uparrow} \times L^{\alpha.\theta} \times T^{t(\alpha),\mathrm{ty}\,\alpha,\chi(\theta)} \right] .$$

(2) Compute the reflexive and transitive closure of L^α in place.

Complexity. The initial topological sorting is done in $O(\|t\|)$, and the construction of the $T^{\sigma,\tau,\mu}$ matrices is done in $O(|\Sigma| \cdot |Q|^2 + |\Delta|)$. Within each call of f we have the following complexities:

(1) $O(|Q|^{2.3727})$ using the latest version of the Coppersmith–Winograd algorithm [Coppersmith & Winograd, 1990; Stothers, 2010; Williams, 2011] – or simply $O(|Q|^3)$ with the conventional product.

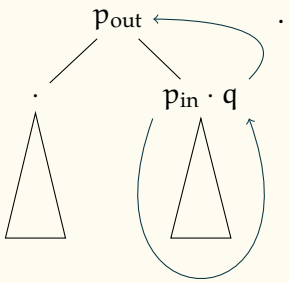
(2) $\Theta(|Q|^3)$ using the Roy–Floyd–Warshall algorithm [Roy, 1959; Warshall, 1962; Floyd, 1962].

The complexity of any call to f is therefore $O(|Q|^3)$; there are $\|t\|$ calls to f. Hence the announced total complexity of $O\big(|\Delta| + \|t\| \cdot |Q|^3\big)$.

Correctness. After the call to $f(\alpha)$, it is plain that L^α encodes $\mathfrak{O}^{\mathrm{ty}\,\alpha}(t|_\alpha)$, as the computation of (1) and (2) is a straight-forward reformulation of the formula of Thm. 9.5[p170] in terms of a boolean matrix representation. The recursive nature of that formula has been unwound in this algorithm by the prior topological sorting of the positions. ■

9.2.3 From Loops to Overloops

We now introduce a new notion related to tree loops: *tree overloops*. An overloop is formed by a looping run followed by a move up; this apparently minor change has a number of positive consequences which we discuss in the next sections. In particular, this notion has a great advantage over loops in the deterministic case. Schematically, an overloop $(p_{\mathrm{in}}, p_{\mathrm{out}})$ based on a loop (p_{in}, q) looks like this:



Of course, this immediately raises the pressing question of what is supposed to happen if the overloop starts – in p_{in} – at the root of the tree. In order for overloops to be defined for all starting positions, we need to make moving up from the root legal.

▽ Definition 9.8: Over-Root, Extended Positions and Transitions

$\overline{\mathcal{P}}(t)$: *extended positions of* t

$\overline{\varepsilon}$: *overroot:* $\varepsilon \lhd \overline{\varepsilon}$

The *extended positions* $\overline{\mathcal{P}}(t)$ of a tree $t \in \mathcal{T}(\mathbb{A})$ are the set $\mathcal{P}(t) \cup \{\overline{\varepsilon}\}$, where $\overline{\varepsilon}$ is called the *overroot*. The parent function $\mathsf{parent}(\cdot)$ is extended over $\overline{\mathcal{P}}(t)$ into the *extended parent function* $\overline{\mathsf{parent}}(\cdot)$, such that $\overline{\mathsf{parent}}(\varepsilon) = \overline{\varepsilon}$ and $\varepsilon \lhd \overline{\varepsilon}$. The notion of configuration is extended as well, so that the transitions of $\langle \mathbb{A}, Q, \star \to \uparrow, Q\rangle$ become valid. Their application yields configurations of the form $(\overline{\varepsilon}, q)$.

▽ Definition 9.9: Tree Over-Loops

tree overloop

Let $\mathcal{A}$ be a TWA and t a tree. A pair of states $(p, q) \in Q^2$ forms an *overloop* of $\mathcal{A}$ on $t|_\alpha$ if there exists a run

$$(\alpha, p), (\beta_1, s_1), \ldots, (\beta_n, s_n), (\overline{\mathsf{parent}}(\alpha), q)$$

such that $\beta_k \trianglelefteq \alpha$ for all $k \in [\![1, n]\!]$.

A way to compute overloops is to compute loops, then check for ↑-transitions:

Definition 9.10: Up-Closure

Let $L \subseteq Q^2, \tau \in \mathbb{T}$ and $\sigma \in \mathbb{A}$:

$$\mathcal{U}^\tau_\sigma[L] = \left\{ (p,q) \in Q^2 \;\middle|\; \exists p' \in Q : (p,p') \in L \text{ and } \langle \sigma, p', \tau \to \uparrow, q\rangle \in \Delta \right\}.$$

$\mathcal{U}^\tau_\sigma[L]$: *up-closure of loops* L

Lemma 9.11: Up-Closure

Let $\mathcal{A}$ be a TWA. If L is the set of all loops of $\mathcal{A}$ on a subtree $u = t|_\alpha$, then $\mathcal{U}^{\mathrm{ty}\,\alpha}_{t(\alpha)}[L]$ is the set of all overloops of $\mathcal{A}$ on u.

Proof. Immediate from Def. 9.9, as we have necessarily $\beta_n = \alpha$. Thus any overloop is a loop followed by a move up, and conversely. ∎

Similarly to loops, we denote by $\overset{\frown}{\mathfrak{O}}^\tau(u)$ the set of all overloops of $\mathcal{A}$ on a subtree u, where τ is the type of the root of u. By Lem. 9.11 we have $\overset{\frown}{\mathfrak{O}}^\tau(u) = \mathcal{U}^\tau_{u(\varepsilon)}[\mathfrak{O}^\tau(u)]$, and in the case of leaves this yields $\overset{\frown}{\mathfrak{O}}^\tau(a) = \mathcal{U}^\tau_a[(\mathcal{H}^\tau_a)^*]$. However, in the case of inner nodes – e.g. $u = f(u_0, u_1)$ – in order to have an inductive computation of overloops instead of one based on loops, we need to compute the overloops of the father, knowing the overloops of the children. The simplest way is to compute the loops of the father and take the up-closure. We start by computing the simple loops, for which one only needs to check whether

$\overset{\frown}{\mathfrak{O}}^\tau(u)$: *overloops on* u, *type* τ

(1) the automaton can go down and left (resp. right) from p to a state p_θ, and

(2) there is a left (resp. right) overloop (p_θ, q_θ): this forms a loop (p, q_θ).

The reflexive and transitive closure yields all the loops, and then the up-closure yields all overloops. Formally, the above describes the computation:

$$\overset{\frown}{\mathfrak{O}}^\tau(u) = \mathcal{U}^\tau_f\left[\left(\mathcal{H}^\tau_f \cup \left\{ (p, q_\theta) \;\middle|\; \begin{array}{l} \exists \theta \in \mathbb{S} : \\ \exists p_\theta \in Q \end{array} \text{ st. } \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta\rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in \overset{\frown}{\mathfrak{O}}^\theta(u_\theta) \end{array} \right\}\right)^*\right].$$

Theorem 9.12: Overloops

Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\mathbb{A})$. Then for all $\alpha \in \mathcal{P}(t)$, $\overset{\frown}{\mathfrak{O}}^{\mathrm{ty}\,\alpha}(t|_\alpha)$, as defined above, is the set of all overloops of $\mathcal{A}$ on $t|_\alpha$.

Example: For the TWA $\mathcal{X}$, $\overset{\frown}{\mathfrak{O}}^o(a) = \mathcal{U}^o_a[\mathfrak{O}^o(a)] = \{(q_u, q_u), (q_\ell, q_u)\}$. However $\mathfrak{O}^\star(f(a,b))$ is the empty set. Thus a small adjustment is needed to test membership using overloops, as standard TWA – such as $\mathcal{X}$ – never admit any overloop at the root of a tree, for the lack of ↑-transitions. ♢

Definition 9.13: Overfinal State & Escaped TWA

Let $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta\rangle$ be a TWA; it can be transformed into an *escaped TWA*

$$\mathcal{A}' = \left\langle\, \mathbb{A},\ Q \uplus \{\checkmark\},\ I, F,\ \Delta \uplus \langle \mathbb{A}, F, \star \to \uparrow, \checkmark\rangle \,\right\rangle,$$

✓: *overfinal state*

escaped TWA

where $\checkmark \notin Q$ is a fresh state, called *overfinal state*. Clearly $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Example: Once $\mathcal{X}$ is escaped, we have $\overset{\star}{\mathfrak{O}}(f(a,b)) = \{(q_u, \checkmark), (q_\ell, \checkmark)\}$. ♢

Corollary 9.14: TWA Membership Redux

Let $\mathcal{A}$ be an escaped TWA and $t \in \mathcal{T}(\mathbb{A})$. Then $t \in \mathcal{L}(\mathcal{A})$ if and only if $\overset{\star}{\mathfrak{O}}(t) \cap (I \times \{\checkmark\}) \neq \varnothing$.

Proof. The couple $(q_i, \checkmark) \in I \times \{\checkmark\}$ is an overloop if and only if there is a run $(\varepsilon, q_i), \ldots, (\varepsilon, q_f), (\overline{\varepsilon}, \checkmark)$. By Def. 9.13, we must have $q_f \in F$; therefore, by Cor. 9.6 we have immediately $t \in \mathcal{L}(\mathcal{A})$. ∎

9.3 Transforming TWA into equivalent BUTA

Data: A TWA $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta \rangle$
Input: $\langle P_{\text{init}} \rangle$, $\langle P_0 \rangle$, $\langle P_1 \rangle$, $\langle P_{\text{indu}} \rangle$, $\langle F \rangle$
Result: A BUTA $\mathcal{B}$

initialise States and Rules to $\varnothing$
foreach $a \in \mathbb{A}_0, \tau \in \mathbb{T}$ **do**
A add $a \to \langle P_{\text{init}} \rangle$ to Rules and $\langle P_{\text{init}} \rangle$ to States
repeat
  foreach $f \in \mathbb{A}_2, \tau \in \mathbb{T}$ **do**
B add every $f(\langle P_0 \rangle, \langle P_1 \rangle) \to \langle P_{\text{indu}} \rangle$ to Rules and $\langle P_{\text{indu}} \rangle$ to States
    where $\langle P_0 \rangle, \langle P_1 \rangle \in$ States
until Rules *remains unchanged*
return $\mathcal{B} = \langle \mathbb{A}, \text{States}, \langle F \rangle, \text{Rules} \rangle$

Algorithm 1: Meta-Transformation into BUTA

Data: A TWA $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta \rangle$
Result: A BUTA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$

Meta-Algorithm 1 **where**

$$\langle P_{\text{init}} \rangle \equiv (a, \tau, \mathcal{H}_a^{\tau *}) \qquad \langle P_{\text{indu}} \rangle \equiv (f, \tau, (\mathcal{H}_f^{\tau} \cup S)^*)$$
$$\langle P_0 \rangle \equiv (\sigma_0, 0, S_0) \qquad \langle P_1 \rangle \equiv (\sigma_1, 1, S_1)$$
$$\langle F \rangle \equiv \{(\sigma, \star, L) \in \text{States} \mid L \cap (I \times F) \neq \varnothing\}$$

$$S = \left\{ (p,q) \;\middle|\; \exists \theta \in \mathbb{S}, (p_\theta, q_\theta) \in S_\theta : \middle| \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \text{ and} \\ \langle \sigma_\theta, q_\theta, \theta \to \uparrow, q \rangle \in \Delta \end{array} \right\}$$

Algorithm 2: Transformation into BUTA, with loops

It is well-known that every TWA is equivalent to a BUTA; a more general version of this result has been proven in [Cosmadakis, Gaifman, Kanellakis & Vardi, 1988] –

Data: An escaped TWA $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta \rangle$ (see Def. 9.13)
Result: A BUTA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$

Meta-Algorithm 1 **where**

$$\langle P_{\text{init}} \rangle \equiv (\tau, \mathcal{U}_a^\tau[\mathcal{H}_a^{\tau *}]) \qquad \langle P_{\text{indu}} \rangle \equiv (\tau, \mathcal{U}_f^\tau[(\mathcal{H}_f^\tau \cup S)^*])$$
$$\langle P_0 \rangle \equiv (\sigma_0, S_0) \qquad \langle P_1 \rangle \equiv (\sigma_1, S_1)$$
$$\langle F \rangle \equiv \{ (\star, O) \in \mathsf{States} \mid O \cap (I \times \{\checkmark\}) \neq \varnothing \}$$

$$S = \left\{ (p, q_\theta) \;\middle|\; \exists \theta \in \mathbb{S}, p_\theta \in Q : \;\middle|\; \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in S_\theta \end{array} \right\}$$

Algorithm 3: Transformation into BUTA, with overloops

using game-theoretic arguments – and the main idea of a loop-based transformation from TWA into BUTA is outlined in [Bojańczyk, 2008] and [Samuelides, 2007, p143].

In this section we present two versions of it: the classical, loop-based construction is presented as Algo. 2, and an overloop-based variant is described in Algo. 3. Since those algorithms share a strong common structure, they are given as instantiations of Meta-Algorithm 1, whose inputs – between angle brackets $\langle \cdot \rangle$ – are syntactically substituted into its body. We go on to show that, in the case of deterministic TWA, the overloop-based construction results in much smaller equivalent BUTA than the classical one.

9.3.1 Two Variants: Loops and Overloops

Lemma 9.15: Loop-Based Algorithm

Let $\mathcal{A}$ be a TWA, $\mathcal{B}$ the BUTA constructed by Algorithm 2, $t \in \mathcal{T}(\mathbb{A})$ and a position $\alpha \in \mathcal{P}(t)$. Then for every type $\tau \in \mathbb{T}$ there is a unique run ρ of $\mathcal{B}$ on $t|_\alpha$, which is such that $\rho(\varepsilon) = (t(\alpha), \tau, \mho^\tau(t|_\alpha))$.

Proof. By structural induction on $u = t|_\alpha$.

Base Case: $u = a \in \mathbb{A}_0$. By line A in Algorithm 2, $\rho(\varepsilon) = P = (a, \tau, \mathcal{H}_a^{\tau *}) = (t(\alpha), \tau, \mathcal{H}_a^{\tau *})$. This is the only possible run, as only one transition $a \to P$ is generated for each couple a, τ. By Theorem 9.5 we have $\mathcal{H}_a^{\tau *} = \mho^\tau(a)$.

Inductive Case: $u = f(u_0, u_1)$. By induction hypothesis the run ρ_0 on u_0 is such that $\rho_0(\varepsilon) = P_0 = (u_0(\varepsilon), 0, \mho^0(u_0))$, and the run ρ_1 on u_1 is such that $\rho_1(\varepsilon) = P_1 = (u_1(\varepsilon), 1, \mho^1(u_1))$. By line B in Algo. 2 we use the rule $f(P_0, P_1) \to P$ to build a run ρ such that $\rho(\varepsilon) = P = (f, \tau, (\mathcal{H}_f^\tau \cup S)^*) = (u(\varepsilon), \tau, (\mathcal{H}_f^\tau \cup S)^*)$, $\rho|_0 = \rho_0$ and $\rho|_1 = \rho_1$. Since ρ_0 and ρ_1 are unique, so is ρ. By Theorem 9.5, $(\mathcal{H}_f^\tau \cup S)^* = \mho^\tau(u)$. ∎

Theorem 9.16

Algorithm 2 is correct; that is, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Proof. The following statements are equivalent by Lem. 9.15 and Cor. 9.6:

(1) $t \in \mathcal{L}(\mathcal{A})$.
(2) there is a loop $(q_i, q_f) \in I \times F$ of $\mathcal{A}$ on t.
(3) the run ρ of $\mathcal{B}$ on t is such that $\rho(\varepsilon) = (t(\varepsilon), \star, \mathfrak{O}^\star(t))$, with $(q_i, q_f) \in \mathfrak{O}^\star(t)$.
(4) $\rho(\varepsilon)$ is a final state for $\mathcal{B}$.
(5) $t \in \mathcal{L}(\mathcal{B})$. ∎

Two short but important remarks are in order.

(1) It might seem strange that our states are in $\mathbb{A} \times \mathbb{T} \times 2^{Q^2}$, and not more simply in $\mathbb{T} \times 2^{Q^2}$, as suggested in [Samuelides, 2007]. In [Bojańczyk, 2008] a similar construction – albeit deterministic, see the second remark – is proposed, which does not include $\mathbb{A}$ either. However, it is not clear how loops could be considered independently from the root symbol of the subtree that bears them. Consider for instance $a, b \in \mathbb{A}_0$ with only the transitions $\langle \{a, b\}, p, \tau \to \circlearrowleft, q \rangle$ and $\langle b, q, \tau \to \uparrow, s' \rangle \in \Delta$. Then the loops on a and b are exactly the same – $\{(p, q)\}^*$ – and yet, from their father's point of view, they behave very differently. If $\mathcal{A}$ can go down from a state s to p, it can form a loop (s, s') if the child is b, but not if it is a. In contrast to the loop-based construction, the overloop-based algorithm – Algo. 3 – suppresses this problem completely.

(2) The observation made in Lemma 9.15 that the run of $\mathcal{B}$ is unique, given a subtree and a type, makes it easy to adapt the algorithm to yield a deterministic BUTA. Indeed, every tree in $\mathcal{T}(\mathbb{A})$ is non-deterministically evaluated by $\mathcal{B}$ into one of exactly three possible states, each corresponding to a type; the correct one is chosen according to the context during the run. Recall that rules $f(P_0, P_1) \to P$ are built such that the "type" component of P_θ is θ, and final states bear the root type $\star$. Hence, it suffices to group those three possible states into one element of $\mathbb{A} \times (2^{Q^2})^{|\mathbb{T}|}$ to achieve determinism which brings us back to the states suggested in [Bojańczyk, 2008]. Of course, if one does that, there are a number of optimisations which can be performed. For instance, since the star-component is only ever useful at the root, it suffices to replace it with a boolean indicating whether it contains a loop in $I \times F$, i.e. whether it is a final state. Then we get states in $\mathbb{A} \times (2^{Q^2})^{|\mathbb{S}|} \times \{0, 1\}$.

∀ Lemma 9.17: Overloop-Based Algorithm

Let $\mathcal{A}$ be a TWA, $\mathcal{B}$ the BUTA constructed by Algorithm 3, $t \in \mathcal{T}(\Sigma)$ and a position $\alpha \in \mathcal{P}(t)$. Then for every type $\tau \in \mathbb{T}$ there is a unique run ρ of $\mathcal{B}$ on $t|_\alpha$, which is such that $\rho(\varepsilon) = (\tau, \overset{\frown}{\mathfrak{O}}^\tau(t|_\alpha))$.

Proof. See proof of Lemma 9.15. The only change is that this time, we build the loops, then deduce the overloops from them (Lem. 9.11[p173], Thm. 9.12). ∎

△ Theorem 9.18

Algorithm 3 is correct; that is, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Proof. By construction $(i, \checkmark) \in I \times \{\checkmark\}$ is an overloop if and only if there exists $f \in F$ such that (i, f) is a loop. Same proof as Theorem 9.16. ∎

Note that this construction can be adapted to yield deterministic BUTA in exactly the same way as for Algo. 2.

9.3.2 Overloops: Deterministic Size Upper-Bound

Definition 9.19: Deterministic TWA

A TWA $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta \rangle$ is *deterministic* – ie. is a *DTWA* – if

$$|\langle \sigma, p, \tau \to \mathbb{M}, Q \rangle \cap \Delta| \leqslant 1 \ ,$$

for all $\sigma \in \mathbb{A}$, $p \in Q$, $\tau \in \mathbb{T}$.

For our purposes, we do not need to add to that the usual condition that I must be a singleton.

Example: The running example TWA $\mathcal{X}$ happens to be a deterministic tree-walking automaton. ♢

Let us be reminded that a relation $R \subseteq Q^2$ is *functional* (or *right-unique*, or *a partial function*) if, for all $p, q, q' \in Q$, pRq and $pRq' \implies q = q'$.

This was previously defined in section 2.1[p23].

Remark 9.20

There are $2^{|Q|^2}$ binary relations on Q, of which $|Q+1|^{|Q|}$ are partial functions, of which $|Q|^{|Q|}$ are total functions.

Remark 9.21

If a relation R is functional, then so is R^k, for any $k \in \mathbb{N}$.

By construction, a BUTA built by Algo. 2 (loop-based) has at most $|\mathbb{A}| \cdot |\mathbb{T}| \cdot 2^{|Q|^2}$ states, while one built by Algo. 3 (overloop-based) has at most $|\mathbb{T}| \cdot 2^{|Q|^2}$. We shall see in this section that, in the deterministic case, this upper bound is in fact much lower for the overloop-based algorithm than for the traditional loop-based one. More specifically, we show that the following holds:

Theorem 9.22: Deterministic Upper-Bound

Let $\mathcal{A}$ be a deterministic TWA and $\mathcal{B}$ its equivalent BUTA built by an application of Algorithm 3. Then $\mathcal{B}$ has at most $|\mathbb{T}| \cdot 2^{|Q| \log_2(|Q|+1)}$ states.

The idea is that every state which we build corresponds exactly to the set L of all loops (resp. overloops) of the automaton $\mathcal{A}$ on a certain subtree u. Since $L \subseteq Q^2$, we can see it as a binary relation on the states. The intuition here is that, if $\mathcal{A}$ is deterministic, and enters the root of u in one given state p, then there "should be" only one possible outcome. More formally:

∀ Lemma 9.23

If $\mathcal{A}$ is a deterministic TWA, then $\twoheadrightarrow_{\mathcal{A}}$ is functional.

Proof. In a given configuration (α, p), over a tree t, $|\langle t(\alpha), p, \mathsf{ty}\,\alpha \to \mathbb{M}, Q\rangle \cap \Delta| \leqslant 1$. Therefore, (α, p) has at most one successor. ∎

However, in the case of loops, this does not suffice to make L functional because, determinism notwithstanding, a single (non-trivial) loop may reach the root several times, and in different states, before exiting the subtree. Indeed, we have seen such a behaviour above, in the run $(9.1)_{[p168]}$. Thus there is nothing to prevent us from having both pLq and pLq', for $q \neq q'$; we show next that in that case, one of these loops is simply an extension of the other.

∀ Lemma 9.24: Hidden Loops

Let (p, q) and (p, q') be loops of the TWA $\mathcal{A}$ on a given subtree $t|_\alpha$, such that $q \neq q'$. Then if $\mathcal{A}$ is deterministic, either (q, q') or (q', q) must be a loop of $\mathcal{A}$ on $t|_\alpha$.

Proof. By Definition 9.1, there exist two runs $c_0, \dots, c_n$ and $d_0, \dots, d_m$ such that $c_0 = d_0 = (\alpha, p)$, $c_n = (\alpha, q)$ and $d_m = (\alpha, q')$. If $n = m$ then $c_0 \twoheadrightarrow^n c_n$ and $c_0 \twoheadrightarrow^n d_n$ and by Lemma 9.23 and Remark 9.21, it follows that $c_n = d_m$. But this contradicts $q \neq q'$, so we must have $n \neq m$. Say that $n < m$. Then $c_n = d_n$, and $(\alpha, q) = d_n, \dots, d_m = (\alpha, q')$ forms a run. Therefore (q, q') is a loop. Similarly, if $n > m$, then by the same arguments (q', q) is a loop. ∎

Contrariwise, two overloops cannot be combined to form another overloop on the same subtree, which satisfies the above intuition of a "single outcome":

∀ Lemma 9.25

Let $p, q, q' \in Q$, such that (p, q) and (p, q') are overloops of the TWA $\mathcal{A}$ on a given subtree $t|_\alpha$. Then if $\mathcal{A}$ is deterministic, $q = q'$.

Proof. By Def. 9.9, there exist $s, s' \in Q$ such that $(\alpha, p), \dots, (\alpha, s), (\overline{\mathsf{parent}}(\alpha), q)$ and $(\alpha, p), \dots, (\alpha, s'), (\overline{\mathsf{parent}}(\alpha), q')$ are runs; thus (p, s) and (p, s') are loops. If $s \neq s'$, then by Lem. 9.24, say, (s, s'), is a loop. So there exist $s_1, \dots, s_n \in Q, \beta_1 \trianglelefteq \alpha, \dots, \beta_n \trianglelefteq \alpha$ such that $(\alpha, s), (\beta_1, s_1), \dots, (\beta_n, s_n), (\alpha, s')$ is a run. Thus we have in particular $(\alpha, s) \twoheadrightarrow (\overline{\mathsf{parent}}(\alpha), q)$ and $(\alpha, s) \twoheadrightarrow (\beta_1, s_1)$. It follows that $\overline{\mathsf{parent}}(\alpha) = \beta_1 \trianglelefteq \alpha$, which is contradictory. Hence $s = s'$. We have both $(\alpha, s) \twoheadrightarrow (\overline{\mathsf{parent}}(\alpha), q)$ and $(\alpha, s) \twoheadrightarrow (\overline{\mathsf{parent}}(\alpha), q')$. Since $\twoheadrightarrow$ is functional (Lem. 9.23), we have finally $q = q'$. ∎

With this, we can conclude the proof of Theorem 9.22.

Proof of Theorem 9.22. By construction, for every state $P = (\tau, L)$ generated for $\mathcal{B}$ by Algorithm 3, there exists at least a subtree t such that L is the set of overloops of $\mathcal{A}$ on t. Thus, by Lemma 9.25, L is functional. Therefore, by Remark 9.20, there are at most $|\mathbb{T}| \cdot |Q + 1|^{|Q|}$ states – or, equivalently, $|\mathbb{T}| \cdot 2^{|Q| \log_2(|Q|+1)}$. ∎

Note that the same bound – with a $|\mathbb{A}|$ factor – might be achievable using loops, if special provisions are made to determine which of the two loops (p, q) and (p, q') subsumes the other, and to remove the superfluous loops from the states as they are built. However, such provisions would be invalid if $\mathcal{A}$ is not deterministic, unlike the overloops method, which is applicable in all generality.

9.4 A Polynomial Over-Approximation for Emptiness

Data: An escaped TWA $\mathcal{A} = \langle \mathbb{A}, Q, I, F, \Delta \rangle$ (see Def. 9.13)
Result: Empty (only if $\mathcal{L}(\mathcal{A}) = \varnothing$) or Unknown

initialise $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_\star$ to $\varnothing$; **foreach** $a \in \mathbb{A}_0, \tau \in \mathbb{T}$ **do** $\mathcal{L}_\tau \leftarrow \mathcal{L}_\tau \cup \mathcal{U}^\tau_a[\mathcal{H}^{\tau\,*}_a]$
repeat
  foreach $f \in \mathbb{A}_2, \tau \in \mathbb{T}$ **do** $\mathcal{L}_\tau \leftarrow \mathcal{L}_\tau \cup \mathcal{U}^\tau_f\left[(\mathcal{H}^\tau_f \cup S)^*\right]$
  where $S = \left\{ (p, q_\theta) \;\middle|\; \exists \theta \in \mathbb{S}, p_\theta \in Q : \middle| \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in \mathcal{L}_\theta \end{array} \right\}$
until $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_\star$ *remain unchanged*
return Empty *if* $\mathcal{L}_\star \cap (I \times \{\checkmark\}) = \varnothing$, *else* Unknown

Algorithm 4: Approximation for emptiness, with overloops

Testing emptiness of a TWA $\mathcal{A}$ is an ExpTime-complete problem [Bojańczyk, 2008]. This is rather unfortunate, as there are practical questions – as sketched in the previous chapter – which reduce to the emptiness of the language of a TWA, or of closely related variants in the tree-walking family. We present in this section a crude but astonishingly accurate and very expeditious overloops-based algorithm capable of detecting emptiness in a number of cases. Algorithm 4 is a variant of Algorithm 3 with the following properties:

∀ Lemma 9.26: Overloops Over-Approximation

Let $\mathcal{A}$ be a TWA; when the execution of Algorithm 4 ends, then for any $\tau \in \mathbb{T}$,

$$\mathcal{L}_\tau \supseteq \bigcup_{t \in \mathcal{T}(\Sigma)} \overset{\frown}{\mathfrak{O}}^\tau(t)\,.$$

Proof. This result is fairly clear when comparing Algorithms 3 and 4. Let us consider a tree t and a subtree $u = t|_\alpha$, with $\tau = \mathrm{ty}\,\alpha$. We show that $\overset{\frown}{\mathfrak{O}}^\tau(u) \subseteq \mathcal{L}_\tau$.

Base case:. $u = a \in \mathbb{A}_0$. Then by the first line of Algo. 4, we have $\overset{\frown}{\mathfrak{O}}^\tau(a) = \mathcal{U}^\tau_a[\mathcal{H}^{\tau\,*}_a] \subseteq \mathcal{L}_\tau$.

Inductive case:. If $u = f(u_0, u_1), f \in \mathbb{A}_2$, then by induction hypothesis we have $\overset{\frown}{\mathfrak{O}}^0(u_0) \subseteq \mathcal{L}_0$ and $\overset{\frown}{\mathfrak{O}}^1(u_1) \subseteq \mathcal{L}_1$. The expression computed in the main loop is almost the same as that of Thm. 9.12 for $\overset{\frown}{\mathfrak{O}}^\tau(u)$, the only difference being that $\mathcal{L}_\theta$ is used instead of $\overset{\frown}{\mathfrak{O}}^\theta(u_\theta)$. Since we have $\overset{\frown}{\mathfrak{O}}^\theta(u_\theta) \subseteq \mathcal{L}_\theta$ for all $\theta \in \mathbb{S}$, the expression in Algo. 4 computes *at least* all overloops of $\overset{\frown}{\mathfrak{O}}^\tau(u)$ — and adds them to $\mathcal{L}_\tau$. Thus $\overset{\frown}{\mathfrak{O}}^\tau(u) \subseteq \mathcal{L}_\tau$. ∎

Theorem 9.27

Algorithm 4 is correct; that is, it yields Empty only if $\mathcal{L}(\mathcal{A}) = \varnothing$.

Proof. Suppose that Algo. 4 yields Empty. By definition, this is the case if and only if $\mathcal{L}_\star \cap (I \times \{\checkmark\}) = \varnothing$. By Lemma 9.26, we have $\bigcup_{t \in \mathcal{T}(\Sigma)} \overset{\Uparrow}{\mathfrak{O}}^\tau(t) \subseteq \mathcal{L}_\tau$ for all types τ, and it follows that in particular

$$\bigcup_{t \in \mathcal{T}(\Sigma)} \overset{\Uparrow}{\mathfrak{O}}^\star(t) \ \cap \ (I \times \{\checkmark\}) = \varnothing \, .$$

This can be equivalently rephrased as $\forall t \in \mathcal{T}(\Sigma)$, $\overset{\Uparrow}{\mathfrak{O}}^\star(t) \cap (I \times \{\checkmark\}) = \varnothing$. By Corollary 9.14, this translates into: for all $t \in \mathcal{T}(\Sigma), t \notin \mathcal{L}(\mathcal{A})$, that is to say, $\mathcal{L}(\mathcal{A}) = \varnothing$. ∎

Corollary 9.28: Complexity of the Approximation

The execution of Algorithm 4 is done in time polynomial in the size of $\mathcal{A}$ — more precisely: $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^5)$.

Proof. For all types τ, all operations in Algo. 4 which alter $\mathcal{L}_\tau$ add elements to it. The first loop executes a fixed number of times: $|\Sigma_0| \times |\mathbb{T}|$. The main loop contains only an inner loop which executes a fixed number of times as well – $|\Sigma_2| \times |\mathbb{T}|$ – and the main loop itself executes until no element is added to $\mathcal{L}_\mathbf{0}$, $\mathcal{L}_\mathbf{1}$ or $\mathcal{L}_\star$ during the iteration. Since an iteration can only add elements, and each iteration adds at least one, there can be at most

$$\sum_{\tau \in \mathbb{T}} |\mathcal{L}_\tau| = \sum_{\tau \in \mathbb{T}} |Q|^2 = |\mathbb{T}| \times |Q|^2$$

iterations of the main loop. Each iteration of both the first loop and the main inner loop computes a set of overloops, based on two sets of previously-computed (potential) overloops. This operation executes in a time which is bound as $O(|Q|^2 \cdot |\Delta|)$ for the initial computation and $O(|Q|^3)$ for the computation of the transitive closure. It is executed in total

$$|\Sigma_0| \cdot |\mathbb{T}| + |\mathbb{T}| \cdot |Q|^2 \cdot (|\Sigma_2| \cdot |\mathbb{T}|)$$

times. Overall, the number of executions is in $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^2)$. Globally, the execution time of Algo. 4 is in $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^5)$. ∎

This is of course a *very* loose bound, which could be improved drastically; the important point is that it is in PTIME. Note that Algorithm 4 can easily be made just as coarse or as fine as the need dictates. At the coarse end of that gamut we have a variant of Algorithm 4 which forgoes type information, thus hoarding up all overloops in a single set $\mathcal{L}$ instead of three, and at the fine end we find something equivalent to Algorithm 3.

9.5 Experimental Results

As always when confronted to an approximation, one must take care that it is good enough for practical use; an algorithm may answer "Unknown" systematically, and still be a *very* efficient approximation stricto sensu, but that does not make it interesting. The final arbiter of whether an approximation is of any use is of course how it performs the "real world", which is hard to formalise a priori. However, random tests can serve to sort the wheat from the chaff, especially if the test cases are generated according to a precise distribution.

In this section, we present experimental results for an ad-hoc generation scheme – very briefly – and for a uniform scheme over deterministic TWA.

9.5.1 Evaluating the Approximation's Effectiveness

Tests have been conducted against two different sets of randomly generated TWA. The first set comprised roughly twenty thousand random automata of various sizes – $2 \leqslant |Q| \leqslant 20$ – with a small number of rules – $|\Delta| \approx 3 \times |Q|$ – and the same alphabet as for our running example $\mathcal{X}$. The random generation scheme which produced them was ad hoc and did not have any pertinent statistical grounds. The approximation yielded astonishingly good results on this set: about 75% of the automata had empty languages, yet the approximation failed to detect emptiness in only two cases.

The gritty details of the first, ad hoc generation scheme are available in the source code of our testing tool; cf. the end of the section. More specifically, in `twa.ml`, module `Gene`, functions `make` and `gene`.

To confirm those encouraging results, we generated a second set of – complete and deterministic[(a)] – TWA, this time according to a uniform probability distribution [Héam et al., 2009]. The `REGAL` library [Bassino, David & Nicaud, 2007] was used as back-end to generate the underlying finite-state automata. More specifically, 2 000 TWA were uniformly generated for each $|Q|$ within the range $2 \leqslant |Q| \leqslant 25$. Figure 9.1 summarises the performance of the approximation on this set. The first curve presents the percentage of TWA whose language is detected to be empty by the approximation among the whole 2 000 TWA, for each $|Q|$. The second curve presents the same results, but only for the first 200 TWA[(b)] for each $|Q| \leqslant 10$; the third curve presents the *exact* results for the same data as the second. It is visible that the approximation performs very well again, as the second and third curves are almost indistinguishable.

Out of the 1 724 TWA for which both the approximation and an exact algorithm were run, of which 398 had empty languages, only four failures of the approximation were observed. Furthermore, the first curve shows that the approximation continues to catch cases of emptiness even for sizes completely intractable with exact algorithms.

(a) Note that $|Q|$ is therefore proportional to the size of the generated TWA.

(b) With the exception of the last data point (namely, $|Q| = 10$), for which only the first 124 TWA were tested; this is due to both time constraints and memory limitations of the computer used for the exact tests. The idea is of course to compare comparable things, and to show the exact and approximate methods competing on the same automata. The exact method is obviously the bottleneck in such an experiment.

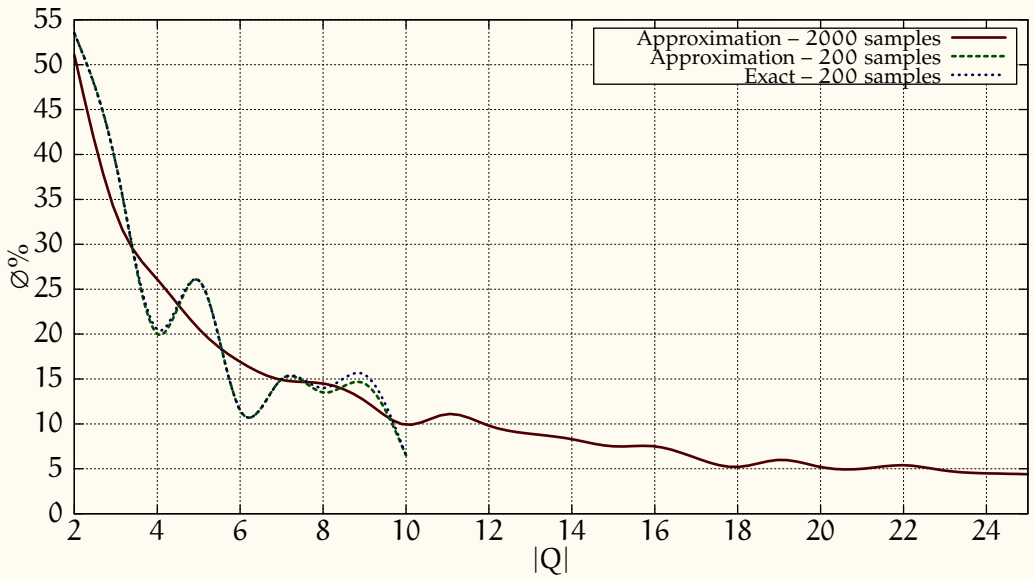


Figure 9.1: Uniform random TWA: emptiness tests.

Those results, though statistically sound, are probably much better than what can be expected in practical applications; it is likely that random instances are in some sense trivial wrt. emptiness. In the absence of substantial testbeds from real-world applications of TWA, a study similar to [Héam, Hugot & Kouchnarenko, 2010a] could be conducted to flesh out the properties which would make an instance "difficult" wrt. emptiness.

9.5.2 Overloops Yield Smaller BUTA

Comparing the output of Algos. 2 & 3, we noted that the latter generates smaller automata. By way of example, if $\mathcal{B}_l$ is the equivalent BUTA obtained from $\mathcal{X}$ by Algo. 2, and $\mathcal{B}_o$ by Algo. 3, then we have $\|\mathcal{B}_l\| = 1986$ and $\|\mathcal{B}_o\| = 95$, where the size of a BUTA $\mathcal{B} = \langle \mathbb{A}, Q, F, \Delta \rangle$ is defined – in the usual way, as seen in section 2.6[p37] and in [Comon et al., 2008] – as:

$$\|\mathcal{B}\| = |Q| + \sum_{f(p_1,\ldots,p_n)\to q\in\Delta} (n+2) \,.$$

Note that the resulting automata are quite large, even for such a trivial TWA as $\mathcal{X}$! For comparison, consider the manually constructed (deterministic) minimal BUTA $\mathcal{B}_m$, and the[(c)] smallest possible non-deterministic BUTA $\mathcal{B}_s$ equivalent to the TWA $\mathcal{X}$: we have $\|\mathcal{B}_m\| = 56$ and $\|\mathcal{B}_s\| = 34$. In other words, the overloop and loop-based constructions are about three and sixty times larger than the optimal, respectively.

More important than the size of the final BUTA is the computation time; it just happens in practice to be roughly proportional to the size of the result, as far as our two transformations are concerned. Using a deterministic variant of either transformation and minimising the result would yield $\mathcal{B}_m$, but at the cost of a considerable increase of the worst-case complexity and average computation time.

Another important point is that the huge size discrepancy between $\mathcal{B}_l$ and $\mathcal{B}_o$ cannot be reduced "in post-processing" using the standard BUTA reduction, that is

[(c)] It happens to be unique (up to homomorphism) in this particular case.

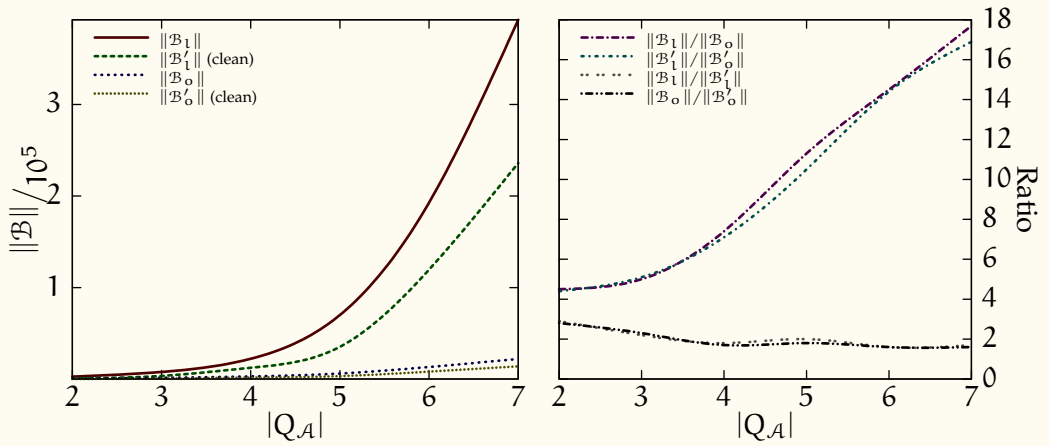


Figure 9.2: Uniform random TWA: size results.

to say the elimination of unreachable states [Comon et al., 2008]: it would have no effect whatsoever, because Algorithms 2 and 3 yield reduced BUTA by construction. A more powerful operation such as the `cleanup` method described in [Héam, Hugot & Kouchnarenko, 2010a] for TAGE, that, among other things, removes states which are not co-accessible as well as unreachable states, can bring down the sizes of $\mathcal{B}_l$ and $\mathcal{B}_o$, but does in no way bridge the gap between them. Case in point, the automata after cleanup $\mathcal{B}'_l$ and $\mathcal{B}'_o$ are of sizes[(d)] $\|\mathcal{B}'_l\| = 1617$ and $\|\mathcal{B}'_o\| = 78$, which yield the following ratios:

$$\frac{\|\mathcal{B}_l\|}{\|\mathcal{B}_o\|} \approx 20.9 \quad \text{and} \quad \frac{\|\mathcal{B}'_l\|}{\|\mathcal{B}'_o\|} \approx 20.7 \quad \text{and} \quad \frac{\|\mathcal{B}_l\|}{\|\mathcal{B}'_l\|} \approx \frac{\|\mathcal{B}_o\|}{\|\mathcal{B}'_o\|} \approx 1.2 \,.$$

These figures suggest that the – substantial – size gains originating from the switch from loops to overloops-based algorithms are completely unrelated to, and do not interfere with, the relatively modest size gains from post-processing. The observations drawn from this single example have been substantiated by more thorough experiments conducted on the same uniformly generated random TWA as in Fig. 9.1, the results of which are summarised in Fig. 9.2. The legend uses the same notations as above. Two hundred TWA have been used to construct each data point.[(e)]

9.5.3 Demonstration Software

Readers interested in experimenting with this chapter's algorithms will find online a proof of concept, meaning both executable binaries – at least for Linux and Windows – and, for souls undaunted by the prospect of confronting twisty, user-unfriendly code, the complete `OCaml` source. The dedicated web page also provides comprehensive instructions for using the executables.

`http://lifc.univ-fcomte.fr/~vhugot/TWA`

[(d)] In this trivial example, the sizes of the TWA $\mathcal{X}$, of the "optimal" equivalent BUTA $\mathcal{B}_m$ and $\mathcal{B}_s$, and of the post-cleanup overloops-based BUTA $\mathcal{B}'_o$ happen to be quite close. This observation should of course not be generalised.

[(e)] The same remark as for Fig. 9.1 applies: Fig. 9.2[p183] uses only 156 TWA for its last data point ($|Q| = 7$).

The appendix to the conference version of the paper, available at the above address, also shows the outputs of the two transformations of $\mathcal{X}$ by Algo. 2, and Algo. 3, which are a bit too large – especially for Algo. 2 – to be reproduced in this thesis.

9.6 Conclusions

In this chapter we have introduced tree overloops, and applied both loops and overloops to common operations on TWA: deciding membership, transforming a TWA into a BUTA, and inexpensively testing emptiness. We have shown that the use of overloops simplifies the transformation into BUTA, and substantially lowers the upper bound in the deterministic case.

We intend to pursue this further by using overloops to characterise useful classes of TWA and perform significant simplifications on the automata, hopefully leading to applications to XPath.

Furthermore, while our theoretical results and experiments show that the overloops-based transformation yields much smaller BUTA than the loops-based one, both asymptotically and in average – and yields them proportionally faster, – it is clear that further advances remain possible in that respect. On-the-fly variants enabling to test emptiness (for instance) while forgoing the computation of the whole BUTA would also be of interest.

Acknowledgements. We would like to thank the members of the INRIA ARC ACCESS for interesting discussions on this topic. Our thanks go as well to the anonymous reviewers of the published papers – for both the conference version [Héam, Hugot & Kouchnarenko, 2011] and the journal version [Héam, Hugot & Kouchnarenko, 2012b] – who provided the tighter complexity bound for Cor. 9.7, and whose careful proofreading improved the readability of both papers.

— Part V —

Summary and Perspectives

Chapter 10
Summary and Future Works

The common thread of our contributions is the use of various strains of tree automata in validation and verification problems, with a particular emphasis on the use of constraints, as a means of achieving the expressive power necessary to carry out the task at hand, and approximated methods, as a tool to palliate the unavoidable increase in algorithmic complexity. The main focus of this thesis is the verification of temporal specifications for infinite state systems, and it is that line of inquiry which motivates our study of tree automata with global constraints. While this strain of automata was originally developed in the context of queries on semi-structured documents, their interesting properties with respect to rewriting make them a central component of the tree regular model-checking framework which we develop. Or more accurately, tree "not quite regular" model-checking, as the point of using these automata is precisely to obtain exact representations of non-regular languages which would otherwise require approximation techniques. Other verification problems of interest to us include the validation and querying of semi-structured documents, which impelled our work on tree-walking automata.

10.1 Summary of Contributions

In Part II, we have generalised the previous work of [Courbis et al., 2009] into a full verification chain on a fragment of LTL sufficient to express a large range of safety properties for term rewriting systems. There are two aspects to this process: the translation of temporal formulæ into rewrite propositions, which is a completely abstract operation, although one may perform optimisations even at that point, and the instrumentation of the rewrite proposition into an automata-based procedure. This second step is tied to the properties of the underlying models, namely tree automata with or without global equality constraints. To solve the first step, we have introduced the notion of signatures, which captures a temporally flattened view of a fragment of LTL, that we use to keep track of the generated tree languages

at different points in time. Then the construction and deconstruction of signatures acts as a central part of the set of translation rules which we define to effect an automatic translation into rewrite propositions. The second step is characterised by other sets of transformations – this time called generation rules – which we introduce to automatically keep track of the expressive power required to represent the languages involved, and apply approximated methods when necessary. This yields a set of theorems, each corresponding to different, incomparable ways in which the approximations could be invoked, and therefore to different positive approximated procedures.

In Part III, we focus on tree automata with global equality constraints; in particular, we investigate the effects of bounds on the number of constraints, from the point of view of algorithmic complexity and expressive power. In particular, we show that membership becomes testable in polynomial time given any fixed bound, while emptiness and finiteness reach full complexity – ExpTime-complete – with as few as two constraints. We also propose a SAT encoding of the unbounded membership problem, with encouraging experimental results.

In Part IV, we have introduced the notion of overloops and thereby improved the transformation of tree-walking automata into equivalent BUTA – very significantly in the deterministic case, from the order of 2^{x^2} to $2^{x \log x}$ – and developed an efficient positive approximated procedure for emptiness testing, which boasts great accuracy on randomly generated samples.

10.2 Future Works & Perspectives

In our model-checking framework, the first translation step may still be improved in several ways. For one thing, a formal proof – within a formal proof system such as Coq [Castéran, Herbelin, Kirchner, Monate & Narboux, 2012] or Isabelle [Paulson, 1989] – would be a boon. We have provided complete and detailed manual proofs for the translation and, in retrospect, it seems to us that it would not have been that much harder to write them directly in a formal system. The mathematics involved are occasionally tedious, but always confined within relatively simple arithmetic theories. Apart from the increased confidence in the correctness of the proofs which a full formalisation would bring, this would also provide guaranteed-correct implementations, through code generation. Since performance is not a concern in this first step – the runtime of the automata-based procedure will always dwarf that of the translation into rewrite propositions, – and given the sensitivity of the algorithm to even small bookkeeping mistakes, it is our contention that the proper way of implementing this step is by mechanical derivation from a formal proof, rather than directly.

Currently, the main limitation of this translation is its inability to deal with eventuality. To an extent, this seems to be an intrinsic feature of rewrite propositions, as the languages which are computed are the product of indistinct bulks of traces. To escape this limitation, it seems advisable to go beyond the language-centred definition of rewrite propositions, and to include properties of the rewrite system itself and of the starting language. For instance, given a rule $r = f(x) \to x$ and

any starting language such that the height of terms containing f is bounded, it is clear that $\Diamond \neg\{r\}$ holds. A systematic investigation of the classes of languages and systems for which such arguments hold may yield sufficient information to approximate useful liveness properties. Of course, a drawback of going in that direction is the need to extend the underlying algorithmic toolbox beyond automata. Another related limitation is the handling of those disjunctions and negations that cannot be coerced into a translatable form through manipulations of the formula.

This being said, there are a number of obvious extensions to the system, even without changing its expressive power stricto sensu. For instance, one could add past-time modalities to the supported language, as they can be handled in the same way as their future-time counterparts, simply by substituting $\mathcal{R}^{-1}$ for $\mathcal{R}$. While past operators do not bring any additional expressive power to LTL [Gabbay, Pnueli, Shelah & Stavi, 1980; Gabbay, 1987], they do bring greater succinctness [Laroussinie, Markey & Schnoebelen, 2002], and furthermore their pure-future translations are heavy on *until* operators, and thus untranslatable by our methods. Thence past modalities do extend the range of properties which we can check, at no cost at all whether on the theoretical or computational front – though the desirable linearity properties are mirrored.

If one is willing to leave the confines of LTL and step into CTL* territory, then one may add $\pi := \neg\pi$ to the grammar of rewrite propositions – cf. section 4.1.3[p58] – and thus trivially add support for a fragment of existential LTL corresponding to the negation of the currently supported, implicitly universal, fragment. This is of limited interest for verification, however. More generally, it would be interesting to characterise the full expressive power of rewrite propositions within the larger contexts of CTL*, μ-calculus, and beyond.

It would also be very interesting to extend the scheme to support both state-based and transitions-based properties at the same time. In both cases, the verification boils down to tests on languages, and it is thus a natural extension, which would increase the practical applicability of the method.

As for the second step, that is to say translation into automata-based procedures, looking at existing encodings of systems by means of term rewriting, some domains tend to exhibit good linearity properties, which are portents of good precision and tractable complexities – for instance byte-code semantics or CCS. Other domains, such as protocol verification, appear trickier, as non-linearity is integral to the operations involved; what to do then? The crux of the matter is to achieve a representation of the languages involved, using either a tractable number of constraints – an aspect studied in Part III, – diagonal constraints, or any other mixture of constraints with sufficient expressive power and good decision procedures.

In particular, we intend to investigate a new class of tree automata with constraints, where constraints are not allowed to nest in a run, and such that every class of the togetherness relation (6.2)[p123] on the active constrained states (6.5)[p126] of any run is of a cardinality bounded by some integer m. Let us call them *tree automata with flat equality constraints* (*TAFE*). Such automata are strictly more expressive than $\mathrm{TA}^{=}_{1}$, and incomparable with $\mathrm{TA}^{=}_{k}$, for $k > 1$, because they accept the languages ℓ_k of the general form (6.3)[p126], for any k, and not the language of Fig. 6.1[p120], for any $n > m$. Intuitively, emptiness for TAFE must be decidable in polynomial

time – $O(\|\mathcal{A}\|^m)$ – by straightforward generalisation of the Rigidification Lemma (6.2[p119]). Thus they seem to strike a good compromise between expressive power and algorithmic complexity.

There remain many important open questions for $TA^{=}_{k}$, the most prominent of which is whether there is any $k > 0$ such that containment is decidable; it would certainly be extremely convenient for us if there were. In any case, finding good positive approximated procedures for containment, at least for RTA, is necessary to implement our framework. Furthermore, the overarching question is to find the "best" strains of automata. That is to say, amongst all possible automata with good properties with respect to rewriting – better than BUTA – as well as efficient emptiness decision – let us say PTIME – and, if at all possible, decidable containment, we want to find those with maximal expressive powers.

It is possible that capabilities could be added to some variants of automata with equality constraints without making them harder to handle. For instance, what happens if disequality constraints are added to the mix? And for that matter, what is the complexity of the emptiness problem for TAGD with one constraint? Is containment decidable for TAGD, and if not, for up to how many constraints might it be decidable? On a different front, what happens if we add constraints between brothers, à la NParikh+EDB (cf. section 5.2.1[p112])? Or rather – as it is clear that just adding even a single transition with one equality constraint between brothers to $TA^{=}_{1}$ entails EXPTIME-hardness again – how might we restrict them so as to increase expressive power while preserving the desired complexities?

Another open question, of great practical interest to our model-checking framework, is that of finding a class of automata closed through one-step rewriting, and therefore through any finite number of rewriting steps. As it turns out, while TAGE do capture one step of rewriting from a regular language, they are not closed in that sense. That is to say, given a TAGE-accepted language $\Pi^{=}$, it is not the case in general that $\mathcal{R}(\Pi^{=})$ is also TAGE-recognisable – simple counter-examples can be built by pumping arguments, similarly to the proof of Prop. 6.8[p128]. If a class closed by one-step rewriting exists, and is reasonably tractable, it might eventually replace TAGE in our verification scheme. In the meantime, an easier question would be whether $\mathcal{R}(\Pi)$ is TAFE-recognisable, where Π is regular. If so, they are a drop-in replacement for TAGE, with polynomial emptiness tests, instead of EXPTIME-complete.

With regards to Part IV and tree-walking automata, there remains to see how the overloops-based construction would fare with extended tree-walking models, for instance with pebbles, registers or stacks. It is also possible that, by a kind of converse of Lemma 9.25[p178], an analysis of TWA in terms of their overloops might yield a procedure to determinise them whenever it is possible to do so, or at least in a number of cases. Furthermore, it seems that a number of immediate questions regarding their binary encodings have yet to be studied; for instance, as seen at the end of section 8.4.1[p163], the first-child next-sibling encoding works well for TWA, but what of the tree-currying encoding (cf. example (8.5)[p155])? Would a variant model of TWA whose transitions depend on the symbol at the parent position be an appropriate solution to the problem of checking the parent on unranked trees? Wouldn't it be simpler to use a "first-child, next-sibling, and parent-symbol" ternary encoding?

Chapter 11
Appendix

Contents

—Where we visit some relatives of our favourite automata.

Some interesting classes of automata were omitted from the surveys of Chapters 5[p107] and 8[p143] because, while related to our interests, they do not have any direct bearing on the questions asked in this thesis. For the sakes of curiosity and exhaustiveness, we say a few words about them in the present appendix.

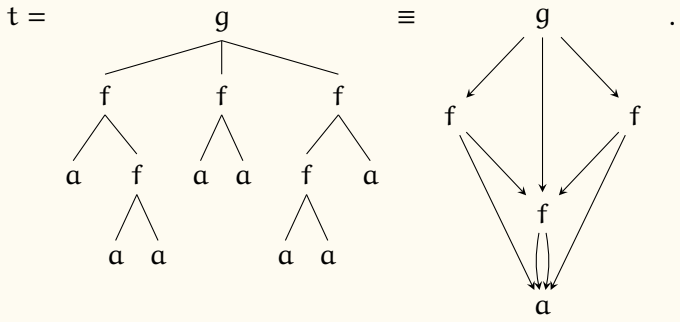
11.1 More Relatives of Automata With Constraints

11.1.1 Directed Acyclic Ordered Graph Automata

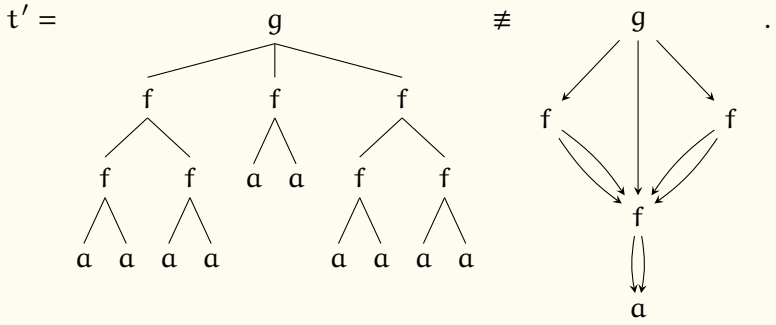
While automata with constraints were not studied as such before 2008, there are closely related classes which were known well before then. One such class is that of automata on directed acyclic ordered graph (DAG) representations of terms with maximal sharing of structure, or more simply *DAG automata* (*DAGA*) [Charatonik, 1999]. Instead or running directly on terms, those automata run on their DAG representations, where the maximal sharing property ensures that the DAG corresponding to a term is unique, up to isomorphism on labelling and structure. That property states – informally – that no two isomorphic closed subgraphs may be rooted in different positions in the DAG. Let us take the – typical – example of the term $t = g\big(f(a, f(a, a)), f(a, a), f(f(a, a), a)\big)$, writing $t \equiv d$ for "t

DAG automata
DAGA

has the DAG representation with maximal sharing property d″:



This is to be put in contrast with the following counterexample, whose DAG violates maximal sharing by duplicating $f(f(a,a),f(a,a))$, and is therefore not a valid representation of the term $t' = g\big(f\big(f(a,a),f(a,a)\big),f(a,a),f\big(f(a,a),f(a,a)\big)\big)$:



Since this property ensures uniqueness of the DAG representation, the expressive powers of DAGA and those of tree automata may be compared by considering the tree language accepted by a DAGA to be the set of terms whose DAG representation is accepted by the DAGA. Under this interpretation, DAGA are strictly more expressive than BUTA. DAGA are closed by union and intersection, but not by complementation – as most of the classes in this chapter, they are not determinisable. Membership is testable in linear non-deterministic time, and emptiness is NP-complete.

We mentioned earlier that DAGA are closely related to tree automata with global constraints; let us now explain in what sense. The key is in the maximal sharing property: since equal subterms will, by definition, be rooted in the same position of the DAG, and a run of the DAGA being a relabelling of the DAG, it follows that those subterms will be evaluated in the same state. Therefore, should one want to ensure that two subterms are different, it suffices to arrange for them to be taken in two different states by the run. Recall the language $L_{\neq}$ (2.5)$_{[p35]}$; it is accepted by a DAGA with $Q = \{p, q, q_f\}$, $F = \{q_f\}$, and transitions

We abbreviate multiple rules in the obvious way: the first rule stands for $a \to p$, $b \to p$, $a \to q$, $b \to q$. There are seven rules in total.

$$\Delta = \{\, a, b \to p, q;\ f(p,p) \to p, q;\ f(p,q) \to q_f \,\}.$$

Thus it seems that DAGA are capable of simulating disequality constraints. What about equality constraints? That two nodes are evaluated into the same state says nothing about whether the subterms are the same – no more so for DAG than for trees. DAGA are actually incapable of simulating equality constraints. The gist of the argument relies on the pumping lemma, which carries over to DAGA. All in all, DAGA have exactly the same expressive power as TAGD. It is in fact easy to define

a TAGD equivalent to a DAGA, as it suffices to define $p \not\cong q$ for all distinct states $p \neq q$. The reciprocal construction is more involved, and incurs an exponential blow-up [Vacher, 2010, Thm. 4.1].

11.1.2 Tree Automata With One Memory

tree automata with one memory
TA1M
PDTA
PDA

Another way in which automata may hold the capability to test equalities is exemplified by *tree automata with one memory* (*TA1M*) [Comon & Cortier, 2005; Comon, Jacquemard & Perrin, 2008] and their subclasses. They generalise pushdown tree automata (*PDTA*) [Guessarian, 1981, 1983] – which themselves generalise pushdown word automata (*PDA*) as well as BUTA, and accept context-free tree languages. TA1M carry an unbounded memory in the form of a tree structure, instead of the stack of PDTA. They are capable of testing equality between parts of their currently stored memory, in which respect they generalise TABB – with a number of caveats which we shall come to shortly. The exact formulation of the definition of TA1M varies from one paper to another; here we define their capabilities in terms of rewrite rules, as in [Vacher, 2010; Comon et al., 2008], while allowing the full range of operations from both these sources and the original definition of [Comon & Cortier, 2005]. This synthesis is summarised in Fig. 11.1 and discussed below.

Note that the tree automata with memory (*TAM*) of [Comon et al., 2008] are a more general class, and as powerful as Turing machines, since there is no restriction whatsoever on memory operations.

A TA1M is a quintuple $\langle \mathbb{A}, \mathbb{M}, Q, F, \Delta \rangle$, where $\mathbb{M}$ is a ranked alphabet, called the memory signature, which serves to encode *memories* as ground terms of $\mathcal{T}(\mathbb{M})$. Memories are stored in the states, which are taken as unary symbols for this purpose. Each transition of a TA1M is of one of three specific forms, all of which are specialisations of the general pattern

$$\sigma\big(p_1(m_1), \ldots, p_n(m_n)\big)[C] \to q(m) \,,$$

with $q, p_1, \ldots, p_n \in Q$, $m, m_1, \ldots, m_n \in \mathcal{T}(\mathbb{M}, \mathbb{X})$, and $C \subseteq \mathbb{X}^2$ a set of constraints of the form $x_i \cong x_j$, whose operands must appear in the left-hand-side of the rule. The three kinds of transitions correspond to a generalisation to trees of the usual pushdown operations push and pop, as well as an internal operation. Using $x_1, \ldots, x_n, y_1, \ldots, y_n \in \mathbb{X}$, $M, k \in [\![1, n]\!]$, $h \in \mathbb{M}_k$, $N \in [\![1, k]\!]$, and ρ a permutation on $[\![1, n]\!]$, they are:

$$\sigma\big(p_1(x_1), \ldots, p_n(x_n)\big)[C] \to q\big(h(x_{\rho\,1}, \ldots, x_{\rho\,k})\big) \qquad \text{(push)}$$

$$\sigma\big(p_1(x_1), \ldots, p_i(h(y_1, \ldots, y_k)), \ldots, p_n(x_n)\big)[C] \to q(y_N) \qquad \text{(pop)}$$

$$\sigma\big(p_1(x_1), \ldots, p_n(x_n)\big)[C] \to q(x_M) \,, \qquad \text{(int)}$$

where each transition may only be applied if, for every $x_i \cong x_j \in C$, it holds that $t_i = t_j$, where t_i and t_j are the subterms matched by x_i and x_j, respectively. The stored memory is irrelevant to the acceptance or rejection of a term: the language accepted by a TA1M $\mathcal{A}$ in state q is defined as

$$\mathcal{L}^q(\mathcal{A}) = \{\, t \in \mathcal{T}(\mathbb{A}) \mid \exists m \in \mathcal{T}(\mathbb{M}) : t \to_\Delta^* q(m) \,\} \,,$$

where the notion of rewriting is extended with the satisfaction of constraints. As mentioned above, this characterisation is a synthesis of the definitions in the existing literature. Figure 11.1 offers a high-level summary of the differences between the definitions of the transition rules in our sources. The comparison considers the following points:

	[Comon & Cortier, 2005]	[Comon et al., 2008]	[Vacher, 2010]	Our version
Permutations	**yes**	no	no	**yes**
Partial push	**yes**	no	no	**yes**
Tests everywhere	**yes**	no	no	**yes**
Duplicates in push	no	∅	∅	**yes**
Multiple tests	**yes**	no	no	**yes**

Figure 11.1: TA1M: capabilities of transitions in the literature.

(1) Permutations: in a push operation, is it possible to change the order in which the variables appear? While this seems permissible in [Comon & Cortier, 2005], push transitions are defined as

$$\sigma\big(p_1(x_1),\dots,p_n(x_n)\big) \to q\big(h(x_1,\dots,x_n)\big) \tag{11.1}$$

in [Comon et al., 2008; Vacher, 2010], and thus the variables cannot be reordered in memory. This probably does not have any effect on complexity or decidability.

(2) Partial push: For the same reason, it is not possible to drop memories in a push with the definition of [Comon et al., 2008]: the right-hand side of (11.1) is $q(h(x_1,\dots,x_n))$, and could not be $q(h(x_1,\dots,x_{n-1}))$, for instance. This seems to be allowed in [Comon & Cortier, 2005], although the wording does not make that explicit.

(3) Tests everywhere: [Comon & Cortier, 2005] allows to perform equality tests on any transition, while [Comon et al., 2008] restricts tests to internal transitions; their purpose in doing so is to allow for a sufficient characterisation of constraints for which emptiness is decidable.

(4) Duplicates in push: The original definition of [Comon & Cortier, 2005] requires to group the matched terms into C-equivalence classes and choose – at most – one representative per class to be stored in memory. This does not match (push), and even less (11.1), which does not have any constraints. We allow duplicates in our own synthesis – with unknown effects on complexity – because it is quite convenient to use that feature to simulate equality constraints.

(5) Multiple tests: The original definition permits several equalities to be tested simultaneously on a single transition. The other definitions are of the form

$$\sigma\big(p_1(x_1),\dots,p_n(x_n)\big)[x_i \cong x_j] \to q(x_M)\,,$$

with only one equality on each – internal – transition.

Let us use this model to accept $L_=$ once again. We take $\mathbb{M} = \mathbb{A}$, as we are simply going to memorise the tree as we evaluate it, along with the usual states $Q =$

$\{p, q, q_f\}$, $F = \{q_f\}$, and the following transitions:

$$a \to p(a) \qquad f(p(x), p(y)) \to p(f(x, y))$$
$$b \to p(b) \qquad f(p(x), p(y))[x \cong y] \to q_f(f(x, y)) \,.$$

Note that if our definition did not allow duplicates, the transition rule

$$f(p(x), p(y))[x \cong y] \to q_f(f(x, y))$$

would not be legal. However in that case, we could still replace it by

$$f(p(x), p(y))[x \cong y] \to q_f(f'(x)) \,,$$

with $\mathbb{M} = \mathbb{A} \uplus \{f'/_1\}$, and achieve the desired result, although the automaton no longer memorises the exact visited tree. However, this trick may not work in all circumstances; within a run, the same subterm may be stored in two different forms – primed and unprimed – depending on whether an equality test was performed.

Nevertheless, it is clear that by this method – allowing duplicates – we can simulate equality constraints between brothers in the style of TABB, although the constraints may only be taken conjunctively. To our knowledge, the class of TA1M with propositional constraints has not been studied in the literature. Such as defined in [Comon & Cortier, 2005], TA1M are closed by union, but not by intersection nor complementation, and emptiness decision is ExpTime-complete. Subclasses with better closure properties have been defined in [Comon et al., 2008], in particular *visibly tree automata with memory* (*VTAM*), which sport PTime-complete emptiness decision, PTime membership tests, ExpTime-complete universality and inclusion problems, and are closed under all boolean operations. VTAM restrict TA1M by adding a visibility requirement, in that the type of a transition – push, pop or internal – is determined by the symbol in which it is rooted. They remain strictly more general than BUTA, however, since the type of rules only influences what becomes of the memory, and all three types subsume the standard bottom-up transitions. Over and above the visibility condition, the basic version of VTAM does not allow equality tests either.

visibly tree automata with memory
VTAM

However, the authors of [Comon et al., 2008] went further, and studied classes – arguably even meta-classes – of VTAM with constraints. The particular nature of the constraints is abstracted into any equivalence relation $R \subseteq \mathcal{T}(\mathbb{M})^2$ on memories, and sufficient conditions are found for emptiness to be decidable for VTAM equipped with positive and negative R-constraints, under the restriction that only internal transition may effect R-tests. It is found that equality of terms satisfies those conditions and that therefore, for VTAM with equality and disequality constraints, emptiness is ExpTime-complete and membership is NP-complete, but universality is undecidable, and they are not determinisable nor closed by complementation. The class of VTAM with positive and negative structural equivalence tests — $u \cong v \Leftrightarrow \mathcal{P}(u) = \mathcal{P}(v)$ — have even better properties, as they are closed under boolean operations with the same complexities as for BUTA, and universality and inclusion are decidable. That subclass has even been extended with the ability to test equality and disequalities between brother positions for the input term, as opposed to the memory. This superclass (*VTAMSB*) is closed under boolean operations, and has decidable emptiness, although with a high complexity.

VTAMSB

TA1M and their relatives are difficult to put in relation with the other strains of automata seen in this thesis. It is not at all clear how they compare with TAGC, from the viewpoint of expressive power. Their memory brings some degree of global reach to the constraints which they can test, which enables TA1M – as defined above – to capture TAGE-languages which TABB cannot express [Vacher, 2010, Ex. 2.21], but it is unknown whether they can actually simulate global constraints in general. It is conjectured [Vacher, 2010, Cj. 2.25] that they cannot. On the front of positional constraints, memory can be used simply to duplicate the input tree, and thus a version of TA1M with disequality constraints should easily be able to simulate the conjunctive variety of TABB. As defined, they should simulate the conjunctive and positive subclass. Note our use of the conditional above; we could not find any clear statement of this in the literature. The same constructions do not necessarily apply for the VTAM variety extended with equality and disequality tests, because they are defined such that only internal rules have access to tests. Since internal rules cannot push, this means that effecting a test would instantly destroy the synchronisation of the memory with the visited term: it would hereafter be missing a symbol. This does not seem to have been explicitly studied in the literature.

11.2 More Relatives of Tree-Walking Automata

There are many classes of extended tree automata; invariably with increased expressive power. The following should be close to an exhaustive survey – albeit a superficial one.

11.2.1 Tree-Walking Pebble Automata

tree-walking pebble automata
TWPA

[Engelfriet & Hoogeboom, 1999] introduces *tree-walking pebble automata* (*TWPA*) as a remedy against the unfortunate tendency of TWA to get lost in trees; as they put it, in a binary tree of which all internal nodes have the same label, all nodes look pretty much the same. A well-researched means of palliating such problems is to use pebbles, so as to identify places which have already been visited [Thumb, 1697; Gretel & Hänsel, 1812]. Slightly more recently, this has been applied to mazes, which are shown to be solvable in general by maze-walking finite automata equipped with two pebbles [Blum & Kozen, 1978].

TWPA follow the same basic principles. A TWPA is a tree-walking automaton which is equipped with a fixed finite set $\{1, \ldots, n\}$ of pebbles, and supports two new commands and a new test:

Test: is pebble k on the current node?
Command: drop pebble k on the current node.
Command: remove pebble k from the current node.

Of course, the behaviour follows the metaphor of pebbles closely: you can only drop a pebble once in a row; to drop the same pebble again, it must be retrieved first. Furthermore, and less obviously, there is a stack discipline imposed on the pebbles, so that – numbering pebbles from 1 – any pebble k can only be dropped

if $k-1$ is on the tree (or it is 1), and k can only be lifted if $k+1$ is not on the tree. Without this stack discipline, the expressive power of pebble automata would jump far beyond that of regular tree language, to $\text{NSpace}(\log n)$, and the emptiness problem would become undecidable, even for just two pebbles on words.

With this stack discipline, the expressive power of TWPA is confined to regular languages – in fact, regardless of the number of pebbles, they only recognise a proper subset of them, as shown in [Bojanczyk, Samuelides, Schwentick & Segoufin, 2006; Samuelides, 2007]. The results shown in these papers actually go further; to present them, let us keep track of the number of pebbles available by writing TWPA_n for TWPA with n pebbles, and DTWPA_n for their deterministic counterparts – cf. Def. 9.19[p177]. It was shown that $\mathcal{L}(\text{TWPA}_k) \subset \mathcal{L}(\text{TWPA}_{k+1})$, and $\mathcal{L}(\text{DTWPA}_k) \subset \mathcal{L}(\text{DTWPA}_{k+1})$, for $k \geqslant 0$, and furthermore, for every k, there is a TWA-definable language – i.e. TWPA_0-definable – which is not DTWPA_k-definable – this strengthens the known non-determinisability results for TWA. It is also shown that, although the number of states may explode, it does not change the expressive power of TWPA_k – nor that of DTWPA_k – if one allows the "lift pebble" command to remove the current pebble regardless of the current position of the head – a policy known as the *strong model* of TWPA. Moreover, for all k, DTWPA_k are closed under complementation; this is also presented in [Muscholl et al., 2006]. It is also known [Engelfriet & Hoogeboom, 1999, Sec. 4] that the expressive power of deterministic top-down tree automata is captured by TWPA_1.

It should be pointed out that, although every TWPA may be transformed into an equivalent BUTA, the transformation is necessarily non-elementary [Samuelides & Segoufin, 2007; Samuelides, 2007]. More specifically, there is an unavoidable exponential blowup each time a pebble is added.

In [Engelfriet & Hoogeboom, 2007], it is shown that TWPA characterise first-order logic with positive monadic transitive closure – see section 8.3.3 – and DTWPA characterise first-order logic with deterministic transitive closure – which we shall not define here; see [Neven, 2002, Sec. 5.3]. This result is even more general, as it applies to automata with any number of heads, and any adicity for the transitive closure relation. More specifically, recall the transitive closure operator $+_{x,y}$ presented at the end of section 8.3.2, but extend it so that x and y are not single variables, but sequences of variables of length k; then this defines the k-ary transitive closure. [Engelfriet & Hoogeboom, 2007] shows, in all generality, that first-order logic with positive k-ary transitive closure characterises TWPA with k heads, and the same generalisation for the deterministic variants.

Whether TWPA are closed under complementation is an open problem, equivalent to the problem of whether first-order logic with positive monadic transitive closure is properly included in FOT [ten Cate & Segoufin, 2010, Sec. 8.4].

11.2.2 Tree-Walking Invisible Pebble Automata

This variant was introduced in [Engelfriet, Hoogeboom & Samwel, 2007], although it used both visible and invisible pebbles. It is also discussed in [Bojańczyk, 2008], where it uses only invisible pebbles, and it is this definition which we are going to use.

A tree-walking automaton with invisible pebbles is the same as a TWPA, except for two important differences:

(1) It has an unbounded number of pebbles, each bearing a colour taken from a finite set C. The same colour may be taken any number of times.

(2) At any given time, only the last dropped pebble can be tested for or lifted. The colour, as well as the presence of the pebble, can be tested.

The second restriction is quite important: without it, emptiness would be undecidable. In essence, all pebbles but the last one are invisible to the automaton – hence the name. This definition assumes the weak model of pebble removal, whereby a pebble may only be removed when the head is on its position.

This model captures exactly the regular languages: the intuitive idea is that a branching automaton $\mathcal{A}$ can be simulated as follows, using a colour for each state: $C = \mathcal{A} : Q$. The algorithm is recursive, and starts by evaluating the right subtree. When it is done, it drops a pebble with the colour of the resulting state q_1 at the root of the subtree. Then it does the same for the left subtree, and drops the pebble q_0. After that, it can visit two children, reading the pebbles, and the root, reading its label σ, and choose the resulting pebble q accordingly, for a rule $\sigma(q_0, q_1) \to q$. The converse is also sketched in [Bojańczyk, 2008, Thm. 10], and rests on alternating tree-walking automata, which are the object of section 11.2.5.

11.2.3 Tree-Walking Marbles Automata

A related, but older model was presented in [Engelfriet, Hoogeboom & Best, 1999]. Instead of using coloured pebbles, as above, this variety uses shiny marbles. There again, there is an infinite supply of marbles, each in one of a fixed and finite set of distinct colours. There is no particular stack discipline, and marbles are all visible and may be dropped, tested for and lifted on the current node, with the restriction that only one marble of a given colour may mark any given node. The important restriction which replaces the stack discipline of the other models is that the automaton may not move to the parent of a node which is marked by a marble. In other words, dropping a marble closes off the context of the current node α, at least until every single marble is lifted from α. A consequence of this is that, at any given time, all dropped marbles are along the path from the current node to the root.

This model is equivalent to the tree-walking automata (actually DAG-walking) with synchronised pushdown which appear in [Engelfriet, Rozenberg & Slutzki, 1980; Kamimura & Slutzki, 1981]. There the automaton has a stack, to which it pushes each time it goes down to a child, and from which it pops every time it goes up to a parent. It is clear that this is equally powerful as the marble model – but less convenient if the run might begin at any node, and not necessarily at the root, hence the creation of the marbles strain.

Both marble and synchronised pushdown tree-walking automata, whether deterministic or not, accept exactly the regular tree languages. The proof is very similar to that for invisible pebbles.

11.2.4 Tree-Walking Set-Pebble Automata

We are not quite done yet with pebbles. [Engelfriet & Hoogeboom, 1999, Sec. 6] proposes a variant of TWPA which, instead of merely marking a single node with a pebble, marks an arbitrary set of nodes – hence the name "set-pebbles". The motivation stems from the capability of pebble automata to simulate first-order logic formulæ by using pebbles to encode node quantification. It is therefore natural to lift pebbles from single-node markers to markers for sets of nodes in order to deal with second-order monadic quantification, that is to say set quantification, and thereby capture the regular languages. Note that in this model, dropping and lifting are independent from the current position of the head, lifting means lifting from all the marked nodes simultaneously, and the usual stack discipline applies.

As expected, this variant captures exactly the regular tree languages.

11.2.5 Alternating Tree-Walking Automata

Alternation can be added to tree-walking automata (cf. section 8.1[p144]) in the classical way, by partitioning the states into universal states $Q_\forall$ and existential states $Q_\exists$, and branching when a transition starts from a universal state. A run then becomes a tree instead of a sequence of configurations, and it is accepting if all the leaves are accepting configurations.

Alternation can be used to simulate branching. For instance, it is easy to simulate a non-deterministic top-down tree automaton [Hosoya, 2010, Sec. 12.2.3]. Consider a transition rule $q \to \sigma(q_0, q_1)$; an alternating TWA can simulate that rule by having q as a universal state, and the transitions

$$\langle \sigma, q, \tau \to \swarrow, q_0 \rangle \quad \text{and} \quad \langle \sigma, q, \tau \to \searrow, q_1 \rangle \,,$$

with appropriate type τ. The reciprocal inclusion was shown to hold in [Slutzki, 1985]. Therefore alternating TWA recognise exactly the regular tree languages.

Alternating TWA can be converted into branching automata, with an exponential blowup in the number of states, following the same kind of transformation as basic TWA.

Chapter 12
[FR] Résumé en français

—Où l'on se répète dans la langue patriotique de rigueur et avec entrain.

Le tronc commun de cette thèse est l'utilisation des automates d'arbres – dans leurs diverses incarnations – non seulement pour la vérification de systèmes, mais aussi pour les requêtes et autres tâches relatives aux documents et bases de données semi-structurés.

Notre principal soucis et l'objet central de nos recherches est la vérification de systèmes à états infinis. Plus précisément, l'objectif final est de développer une chaîne de vérification complète et fonctionnelle sur la base d'une méthode de model-checking située à la confluence du model-checking d'arbres régulier, de l'analyse d'accessibilité, et de la logique de réécriture.

L'idée générale de cette méthode a été originellement présentée au moyen de preuves manuelles sur des exemples dans [Courbis et al., 2009]. Elle associe des aspects de model-checking régulier et d'analyse d'accessibilité à la vérification de propriétés exprimées en logique temporelle. Notre objectif est la généralisation de ce processus à un fragment de la logique temporelle linéaire, et afin de l'accomplir nous avons recours aux automates d'arbres avec contraintes globales d'égalité, un modèle au grand pouvoir d'expression développé à l'origine dans le contexte des logiques pour les requêtes XML. Ceci nous amène également à étudier le modèle pour lui-même.

Un objectif secondaire de la thèse est l'amélioration des méthodes algorithmiques pour les automates d'arbres cheminants, un modèle de calcul présentant de fortes connexions avec les documents semi-structurés et, en particulier, leurs langages de navigation.

La partie II constitue le cœur de nos contributions, car elle couvre la méthode de model-checking elle-même, et offre un réponse positive à la question posée: l'idée de [Courbis et al., 2009] peut être généralisée et étendue à une plateforme de vérification automatique pour un fragment de la logique temporelle linéaire. Ceci est réalisé au moyen de deux étapes de traduction distinctes, pour lesquelles nous présentons des ensembles de règles de traduction automatique. La spécification temporelle est dans un premier temps convertie en une forme intermédiaire – une formule de logique propositionnelle dont les atomes sont des comparaisons de langages obtenus par réécriture, que nous appelons une *proposition de réécriture* – en ignorant toutes les propriétés propres du système. Ensuite, la forme intermédiaire est transformée en une procédure (peut-être) approximée – le problème général est indécidable – fondée sur les *automates d'arbres à contraintes globales d'égalité*; les propriétés propres du systèmes sont prises en compte dans cette étape, et affectent la qualité de la procédure approximée résultante. Afin de résoudre le problème

Procédure Approximée

Une procédure approximée positive est une procédure – elle termine toujours – qui répond soit "oui", soit "peut-être" à la question d'un problème de décision.

– assez corsé – de la traduction mécanique d'une spécification temporelle en une proposition de réécriture équivalente, nous introduisons la notion de *signature*, qui nous donne un modèle linéaire de certaines formules temporelles. Cette partie se termine sur une discussion du fragment de logique temporelle couvert par nos méthodes, en termes de la popularité – à en croire les études – des classes de propriétés que la méthode automatique est capable de traiter. Nous écumons également la littérature à la recherche de systèmes intéressants modélisés au moyen de systèmes de réécriture, et nous examinons leur propriétés du point de vue de la seconde étape. Une fraction du contenu de cette partie a été publié dans [Héam, Hugot & Kouchnarenko, 2012a], et la plupart du reste est actuellement en cours de soumission [Héam, Hugot & Kouchnarenko, 2013].

Les auteurs des publications sont cités par ordre alphabétique.

L'utilisation des automates d'arbres à contraintes globales d'egalité (TAGE), dont le pouvoir d'expression est supérieur à celui du modèle standard d'automates d'arbres ascendants, améliore la précision des approximations générées par la plateforme de vérification. Cependant, ce pouvoir d'expression accru vient au prix de hautes complexités algorithmiques pour de nombreux problèmes de décision importants. De plus, il s'agit là d'une classe d'automates relativement récente et, bien qu'elle ait de riches connexions théoriques et de multiples applications à XML et au model-checking, il n'existe à notre connaissance aucune étude – en dehors des nôtres – portant sur l'obtention de procédures de décisions efficaces pour cette classe.

La partie III se concentre sur les TAGE et leurs problèmes de décision; le but est d'obtenir des algorithmes efficaces pour certains problèmes des décision utiles et courants, tels que les tests d'appartenance et de vacuité, ainsi que d'améliorer notre compréhension générale de ce qui rend ces problèmes complexes. Nous présentons un codage SAT du test d'appartenance (un problème NP-complet) et étudions les effets de bornes sur le nombre de contraintes, en montrant que le test d'appartenance est polynomial pour toute borne, et que les tests de vacuité et de finitude sont déjà à pleine complexité avec seulement deux contraintes. Cette étude des bornes a été publiée dans [Héam, Hugot & Kouchnarenko, 2012c], et le codage SAT dans [Héam, Hugot & Kouchnarenko, 2010b]. Nous montrons également que le pouvoir d'expression augmente strictement avec la borne. Dans le même domaine, nous avons aussi travaillé à la réalisation d'heuristiques et de procédés de génération aléatoire pour le test de vacuité (ExpTime-complet); bien que ce travail n'apparaisse pas dans cette thèse, une partie en est disponible sous la forme d'un rapport de recherche [Héam, Hugot & Kouchnarenko, 2010a].

La partie IV est liée à une autre forme de vérification utilisant les automates d'arbres, en relation avec les documents semi-structurés. L'objet de cette partie est l'étude des automates cheminants (TWA), en particulier sous l'aspect de leur conversion en automates d'arbres ascendants. L'introduction de la notion de *surboucle d'arbre* nous permet de réduire considérablement la taille de l'automate généré dans le cas déterministe, environ de 2^{x^2} à $2^{x \log x}$. De plus, nous proposons des algorithmes efficaces pour la décision d'appartenance, et une procédure approximée positive – généralisable à une classe de telles procédures de plus en plus précises – pour le test de vacuité, qui est un problème ExpTime-complet. Ce procédé est testé dans le cadre d'une génération aléatoire uniforme de TWA, et se révèle étonnamment précis. Ce travail est apparu dans les procès-verbaux de conférence

[Héam, Hugot & Kouchnarenko, 2011] et une version étendue a paru dans la revue [Héam, Hugot & Kouchnarenko, 2012b].

Ce résumé en français offre un survol plus détaillé des travaux présentés dans ces trois parties. Le but est d'obtenir une intuituion quant à la structure générale de la thèse, mais en aucun cas d'expliciter les développements techniques, auxquels nous ferons simplement référence. Certaines notations seront employées afin d'éviter d'excessives paraphrases, mais leur définition ne sera qu'esquissée en langue naturelle ou laissée intuitive; là encore, le lecteur avide de rigueur mathématique ou d'exhaustivité devra se référer au corps de texte. Les vocabulaires de base des automates d'arbres, de la réécriture et de la logique temporelle sont des prérequis.

12.1 Approximation de LTL sur réécriture

Grâce aux méthodes symboliques, les techniques de model-checking ne sont en aucun cas limitées au systèmes à espaces d'états finis. Le procédé que nous développons dans la partie II, et plus particulièrement le chapitre 4[p53], emploie la réécriture comme paradigme central, et se rapproche à la fois du model-checking d'arbres régulier, de l'analyse d'accessibilité, et de la logique de réécriture. Dans cette section, nous commençons par résumer le problème à résoudre, offrir quelques éléments de contexte à propos de travaux proches dans ce domaine, et mettre l'accent sur quelques résultats et concepts particulièrement pertinents pour la suite.

L'énoncé précis du problème est donné dans la section 4.1.3[p58]. Le but est de vérifier les propriétés temporelles d'un système – que l'on entende par là un programme, un circuit, ou une machine à billets – dont les états sont représentés par des arbres et le comportement est encodé par un système de réécriture $\mathcal{R}$. Les propriétés ne portent pas sur les évolutions des états du système, mais sur l'enchaînement de ses actions. Il est présupposé dans ce contexte que les règles de réécriture de $\mathcal{R}$ correspondent à des évènements pertinents du système. Ces séquences de règles de réécriture qui capturent l'exécution du système, et qui sont définies formellement et appelées *mots de réécriture maximaux* dans le chapitre 4[p53], fournissent donc la fondation sur laquelle la sémantique temporelle repose.

• versus ∘

Dans la formule $\Box(X \Rightarrow \bullet Y)$, $\bullet$ est l'opérateur "next" de la logique temporelle. Dans ce cas, il s'agit en réalité d'un opérateur next *fort*, par opposition à un next *faible*, qui se note $\circ$. Ceci est expliqué en détail en section 4.1.2[p57], mais n'est pas d'une grande importance dans ce résumé.

Considérons par exemple un langage régulier d'arbres initial $\Pi \subseteq \mathcal{T}(\mathbb{A})$, un système de réécriture $\mathcal{R}$, et la propriété LTL $\Box(X \Rightarrow \bullet Y)$, où $X, Y \subseteq \mathcal{R}$; c'est-à-dire que X and Y sont des ensembles de règles de réécriture, ou actions, du système à vérifier. Cette propriété signifie que lorsqu'un terme accessible est transformé par quelque règle de réécriture de X, l'arbre résultant peut à son tour être transformé par une règle de Y, et pas par une règle n'appartenant pas à Y. Ceci est illustré au moyen de la figure 12.1. Plus concrètement, si $\mathcal{R}$ modélise une machine à billets, et $X = \{\mathsf{demander\_PIN}\}$ et $Y = \{\mathsf{auth}_1, \mathsf{auth}_2, \mathsf{annuler}\}$, alors cette propriété peut être lue comme "chaque fois que l'utilisateur entre son code PIN, quelque-chose se passe immédiatement après, et cela ne peut être que l'authentification de l'utilisateur – par le biais de l'une des deux méthodes disponibles – ou l'annulation de la transaction; ceci exclut toutes autres actions possibles mais indésirables, telles que l'envoi du PIN sur le réseau". Notons que demander_PIN etcetera sont, dans ce contexte, des règles de réécriture sur les arbres représentant les états de la machine.

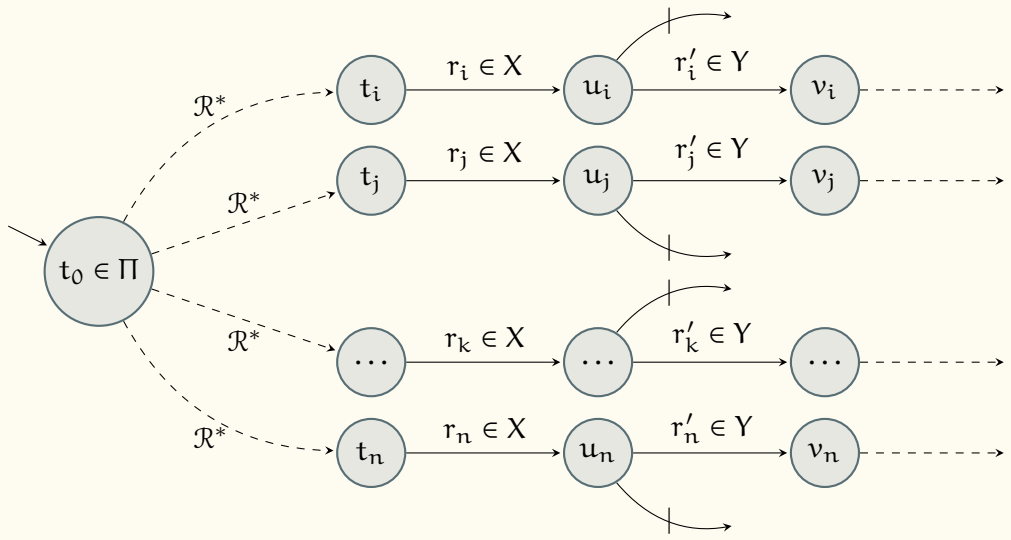


Figure 12.1: [3.1[p42]] Exécutions d'un système de réécriture satisfaisant $\Box(X \Rightarrow \bullet Y)$.

Comme nous allons le voir dans ce qui suit, et en plus de détails dans le chapitre 4[p53], le procédé que nous étudions afin de répondre à de tels problèmes de vérification repose sur le calcul d'automates correspondant aux langages d'arbres atteints après certains nombres d'étapes de réécriture. En ce sens, il est un proche parent de la méthode d'analyse d'accessibilité pour systèmes de réécriture de termes, qu'il généralise sous certains aspects. Là où l'analyse d'accessibilité se ramène à une équation de la forme $\mathcal{R}^*(\Pi) \cap \mathcal{B} = \emptyset$, la vérification de propriétés temporelles requiert la décision – ou tout au moins l'approximation – d'équations de langages plus complexes – appelées *propositions de réécriture* dans le chapitre 4 – telles que, pour l'exemple de la formule $\Box(X \Rightarrow \bullet Y)$,

$$[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \emptyset \ \wedge \ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \ . \tag{12.1}$$

La majeure partie de notre travail consiste à générer de telles équations mécaniquement à partir de la propriété temporelle considérée, ce qui étend les travaux précédents de [Courbis et al., 2009].

La section 3.2[p44] traite des résultats connus en analyse d'accessibilité pour les systèmes de réécriture de termes, qui permettent de traiter des formules telles que (12.1), une fois qu'elles sont obtenues. Ces résultats comprennent entre autres les classes de systèmes de réécriture qui préservent la régularité, i.e. $\mathcal{R}^*(\Pi)$ est régulier si Π l'est, les algorithmes de complétion, et leurs modes de gestion de la non-linéarité – à gauche en particulier. Cette section comporte également une discussion de la complétion pour une étape de réécriture, sujet très peu étudié dans la littérature mais pertinent à notre développement technique, et un rappel de résultats concernant le pouvoir d'expression des TAGE dans ce contexte, qui justifie notre intérêt pour cette classe d'automates. En effet, pour tout système de réécriture $\mathcal{R}$, langage d'arbres régulier Π, et tout langage $\Pi^=$ reconnaissable par un TAGE, les résultats suivants sont connus [Courbis et al., 2009, Prp. 5, 7 & 6] :

(1) $\mathcal{R}^{-1}(\mathcal{T})$ est reconnu par un TAGE – et est régulier si $\mathcal{R}$ is linéaire à gauche,

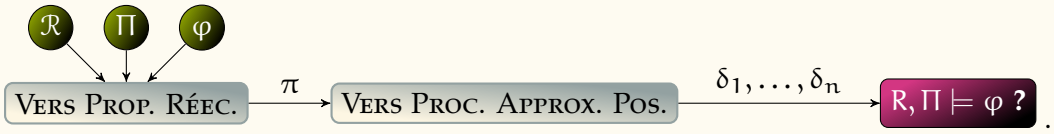
(2) $\mathcal{R}(\Pi)$ est reconnu par un TAGE, and

(3) tester si $\mathcal{R}(\Pi^=) = \emptyset$ est faisable en ExpTime.

Notons que **(1)** $\Rightarrow$ **(3)**: $\mathcal{R}(\Pi^=) = \emptyset \Leftrightarrow \Pi^= \cap \mathcal{R}^{-1}(\mathcal{T}) = \emptyset$.

On peut donc calculer une étape de réécriture exactement, et tester la vacuité du langage obtenu après deux étapes, même sans suppositions supplémentaires concernant la linéarité du système de réécriture.

Revenons maintenant à la question de la provenance de formules telles que (12.1), et au procédé général de vérification. Comme mentionné précédemment, la plateforme de vérification est fondée sur deux étapes de calcul: d'une propriété temporelle à une proposition de réécriture, et de celle-ci à une procédure approximée positive. De manière schématique, l'on commence avec trois entrées: le système de réécriture $\mathcal{R}$, le language d'arbres initial Π, que l'on suppose régulier, et la propriété temporelle φ qui doit être vérifiée. Dans la première étape, la correction du système par rapport à la spécification φ est reformulée en une proposition de réécriture π qui est, au cours de la seconde étape, transformée en une procédure approximée positive δ fondée sur les automates d'arbres avec ou sans contraintes – ou potentiellement plusieurs telles procédures $\delta_1, \ldots, \delta_n$, car il peut y avoir plusieurs manières différentes et incomparables d'effectuer les approximations nécessaires.



Cette approche, inspirée par la méthode de [Genet & Klay, 2000] pour l'analyse de protocoles cryptographiques, a été proposée initialement dans [Courbis et al., 2009], où les deux étapes de traduction sont effectuées et prouvées manuellement sur trois formules spécifiques de logique temporelle linéaire, choisies pour le pertinence en model-checking, en particulier par rapport à la sécurité de MIDLets Java et dans le contexte du projet français ANR RAVAJ. Notre objectif est de généraliser ce travail à un fragment de LTL, c'est-à-dire que les deux étapes de la traduction doivent être mécanisées afin d'obtenir une plateforme de vérification automatique fonctionnelle.

Projet ANR RAVAJ
http://www.irisa.fr/celtique/genet/RAVAJ

La seconde étape demande de jongler avec le pouvoir d'expression requis pour exprimer les langages, et de choisir les lieux d'application des approximations – en particulier les sur-approximations de $\mathcal{R}^*(\Pi)$, mais pas seulement. Nous ne traitons pas cette étape dans ce résumé; le lecteur est invité à consulter la section 4.4[p87] pour plus d'informations. La première étape est nettement plus difficile: il n'est pas clair a priori comment générer les propositions de réécriture. Ceci est visible en résumant les résultats de [Courbis et al., 2009], soit le trio de traductions suivant:

$$\begin{aligned} \mathcal{R}, \Pi \models\ & \Box(X \Rightarrow \bullet Y) \\ & \Leftrightarrow [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \ \wedge\ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \,, \end{aligned} \tag{12.2}$$

$$\begin{aligned} \mathcal{R}, \Pi \models\ & \neg Y \ \wedge\ \Box(\bullet Y \Rightarrow X) \\ & \Leftrightarrow Y(\Pi) = \varnothing \ \wedge\ Y([\mathcal{R} \setminus X](\mathcal{R}^*(\Pi))) = \varnothing \,, \end{aligned} \tag{12.3}$$

$$\begin{aligned} \mathcal{R}, \Pi \models\ & \Box(X \Rightarrow \circ \Box \neg Y) \\ & \Leftrightarrow Y(\mathcal{R}^*(X(\mathcal{R}^*(\Pi)))) = \varnothing \,. \end{aligned} \tag{12.4}$$

Visiblement, la forme générale de la formule temporelle en se reflète pas dans celle de la proposition de réécriture. Ce n'est pas inattendu, car ce n'est après tout pas la syntaxe de la formule qui est traduite, mais la sémantique du fait que le système satisfasse la propriété temporelle exprimée par la formule. De ce fait, on

ne peut discuter de la traduction qu'après avoir clairement défini la sémantique de LTL. Certains choix sont à faire dans ce cas, car les propriétés portent à la fois sur des exécutions finies et infinies, et, contrairement au cas des mots infinis, il y a plusieurs manières de définir la sémantique de LTL sur mots finis. Ceci est traité en section 4.1, qui offre une définition rigoureuse du problème de traduction.

Une fois le problème posé, la section 4.2[p59] offre une idée générale du procédé de traduction que nous développons, ainsi que de ses limites. La principale idée qui nous guide peut être résumée par la comparaison de ces trois traductions:

$$\begin{aligned}
&\mathcal{R}, \Pi \models Y && \iff [\mathcal{R} \setminus Y](\Pi) = \varnothing \ \wedge\ \Pi \subseteq Y^{-1}(\mathcal{T}) \\
&\mathcal{R}, \Pi \models X \Rightarrow \bullet Y && \iff [\mathcal{R} \setminus Y](X(\Pi)) = \varnothing \ \wedge\ X(\Pi) \subseteq Y^{-1}(\mathcal{T}) \\
&\mathcal{R}, \Pi \models \Box(X \Rightarrow \bullet Y) && \iff [\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing \ \wedge\ X(\mathcal{R}^*(\Pi)) \subseteq Y^{-1}(\mathcal{T}) \,,
\end{aligned}$$

Malgré la grande différence syntactique entre ces trois formules – différence qui inclut la présence ou absence de deux opérateurs temporels – il s'agit essentiellement de la même traduction. En effet, les seconde et troisième propositions de réécriture sont obtenues à partir de la première par de simples substitutions: il suffit de remplacer Π par $X(\Pi)$ et $X(\mathcal{R}^*(\Pi))$, respectivement. Cela fonctionne car Y est, d'un certain point de vue, toujours dans la futur par rapport au reste de la formule. La substitution peut donc être vue comme l'accumulation de données concernant le passé – données stockées dans le langage "de départ" Π – qui sont restituées lors de la traduction de l'atome Y, ou plus généralement du conséquent de l'implication à traduire.

Bien entendu, cette intuition n'est en elle-même pas suffisante pour réaliser la traduction; elle échoue lorsque, par exemple, le conséquent Y est dans le passé par rapport à l'antécédent, comme dans $\Box(\bullet X \Rightarrow Y)$. Pourtant cette formule est traduisible: voir l'équation (12.3)[p204], à la notation près. Les moyens techniques que nous introduisons afin de gérer le stockage d'informations ne sont donc pas restreints à une substitution sur Π; nous utilisons également les *signatures*, développées en section 4.2.3 sur un fragment de LTL suffisant pour l'usage que nous en faisons. Elles peuvent être vues comme stockant des informations concernant le futur. La méthode de traduction automatique développée en section 4.3[p73] est un ensemble de règles de traduction reposant sur les signatures; l'essentiel du travail est effectué au moment de la traduction des atomes, pour lesquels les informations sur passé et futur sont extraites du langage et de la signature courants. Une discussion technique des signature ou des règles dépasse la portée de ce résumé; nous en donnons simplement une idée générale par l'exemple, en exhibant une règle de traduction simple mais utile, et un exemple de dérivation – de traduction mécanique – mettant cette règle en œuvre.

$$\updownarrow \frac{\langle \Pi \wr \sigma \ \Vdash\ \Box\, \varphi \rangle \qquad \sigma \text{ est stable}}{\langle \sigma[\omega]^*(\Pi) \wr \star\sigma \ \Vdash\ \varphi \rangle} \qquad (\Box_{*\,[p81]})$$

Cette règle se lit du haut en bas et signifie que, étant donné un passé – i.e. un langage initial – Π et une signature σ, la formule $\Box\,\varphi$ se traduit, sous réserve d'une certaine condition simplificatrice sur φ que nous ignorons dans ce résumé, en la traduction de φ effectuée avec un nouveau langage initial de la forme $X^*(\Pi)$, pour un certain X utilisant des informations de σ, et une certaine modification sur la signature. Dans le cas où la signature est vide – elle est alors notée ε – la règle s'applique et l'on a $X = \mathcal{R}$:

La règle $(\Box_*)_{[p81]}$ est un cas particulier de la règle $(\Box_\hbar)_{[p83]}$ qui gère l'opérateur temporel $\Box$ de manière plus générale.

$$\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \Box \varphi \rangle}{\langle \mathcal{R}^*(\Pi) \fatsemi \star\varepsilon \Vdash \varphi \rangle}$$

Cette règle simplifiée nous dit donc que – nonobstant un détail technique sur la signature – traduire $\Box$ revient à remplacer Π par $\mathcal{R}^*(\Pi)$. Notons que c'est exactement cette opération que l'on observe entre les traductions de $X \Rightarrow \bullet Y$ et $\Box(X \Rightarrow \bullet Y)$ vues ci-dessus. Étant donné que cette dernière formule commence par $\Box$, la dérivation de sa traduction au moyen des règles commence donc par invoquer $(\Box_*)$; elle est reproduite ici:

$$\updownarrow \frac{\updownarrow \frac{\updownarrow \frac{\updownarrow \frac{\updownarrow \frac{\langle \Pi \fatsemi \varepsilon \Vdash \Box(X \Rightarrow \bullet^1 Y)\rangle}{\langle \mathcal{R}^*(\Pi) \fatsemi \star\varepsilon \Vdash X \Rightarrow \bullet^1 Y\rangle}(\Box_*)_{[p81]}}{\langle \mathcal{R}^*(\Pi) \fatsemi ?X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 ! \Vdash \bullet^1 Y\rangle}(\Rightarrow_\Sigma)_{[p76]}}{\langle \mathcal{R}^*(\Pi) \fatsemi ?X \fatsemi \mathcal{R} \mid \overline{\mathbb{N}}_1 ! \Vdash \circ^1 Y\rangle(\bullet^m)_{[p80]}}}{\langle X(\mathcal{R}^*(\Pi)) \fatsemi \star\varepsilon \Vdash Y\rangle(X^\star_\varepsilon)_{[p85]}}(\circ^m)_{[p79]}}{[\mathcal{R} \setminus Y](X(\mathcal{R}^*(\Pi))) = \varnothing} \wedge X(\mathcal{R}^*(\Pi)) \subseteq \mathcal{R}^{-1}(\mathcal{T})\ .$$

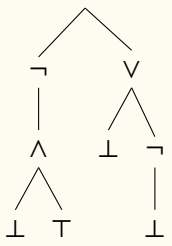
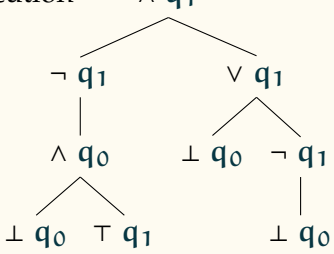
Le résultat final est lu le long des feuilles de l'arbre de dérivation; $\mathcal{R}^{-1}(\mathcal{T})$ est optimisé en $Y^{-1}(\mathcal{T})$ dans une phase subséquente. Là encore, sans rentrer dans les détails et essayer de décrypter toute l'opération, notons simplement la substitution sur Π dans la première étape, et le stockage de l'information X dans la signature lors de la seconde étape, information restituée lors de l'avant-dernière étape; notons enfin que la dernière étape est bien la traduction de l'atome Y, avec $X(\mathcal{R}^*(\Pi))$ remplaçant Π. D'autres exemples de traductions apparaissent en section 4.5[p97].

Écrire ce genre de dérivation à la main n'est pas nécessairement plus agréable que de deviner la traduction et la prouver a posteriori, mais ce procédé a l'avantage d'être entièrement mécanique et programmable, ce qui était notre objectif. De plus il est efficace: dans la plateforme de vérification, le coût algorithmique de la traduction en proposition de réécriture restera négligeable devant celui des algorithmes sur les automates d'arbres avec ou même sans contraintes. La dernière question porte donc sur le fragment de LTL traduisible par ce procédé: est-il suffisant pour les applications ? Ceci est traité en section 4.5[p97], avec des résultats encourageants.

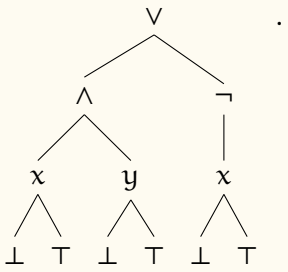
12.2 Problèmes de décisions pour automates à contraintes

Les automates d'arbres à contraintes globales d'égalité (TAGE) sont un composant essentiel de notre plateforme de vérification, car ils permettent d'en étendre nettement la précision. Un TAGE est un automate d'arbres capable d'exprimer des contraintes de la forme $p \cong q$, où p et q sont des états, qui imposent qu'au cours de l'exécution de l'automate, les états p et q ne puissent reconnaître que des sous-termes égaux. Il est bien connu et facile de voir que le problème d'appartenance des TAGE est NP-complet. Rappelons qu'il est trivial d'encoder les règles de la logique propositionnelle sans variables dans un automate d'arbres classique, en utilisant un état par valeur de vérité. Par exemple, si q_1 – le sous-terme est évalué

à "vrai" – est l'état final,

t =  est accepté par l'exécution  .

Mais les automates classiques ne peuvent pas encoder les variables propositionnelles: les sous-termes sont évalués indépendamment les uns des autres, et il n'est donc pas possible de s'assurer qu'une même variable x soit évaluée de façon non-déterministe de la même manière en ses multiples occurrences. Les contraintes globales permettent justement de réaliser ce genre de codage; considérons l'arbre suivant, représentant $(x \wedge y) \vee \neg x$:

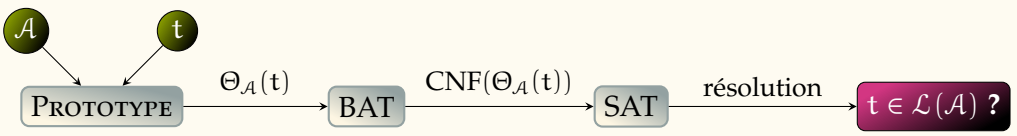
 .

Avec les règles suivantes pour chaque variables x, où ν_x est un état soumis à la contrainte $\nu_x \cong \nu_x$, l'automate à contraintes assure que chaque sous-arbre $x(\bot, \top)$ soit évalué de la même manière:

$$\top \to \nu_x,\ \bot \to \nu_x,\ x(q_0, \nu_x) \to q_1,\ x(\nu_x, q_1) \to q_0\ .$$

Une formule de logique propositionnelle est donc satisfaisable si et seulement s'il existe une exécution acceptante de cet automate. Comme il est également facile de vérifier qu'une exécution satisfait les contraintes, le problème d'appartenance des TAGE est NP-complet. D'une façon générale, les problèmes de décision pour les TAGE ont des complexités très élevées: les tests de vacuité et de finitude sont ExpTime-complets, par exemple.

Dans la partie III[p107], après un survol assez exhaustif de l'histoire et la taxonomie des automates à contraintes – car les TAGE ne sont pas les seuls membres de cette famille, – nous étudions ces problèmes de décision dans l'optique de raffiner les résultats de complexité ou d'obtenir des procédures de décision efficaces en pratique. Le problème d'appartenance étant NP-complet, une façon viable d'aborder sa décision est de le réduire au problème SAT, car les algorithmes de résolution de ce problème – ainsi d'ailleurs que leurs implémentations – sont maintenant extrêmement optimisés et efficaces. Nous effectuons donc un tel codage en chapitre 7[p129], que nous implémentons et testons avec le processus suivant:



.

Étant donné un TAGE et un arbre, notre prototype produit la formule propositionnelle $\Theta_{\mathcal{A}}(t)$, qui est mise en forme normale conjonctive par un outil externe (BAT) qui évite l'explosion d'une transformation naïve utilisant les lois de De Morgan. La satisfaisabilité de la formule résultante est enfin testée en utilisant deux programmes différents. Les résultats expérimentaux montrent une résolution SAT rapide malgré la taille des formules; le goulet d'étranglement dans nos tests était la conversion en CNF avec l'outil externe, non optimisé.

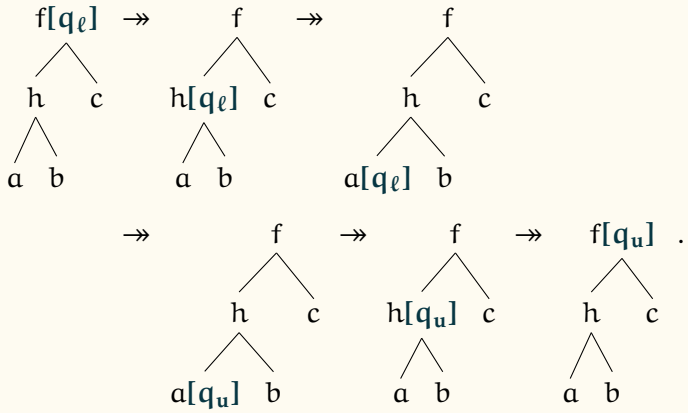
Nous attaquons également les problèmes de décision d'un autre point de vue, qui est celui du nombre de contraintes que l'automate peut exprimer simultanément. Il est connu que se restreindre à certains types de contraintes peut influencer la complexité algorithmique; en particulier, les contraintes dites *d'états rigides*, c'est-à-dire de la forme $q \cong q$ offrent le même pouvoir d'expression que les contraintes générales mais le problème de vacuité devient décidable en temps linéaire. Bien entendu, cela est au prix d'une perte de concision exponentielle, mais en pratique beaucoup de problèmes s'expriment directement et naturellement au moyen d'états rigides – c'est par exemple le cas de l'exemple de SAT vu plus haut.

La question que nous nous posons en chapitre 6[p117] est orthogonale à la restriction du type des contraintes: que se passe-t-il si l'on borne le nombre de contraintes, c'est à dire le nombre de couples $p \cong q$? Nous montrons que chaque contrainte ajoute du pouvoir d'expression – strictement. En d'autres termes, la classe des TAGE restreints à k contraintes permet toujours de reconnaître des langages inexprimables avec $k-1$ contraintes. Du point de vue de la complexité, nous montrons que deux contraintes suffisent pour rendre les problèmes de finitude et vacuité ExpTime-complets. En revanche, le problème d'appartenance devient polynômial dès lors que la borne k est considérée constante, et ce quel que soit k. Ceci suggère la pertinence d'une étude de la complexité paramétrée du problème, et l'espoir d'obtenir des méthodes de décision très efficaces pour de petits nombres de contraintes.

12.3 Problèmes de décision pour les automates cheminants

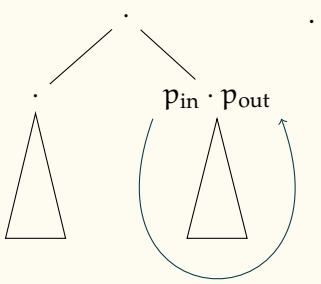
La partie IV[p143] s'ouvre sur un survol des automates d'arbres et de leurs applications à XML; en particulier, les familles d'automates d'arbres cheminants (TWA) et leurs connexions avec les langages de navigation et de requêtes dans les bases de données semi-structurées sont mises en avant. Brièvement, les automates d'arbres cheminants peuvent être vus comme une tête de lecture parcourant les nœuds du graphe d'un arbre binaire de proche en proche, et prenant ses décisions – mouvement et changement d'état – en fonction de l'étiquette du noeud courant et de son type – racine, fils gauche ou fils droit. Une exécution est acceptante si l'automate commence à la racine en un état initial et y retourne en un état final. Par exemple, voici l'exécution d'un TWA dont l'état initial q_ℓ descend toujours à gauche et, s'il atteint une feuille étiquetée par a, passe dans l'état final q_u, qui ne

fait que remonter:



Cette exécution est acceptante, et les règles énoncées ci-dessus décrivent un automate qui accepte exactement les arbres dont la feuille la plus à gauche est étiquetée par a – si ce n'est pas le cas, l'exécution s'arrête en q_ℓ sur la feuille. Les TWA sont connus comme étant strictement moins puissants que les automates ascendants, mais plus compacts sur certains langages, en particulier lorsqu'il n'est pas besoin de parcourir tous les nœuds de l'arbre afin de décider s'il doit être accepté.

Nos contributions – chapitre 9[p165] – sont centrées autour de l'algorithme qui transforme un TWA en un automate ascendant équivalent. L'idée générale de cet algorithme est connue et repose sur le concept de *boucle d'arbre*, c'est à dire de couples d'états (p_{in}, p_{out}) dans lesquels l'automate entre, puis sort d'un sous-arbre donné.



Les états résultant de la transformation sont – entre autres – des ensembles de tels couples d'états. Cet algorithme n'était pas explicite dans la littérature, et les complexités annoncées omettaient en particulier un facteur $|\mathbb{A}|$, la taille de l'alphabet, qui peut être considérable dans les applications. Après une étude formelle des boucles – qui offre au passage des algorithmes efficaces pour le test d'appartenance – nous introduisons la notion de *surboucle*, qui permet d'éliminer ce facteur entièrement. Nous montrons également que la transformation fondée sur les surboucles a une bien meilleure complexité que celle fondée sur les boucles dans le cas où le TWA est déterministe: x étant le nombre d'états, le gain correspond au passage de 2^{x^2} à $2^{x \log x}$.

En utilisant ces notions, nous proposons également un algorithme polynomial de test approximé – répondant oui ou peut-être – de la vacuité du langage accepté par un TWA, qui est un problème ExpTime-complet. Tous ces algorithmes ont été testés au moyen d'une génération aléatoire de TWA déterministes complets, montrant un très clair avantage pour la transformation par surboucles, et une précision remarquable pour la procédure approximée de décision du vide.

Bibliography

Abdulla, P. A., Jonsson, B., Mahata, P., & d'Orso, J. (2002). Regular tree model checking. In [Brinksma & Larsen, 2002], (pp. 555–568).
† Cited page 16.

Abdulla, P. A., Jonsson, B., Nilsson, M., & Saksena, M. (2004). A survey of regular model checking. In Gardner, P. & Yoshida, N. (Eds.), *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, (pp. 35–48). Springer.
† Cited page 15.

Abiteboul, S., Buneman, P., & Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann.
† Cited page 160.

Aggarwal, S., Kurshan, R. P., & Sabnani, K. K. (1983). A calculus for protocol specification and validation. In *Protocol Specification, Testing, and Verification*, (pp. 19–34).
† Cited page 12.

Aho, A. & Ullman, J. (1969). Translations on a context free grammar. *Information and Control*, *19*(5), 439–475.
† Cited 4 times, page 163.

Apt, K. & Kozen, D. (1986). Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, *22*, 307–309.
† Cited page 14.

Armando, A., Basin, D. A., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., & Vigneron, L. (2005). The avispa tool for the automated validation of internet security protocols and applications. In Etessami, K. & Rajamani, S. K. (Eds.), *CAV*, volume 3576 of *Lecture Notes in Computer Science*, (pp. 281–285). Springer.
† Cited page 43.

Baader, F. (Ed.). (2007). *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*.
† Cited twice, pages 211 and 216.

Baader, F. & Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press.
† Cited twice, pages 22 and 30.

Bae, K. & Meseguer, J. (2010). The linear temporal logic of rewriting Maude model checker. In [Ölveczky, 2010], (pp. 208–225).
† Cited twice, pages 43 and 44.

Barguñó, L., Creus, C., Godoy, G., Jacquemard, F., & Vacher, C. (2010). The emptiness problem for tree automata with global constraints. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science (LICS'10)*, (pp. 263–272)., Edinburgh, Scotland, UK. IEEE Computer Society Press.
† Cited thrice, pages 112 and 113.

Barré, N., Hubert, L., Roux, L. L., & Genet, T. (2009). `Copster` homepage. `http://www.irisa.fr/celtique/genet/copster`.
† Cited twice, pages 43 and 103.

Bassino, F., David, J., & Nicaud, C. (2007). `REGAL` : A library to randomly and exhaustively generate automata. In *CIAA, LNCS 4783*, (pp. 303–305).
† Cited page 181.

Benedikt, M. & Koch, C. (2008). XPath leashed. *ACM Comput. Surv.*, *41*(1).
† Cited page 159.

Biere, A. (2008). Picosat essentials. *JSAT*, *4*(2-4), 75–97.
† Cited page 129.

Biere, A., Cimatti, A., Clarke, E. M., & Zhu, Y. (1999). Symbolic model checking without BDDs. In Cleaveland, R. (Ed.), *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, (pp. 193–207). Springer.
† Cited page 13.

Bloem, R. & Engelfriet, J. (2000). A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *J. Comput. Syst. Sci.*, *61*(1), 1–50.
† Cited page 163.

Blum, M. & Kozen, D. (1978). On the power of the compass (or, why mazes are easier to search than graphs). In [Unknown, 1978], (pp. 132–142).
† Cited page 196.

Bogaert, B. & Tison, S. (1992). Equality and disequality constraints on direct subterms in tree automata. In Finkel, A. & Jantzen, M. (Eds.), *STACS*, volume 577 of *Lecture Notes in Computer Science*, (pp. 161–171). Springer.
† Cited page 109.

Boichut, Y., Courbis, R., Héam, P.-C., & Kouchnarenko, O. (2009). Handling non left-linear rules when completing tree automata. *Int. J. Found. Comput. Sci.*, *20*(5), 837–849.
† Cited page 50.

Boichut, Y., Genet, T., Jensen, T. P., & Roux, L. L. (2007). Rewriting approximations for fast prototyping of static analyzers. In [Baader, 2007], (pp. 48–62).
† Cited 4 times, pages 43, 97, and 103.

Boichut, Y., Héam, P.-C., & Kouchnarenko, O. (2006). Handling algebraic properties in automatic analysis of security protocols. In Barkaoui, K., Cavalcanti, A., & Cerone, A. (Eds.), *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, (pp. 153–167). Springer.
† Cited page 50.

Boichut, Y., Héam, P.-C., & Kouchnarenko, O. (2008). Approximation-based tree regular model-checking. *Nord. J. Comput.*, *14*(3), 216–241.
† Cited thrice, pages 46 and 50.

Bojańczyk, M. (2003). 1-bounded TWA cannot be determinized. *FSTTCS'03*, *LNCS*, *2914*, 62–73.
† Cited page 163.

Bojańczyk, M. (2008). Tree-Walking Automata. *LATA'08 (tutorial)*, *LNCS*, *5196*.
† Cited 15 times, pages 143, 144, 148, 160, 161, 162, 166, 175, 176, 179, 197, and 198.

Bojańczyk, M. & Colcombet, T. (2005). Tree-walking automata do not recognize all regular languages. STOC '05, (pp. 234–243). ACM.
† Cited twice, pages 161 and 163.

Bojańczyk, M. & Colcombet, T. (2006). Tree-walking automata cannot be determinized. *Theoretical Computer Science*, *350*(2-3), 164–173.
† Cited page 163.

Bojanczyk, M., Muscholl, A., Schwentick, T., & Segoufin, L. (2009). Two-variable logic on data trees and XML reasoning. *J. ACM*, *56*(3).
† Cited page 112.

Bojanczyk, M., Samuelides, M., Schwentick, T., & Segoufin, L. (2006). Expressive power of pebble automata. In Bugliesi, M., Preneel, B., Sassone, V., & Wegener, I. (Eds.), *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, (pp. 157–168). Springer.
† Cited page 197.

Boronat, A., Heckel, R., & Meseguer, J. (2009). Rewriting logic semantics and verification of model transformations. In *FASE*, volume 5503 of *Lecture Notes in Computer Science*, (pp. 18–33). Springer.
† Cited page 43.

Bouajjani, A. & Touili, T. (2002). Extrapolating tree transformations. In Brinksma, E. & Larsen, K. G. (Eds.), *Computer Aided Verification, CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, (pp. 539–554). Springer-Verlag.
† Cited page 45.

Boyer, B. & Genet, T. (2009). Verifying Temporal Regular Properties of Abstractions of Term Rewriting Systems. In *RULE*, volume 21 of *EPTCS*, (pp. 99–108).
† Cited twice, page 43.

Brainerd, W. S. (1969). Tree generating regular systems. *Information and Control*, *14*(2), 217–231.
† Cited page 45.

Brinksma, E. & Larsen, K. G. (Eds.). (2002). *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer.
† Cited twice, pages 210 and 213.

Bruggemann-Klein, A., Murata, M., & Wood, D. (2001). Regular tree and regular hedge languages over unranked alphabets. Technical report.
† Cited page 149.

Brüggemann-Klein, A. & Wood, D. (2000). Caterpillars: A context specification technique. *Markup Languages*, *2*(1), 81–106.
† Cited 4 times, pages 160, 161, and 163.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., & Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, *98*(2), 142–170.
† Cited page 12.

Carme, J., Niehren, J., & Tommasi, M. (2004). Querying unranked trees with stepwise tree automata. In [van Oostrom, 2004], (pp. 105–118).
† Cited page 154.

Caron, A.-C., Comon, H., Coquidé, J.-L., Dauchet, M., & Jacquemard, F. (1994). Pumping, cleaning and symbolic constraints solving. In Abiteboul, S. & Shamir, E. (Eds.), *ICALP*, volume 820 of *Lecture Notes in Computer Science*, (pp. 436–449). Springer.
† Cited page 111.

Castéran, P., Herbelin, H., Kirchner, F., Monate, B., & Narboux, J. (2012). Coq version 8.4 for the clueless. `http://coq.inria.fr/faq`.
† Cited page 188.

Chambers, B., Manolios, P., & Vroon, D. (2009). Faster sat solving with better cnf generation. In *DATE*, (pp. 1590–1595). IEEE.
† Cited page 137.

Charatonik, W. (1999). Automata on DAG representations of finite trees. *Tech. Report*, (-).
† Cited page 191.

Chevalier, Y. & Vigneron, L. (2002). Automated unbounded verification of security protocols. In [Brinksma & Larsen, 2002], (pp. 324–337).
† Cited page 43.

Clarke, E. M. (2008). The birth of model checking. In Grumberg, O. & Veith, H. (Eds.), *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, (pp. 1–26). Springer.
† Cited page 13.

Clarke, E. M., Biere, A., Raimi, R., & Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*, *19*(1), 7–34.
† Cited page 130.

Clarke, E. M. & Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen, D. (Ed.), *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, (pp. 52–71). Springer.
† Cited 4 times, page 11.

Clavel, M., Palomino, M., & Riesco, A. (2006). Introducing the itp tool: a tutorial. *J. Univ. Comp. Sci.*, *12*(11), 1618–1650.
† Cited page 43.

Comon, H. & Cortier, V. (2005). Tree automata with one memory set constraints and cryptographic protocols. *Theoretical Computer Science*, *331*(1), 143–214.
† Cited 8 times, pages 193, 194, and 195.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., & Tommasi, M. (2008). *Tree Automata Techniques and Applications*. release November 18, 2008.
† Cited 16 times, pages 22, 35, 38, 48, 50, 107, 108, 109, 110, 111, 143, 150, 182, and 183.

Comon, H., Jacquemard, F., & Perrin, N. (2008). Visibly tree automata with memory and constraints. *Logical Methods in Computer Science*, *4*(2).
† Cited 9 times, pages 193, 194, and 195.

Consortium, W. W. W. (1999). XML path language. `http://www.w3.org/TR/xpath/`.
† Cited twice, page 157.

Consortium, W. W. W. (2010). XML path language 2.0. `http://www.w3.org/TR/xpath20/`.
† Cited twice, page 157.

Coppersmith, D. & Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, *9*(3), 251–280.
† Cited page 171.

Coquidé, J.-L., Dauchet, M., Gilleron, R., & Vágvölgyi, S. (1991). Bottom-up tree pushdown automata and rewrite systems. In Book, R. V. (Ed.), *RTA*, volume 488 of *Lecture Notes in Computer Science*, (pp. 287–298). Springer.
† Cited page 45.

Cosmadakis, S., Gaifman, H., Kanellakis, P., & Vardi, M. (1988). Decidable optimization problems for database logic programs. STOC '88, (pp. 477–490). ACM.
† Cited page 174.

Courbis, R. (2011). *Contributions à l'analyse de systèmes par approximation d'ensembles réguliers*. Thèse de Doctorat, LIFC, Université de Franche-Comté.
† Cited 4 times, pages 43, 97, and 103.

Courbis, R., Héam, P.-C., & Kouchnarenko, O. (2009). TAGED Approximations for Temporal Properties Model-Checking. In [Maneth, 2009], (pp. 135–144).
† Cited 26 times, pages 19, 41, 42, 43, 51, 53, 54, 57, 88, 89, 91, 94, 97, 187, 200, 203, and 204.

Courcelle, B. (1990). Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)* (pp. 193–242).
† Cited page 157.

Cousot, P. (2002). Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, *277*(1-2), 47–103.
† Cited page 57.

Dauchet, M., Caron, A.-C., & Coquidé, J.-L. (1995). Automata for reduction properties solving. *J. Symb. Comput.*, *20*(2), 215–233.
† Cited thrice, page 110.

Dauchet, M. & Tison, S. (1990). The theory of ground rewrite systems is decidable. In *LICS*, (pp. 242–248). IEEE Computer Society.
† Cited page 45.

Deransart, P., Jourdan, M., & Lorho, B. (1988). *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer.
† Cited page 163.

Dershowitz, N. & Jouannaud, J.-P. (1990). Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)* (pp. 243–320).
† Cited twice, pages 22 and 30.

Dwyer, M., Avrunin, G., & Corbett, J. (1998). Property specification patterns for finite-state verification. In *FMSP'98*, (pp. 7–15). ACM.
† Cited page 101.

Dwyer, M., Avrunin, G., & Corbett, J. (1999). Patterns in property specifications for finite-state verification. In *ICSE'99*, (pp. 411–420). IEEE.
† Cited 5 times, pages 6, 97, 101, and 102.

Eén, N. & Sörensson, N. (2003). An extensible sat-solver. In Giunchiglia, E. & Tacchella, A. (Eds.), *SAT*, volume 2919 of *Lecture Notes in Computer Science*, (pp. 502–518). Springer.
† Cited page 129.

Eker, S., Meseguer, J., & Sridharanarayanan, A. (2003). The Maude LTL model checker and its implementation. In Ball, T. & Rajamani, S. K. (Eds.), *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, (pp. 230–234). Springer.
† Cited twice, page 43.

Emerson, E. A. & Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In de Bakker, J. W. & van Leeuwen, J. (Eds.), *ICALP*, volume 85 of *Lecture Notes in Computer Science*, (pp. 169–181). Springer.
† Cited page 11.

Engelfriet, J. & Hoogeboom, H. J. (1999). Tree-walking pebble automata. In Karhumäki, J., Maurer, H. A., Păun, G., & Rozenberg, G. (Eds.), *Jewels are Forever*, (pp. 72–83). Springer.
† Cited 4 times, pages 163, 196, 197, and 199.

Engelfriet, J. & Hoogeboom, H. J. (2007). Automata with nested pebbles capture first-order logic with transitive closure. *Logical Methods in Computer Science*, *3*(2).
† Cited twice, page 197.

Engelfriet, J., Hoogeboom, H. J., & Best, J.-P. V. (1999). Trips on trees. *Acta Cybern.*, *14*(1), 51–64.
† Cited page 198.

Engelfriet, J., Hoogeboom, H. J., & Samwel, B. (2007). XML transformation by tree-walking transducers with invisible pebbles. In Libkin, L. (Ed.), *PODS*, (pp.

63–72). ACM.
† Cited page 197.

Engelfriet, J., Rozenberg, G., & Slutzki, G. (1980). Tree transducers, L systems, and two-way machines. *J. Comput. Syst. Sci.*, *20*(2), 150–202.
† Cited page 198.

Escobar, S. & Meseguer, J. (2007). Symbolic model checking of infinite-state systems using narrowing. In [Baader, 2007], (pp. 153–168).
† Cited twice, pages 43 and 44.

Fagin, R. (1975). Monadic generalized spectra. *Mathematical Logic Quarterly*, *21*(1), 89–96.
† Cited page 159.

Feuillade, G., Genet, T., & Tong, V. V. T. (2004). Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, *33*(3-4), 341–383.
† Cited 5 times, pages 43, 45, 46, 137, and 139.

Filiot, E. (2008). *Logics for n-ary queries in trees.* PhD thesis, Université des Sciences et Technologie de Lille - Lille I.
† Cited 4 times, pages 35, 112, 118, and 143.

Filiot, E., Talbot, J.-M., & Tison, S. (2007). Satisfiability of a spatial logic with tree variables. In Duparc, J. & Henzinger, T. A. (Eds.), *CSL*, volume 4646 of *Lecture Notes in Computer Science*, (pp. 130–145). Springer.
† Cited page 112.

Filiot, E., Talbot, J.-M., & Tison, S. (2008). Tree automata with global constraints. In *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, (pp. 314–326). Springer.
† Cited 8 times, pages 35, 88, 92, 110, 112, 113, 120, and 134.

Filiot, E., Talbot, J.-M., & Tison, S. (2010). Tree automata with global constraints. *Int. J. Found. Comput. Sci.*, *21*(4), 571–596.
† Cited 6 times, pages 35, 107, 109, 112, 121, and 134.

Finkel, A. & Goubault-Larrecq, J. (2012). The theory of wsts: The case of complete wsts. In Haddad, S. & Pomello, L. (Eds.), *Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, (pp. 3–31). Springer.
† Cited page 16.

Finkel, A. & Schnoebelen, P. (2001). Well-structured transition systems everywhere! *Theoretical Computer Science*, *256*(1-2), 63–92.
† Cited page 16.

Fisman, D. & Pnueli, A. (2001). Beyond regular model checking. In Hariharan, R., Mukund, M., & Vinay, V. (Eds.), *FSTTCS*, volume 2245 of *Lecture Notes in Computer Science*, (pp. 156–170). Springer.
† Cited page 15.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, *5*(6), 345–.
† Cited page 172.

Frick, M. & Grohe, M. (2002). The complexity of first-order and monadic second-order logic revisited. In *Logic in Computer Science, 2002. Proceedings. 17th Annual*

IEEE Symposium on, (pp. 215–224). IEEE.
† Cited page 156.

Gabbay, D. M. (1987). The declarative past and imperative future: Executable temporal logic for interactive systems. In Banieqbal, B., Barringer, H., & Pnueli, A. (Eds.), *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, (pp. 409–448). Springer.
† Cited page 189.

Gabbay, D. M., Pnueli, A., Shelah, S., & Stavi, J. (1980). On the temporal basis of fairness. In Abrahams, P. W., Lipton, R. J., & Bourne, S. R. (Eds.), *POPL*, (pp. 163–173). ACM Press.
† Cited page 189.

Genet, T. (1998). Decidable approximations of sets of descendants and sets of normal forms. In Nipkow, T. (Ed.), *RTA*, volume 1379 of *Lecture Notes in Computer Science*, (pp. 151–165). Springer.
† Cited twice, pages 43 and 46.

Genet, T. (2009). *Reachability analysis of rewriting for software verification*. Habilitation thesis (habilitation à diriger des recherches), University of Rennes I.
† Cited 11 times, pages 43, 45, 46, 47, 48, 49, and 50.

Genet, T. & Klay, F. (2000). Rewriting for cryptographic protocol verification. In McAllester, D. (Ed.), *CADE*, volume 1831 of *Lecture Notes in Computer Science*, (pp. 271–290). Springer.
† Cited 6 times, pages 43, 53, 97, 103, and 204.

Genet, T. & Rusu, V. (2010). Equational approximations for tree automata completion. *J. Symb. Comput.*, *45*(5), 574–597.
† Cited twice, pages 46 and 48.

Gilleron, R. & Tison, S. (1995). Regular tree languages and rewrite systems. *Fundam. Inform.*, *24*(1/2), 157–174.
† Cited page 44.

Godoy, G., Giménez, O., Ramos, L., & Àlvarez, C. (2010). The hom problem is decidable. In Schulman, L. J. (Ed.), *STOC*, (pp. 485–494). ACM.
† Cited page 109.

Goris, E. & Marx, M. (2005). Looping caterpillars. In *LICS*, (pp. 51–60). IEEE Computer Society.
† Cited thrice, pages 161 and 162.

Gottlob, G. & Koch, C. (2004). Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, *51*(1), 74–113.
† Cited page 157.

Gottlob, G., Koch, C., & Pichler, R. (2002). Efficient algorithms for processing XPath queries. In *VLDB*, (pp. 95–106). Morgan Kaufmann.
† Cited page 158.

Gottlob, G., Koch, C., & Pichler, R. (2005). Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, *30*(2), 444–491.
† Cited twice, page 158.

Guessarian, I. (1981). On pushdown tree automata. In Astesiano, E. & Böhm, C. (Eds.), *CAAP*, volume 112 of *Lecture Notes in Computer Science*, (pp. 211–223). Springer.
† Cited page 193.

Guessarian, I. (1983). Pushdown tree automata. *Mathematical Systems Theory*, *16*(4), 237–263.
† Cited page 193.

Gyenizse, P. & Vágvölgyi, S. (1998). Linear generalized semi-monadic rewrite systems effectively preserve recognizability. *Theoretical Computer Science*, *194*(1-2), 87–122.
† Cited twice, page 45.

Héam, P., Hugot, V., & Kouchnarenko, O. (2010a). Random Generation of Positive TAGEDs wrt. the Emptiness Problem. Technical Report RR-7441, INRIA.
† Cited 5 times, pages 20, 167, 182, 183, and 201.

Héam, P., Hugot, V., & Kouchnarenko, O. (2010b). SAT Solvers for Queries over Tree Automata with Constraints. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, (pp. 343–348). IEEE.
† Cited 4 times, pages 20, 128, 130, and 201.

Héam, P.-C., Hugot, V., & Kouchnarenko, O. (2011). Loops and overloops for tree walking automata. In *CIAA'11, LNCS 6807*, (pp. 166–177).
† Cited 4 times, pages 20, 165, 184, and 202.

Héam, P.-C., Hugot, V., & Kouchnarenko, O. (2012a). From linear temporal logic properties to rewrite propositions. In Gramlich, B., Miller, D., & Sattler, U. (Eds.), *IJCAR'12*, volume 7364 of *Lecture Notes in Computer Science*, (pp. 316–331). Springer.
† Cited 5 times, pages 20, 76, 78, 88, and 201.

Héam, P.-C., Hugot, V., & Kouchnarenko, O. (2012b). Loops and overloops for tree-walking automata. *Theoretical Computer Science*, *450*, 43–53.
† Cited 4 times, pages 20, 165, 184, and 202.

Héam, P.-C., Hugot, V., & Kouchnarenko, O. (2012c). On positive TAGED with a bounded number of constraints. In Moreira, N. & Reis, R. (Eds.), *CIAA*, volume 7381 of *Lecture Notes in Computer Science*, (pp. 329–336). Springer.
† Cited 4 times, pages 20, 118, 128, and 201.

Héam, P.-C., Hugot, V., & Kouchnarenko, O. (2013). Semi-deciding LTL properties over rewrite-rules sequences. *(prepublication)*, *?*
† Cited twice, pages 20 and 201.

Héam, P.-C., Nicaud, C., & Schmitz, S. (2009). Random generation of deterministic tree (walking) automata. In [Maneth, 2009], (pp. 115–124).
† Cited twice, pages 166 and 181.

Heen, O., Genet, T., Geller, S., & Prigent, N. (2008). An industrial and academic joint experiment on automated verification of a security protocol. In *IFIP Networking Workshop on Mobile and Networks Security*.
† Cited page 43.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580.
† Cited page 10.

Hosoya, H. (2010). *Foundations of XML processing: the tree-automata approach*. Cambridge University Press.
† Cited 10 times, pages 19, 35, 143, 148, 151, 152, 160, 161, 163, and 199.

Jacquemard, F. (1996). Decidable approximations of term rewriting systems. In Ganzinger, H. (Ed.), *RTA*, volume 1103 of *Lecture Notes in Computer Science*, (pp. 362–376). Springer.
† Cited page 45.

Jacquemard, F., Klay, F., & Vacher, C. (2009). Rigid tree automata. In Horia Dediu, A., Mihai Ionescu, A., & Martín-Vide, C. (Eds.), *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA'09)*, volume 5457 of *Lecture Notes in Computer Science*, (pp. 446–457)., Tarragona, Spain. Springer.
† Cited page 113.

Jacquemard, F., Klay, F., & Vacher, C. (2011). Rigid tree automata and applications. *Inf. Comput.*, *209*(3), 486–512.
† Cited page 113.

Jacquemard, F., Rusinowitch, M., & Vigneron, L. (2008). Tree automata with equality constraints modulo equational theories. *J. Log. Algebr. Program.*, *75*(2), 182–208.
† Cited thrice, pages 110 and 111.

Jones, N. D. (1987). Flow analysis of lazy higher-order functional programs. (pp. 103–122). Halsted Press.
† Cited page 42.

Jones, N. D. & Andersen, N. (2007). Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, *375*(1-3), 120–136.
† Cited page 42.

Kamimura, T. & Slutzki, G. (1981). Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, *49*(1), 10–51.
† Cited thrice, pages 144, 163, and 198.

Kamp, H. (1968). *Tense Logic and the Theory of Linear Order*.
† Cited page 57.

Kesten, Y., Maler, O., Marcus, M., Pnueli, A., & Shahar, E. (1997). Symbolic model checking with rich assertional languages. In Grumberg, O. (Ed.), *CAV*, volume 1254 of *Lecture Notes in Computer Science*, (pp. 424–435). Springer.
† Cited thrice, pages 13, 16, and 17.

Kirchner, C. & Kirchner, H. (1996). *Rewriting, Solving, Proving*.
† Cited twice, pages 22 and 30.

Klaedtke, F. & Rueß, H. (2002). Parikh automata and monadic second-order logics with linear cardinality constraints. *Tech. Report*, (-).
† Cited page 112.

Knuth, D. (1968). Semantics of context-free languages. *Theory of Computing Systems*, 2(2), 127–145.
† Cited page 166.

Knuth, D. E. (1997). *Fundamental Algorithms* (Third ed.)., volume 1 of *The Art of Computer Programming*. Addison-Wesley Professional.
† Cited page 152.

Knuth, D. E. & Bendix, P. B. (1970). Simple word problems in universal algebras. In *Computational problems in abstract algebra*, (pp. 263–297). Oxford, Pergamon Press.
† Cited twice, page 46.

Ladner, R. E., Lipton, R. J., & Stockmeyer, L. J. (1984). Alternating pushdown and stack automata. *SIAM J. Comput.*, *13*(1), 135–155.
† Cited page 150.

Laroussinie, F., Markey, N., & Schnoebelen, P. (2002). Temporal logic with forgettable past. In *LICS*, (pp. 383–392). IEEE Computer Society.
† Cited page 189.

Maneth, S. (Ed.). (2009). *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, volume 5642 of *Lecture Notes in Computer Science*. Springer.
† Cited twice, pages 214 and 218.

Manna, Z. & Pnueli, A. (1995). *Temporal Verification of Reactive Systems - Safety*. Springer.
† Cited page 57.

Manolios, P., Srinivasan, S. K., & Vroon, D. (2007). Bat: The bit-level analysis tool. In Damm, W. & Hermanns, H. (Eds.), *CAV*, volume 4590 of *Lecture Notes in Computer Science*, (pp. 303–306). Springer.
† Cited page 137.

Martens, W. & Neven, F. (2003). Typechecking top-down uniform unranked tree transducers. In Calvanese, D., Lenzerini, M., & Motwani, R. (Eds.), *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, (pp. 64–78). Springer.
† Cited page 150.

Martens, W. & Niehren, J. (2005). Minimizing tree automata for unranked trees. In Bierman, G. M. & Koch, C. (Eds.), *DBPL*, volume 3774 of *Lecture Notes in Computer Science*, (pp. 232–246). Springer.
† Cited twice, pages 151 and 154.

Martí-Oliet, N. & Meseguer, J. (1996). Rewriting logic as a logical and semantic framework. *Electr. Notes Theor. Comput. Sci.*, *4*, 190–225.
† Cited page 43.

Martí-Oliet, N. & Meseguer, J. (2002). Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, *285*(2), 121–154.
† Cited page 43.

Marx, M. & de Rijke, M. (2005). Semantic characterizations of navigational XPath. *SIGMOD Record*, *34*(2), 41–46.
† Cited page 159.

Meseguer, J. (1992). Conditioned rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, *96*(1), 73–155.
† Cited thrice, page 43.

Meseguer, J. (2008). The temporal logic of rewriting: A gentle introduction. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*. Springer.
† Cited page 43.

Milo, T., Suciu, D., & Vianu, V. (2003). Typechecking for XML transformers. *J. Comput. Syst. Sci.*, *66*(1), 66–97.
† Cited page 163.

Mongy, J. (1981). *Transformation de noyaux reconnaissables d'arbres. Forêts RATEG*. Ph.d. thesis, LIF de Lille, Université des Sciences et Technologies de Lille, Lille, France.
† Cited page 108.

Muscholl, A., Samuelides, M., & Segoufin, L. (2006). Complementing deterministic tree-walking automata. *Inf. Process. Lett.*, *99*(1), 33–39.
† Cited twice, pages 163 and 197.

Nagaya, T. & Toyama, Y. (1999). Decidability for left-linear growing term rewriting systems. In Narendran, P. & Rusinowitch, M. (Eds.), *RTA*, volume 1631 of *Lecture Notes in Computer Science*, (pp. 256–270). Springer.
† Cited page 45.

Nagaya, T. & Toyama, Y. (2002). Decidability for left-linear growing term rewriting systems. *Inf. Comput.*, *178*(2), 499–514.
† Cited page 45.

Neven, F. (1999). Extensions of attribute grammars for structured document queries. In Connor, R. C. H. & Mendelzon, A. O. (Eds.), *DBPL*, volume 1949 of *Lecture Notes in Computer Science*, (pp. 99–116). Springer.
† Cited page 164.

Neven, F. (2002). Automata, logic, and XML. In Bradfield, J. C. (Ed.), *CSL*, volume 2471 of *Lecture Notes in Computer Science*, (pp. 2–26). Springer.
† Cited 7 times, pages 150, 152, 153, 164, and 197.

Neven, F. & Schwentick, T. (2000). Expressive and efficient pattern languages for tree-structured data. In Vianu, V. & Gottlob, G. (Eds.), *PODS*, (pp. 145–156). ACM.
† Cited page 157.

Ölveczky, P. C. (Ed.). (2010). *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*. Springer.
† Cited twice, pages 43 and 210.

Otto, F. (1998). Some undecidability results concerning the property of preserving regularity. *Theoretical Computer Science*, *207*(1), 43–72.
† Cited page 45.

Paulson, L. C. (1989). The foundation of a generic theorem prover. *J. Autom. Reasoning*, *5*(3), 363–397.
† Cited page 188.

Pnueli, A. (1977). The temporal logic of programs. In *FOCS'77*, (pp. 46–57).
† Cited twice, pages 11 and 12.

Queille, J.-P. & Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini, M. & Montanari, U. (Eds.), *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, (pp. 337–351). Springer.
† Cited twice, page 11.

Rabin, M. O. (1969). Decidability of second-order theories and automata on infinite trees. *Trans. Am. Math. Soc.*, *141*, 1–35.
† Cited page 152.

Réty, P. (1999). Regular sets of descendants for constructor-based rewrite systems. In Ganzinger, H., McAllester, D. A., & Voronkov, A. (Eds.), *LPAR*, volume 1705 of *Lecture Notes in Computer Science*, (pp. 148–160). Springer.
† Cited page 45.

Roy, B. (1959). Transitivité et connexité. *CR Acad. Sci. Paris*, *249*, 216–218.
† Cited page 172.

Salomaa, K. (1988). Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. Syst. Sci.*, *37*(3), 367–394.
† Cited page 45.

Salomaa, K., Yu, S., & Zan, J. (2007). Deterministic caterpillar expressions. In Holub, J. & Zdárek, J. (Eds.), *CIAA*, volume 4783 of *Lecture Notes in Computer Science*, (pp. 97–108). Springer.
† Cited page 160.

Salomaa, K., Yu, S., & Zan, J. (2009). Deciding determinism of caterpillar expressions. *Theoretical Computer Science*, *410*(37), 3438–3446.
† Cited page 160.

Samuelides, M. (2007). *Automates d'arbres à jetons*. PhD thesis, Université Paris-Diderot - Paris VII.
† Cited 5 times, pages 166, 175, 176, and 197.

Samuelides, M. & Segoufin, L. (2007). Complexity of pebble tree-walking automata. In Csuhaj-Varjú, E. & Ésik, Z. (Eds.), *FCT*, volume 4639 of *Lecture Notes in Computer Science*, (pp. 458–469). Springer.
† Cited page 197.

Segoufin, L. & Vianu, V. (2002). Validating Streaming XML Documents. In *PODS*, (pp. 53–64). ACM.
† Cited page 163.

Seki, H., Takai, T., Fujinaka, Y., & Kaji, Y. (2002). Layered transducing term rewriting system and its recognizability preserving property. In Tison, S. (Ed.), *RTA*, volume 2378 of *Lecture Notes in Computer Science*, (pp. 98–113). Springer.
† Cited page 45.

Serbanuta, T.-F., Rosu, G., & Meseguer, J. (2009). A rewriting logic approach to operational semantics. *Inf. Comput.*, *207*(2), 305–340.
† Cited page 43.

Shepherdson, J. (1959). The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, *3*(2), 198–200.
† Cited page 166.

Sipser, M. (1978). Halting space-bounded computations. In [Unknown, 1978], (pp. 73–74).
† Cited page 163.

Slutzki, G. (1985). Alternating tree automata. *Theoretical Computer Science*, *41*, 305–318.
† Cited page 199.

Stothers, A. J. (2010). *On the complexity of matrix multiplication*. PhD thesis, The University of Edinburgh.
† Cited page 171.

Suciu, D. (2001). Typechecking for semistructured data. In Ghelli, G. & Grahne, G. (Eds.), *DBPL*, volume 2397 of *Lecture Notes in Computer Science*, (pp. 1–20). Springer.
† Cited twice, page 153.

Takahashi, M. (1975). Generalizations of regular sets and their applicatin to a study of context-free languages. *Information and Control*, *27*(1), 1–36.
† Cited thrice, pages 152, 154, and 161.

Takai, T. (2004). A verification technique using term rewriting systems and abstract interpretation. In [van Oostrom, 2004], (pp. 119–133).
† Cited twice, pages 43 and 45.

Takai, T., Kaji, Y., & Seki, H. (2000). Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In Bachmair, L. (Ed.), *RTA*, volume 1833 of *Lecture Notes in Computer Science*, (pp. 246–260). Springer.
† Cited twice, page 45.

ten Cate, B., Litak, T., & Marx, M. (2007). A complete axiomatization for core XPath 1.0. In J. van den Bussche (Ed.), *Liber Amicorum Jan Paredaens* (pp. 41–56).
† Cited page 159.

ten Cate, B. & Marx, M. (2007). Axiomatizing the logical core of XPath 2.0. In Schwentick, T. & Suciu, D. (Eds.), *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, (pp. 134–148). Springer.
† Cited twice, page 159.

ten Cate, B. & Marx, M. (2009). Axiomatizing the logical core of XPath 2.0. *Theory Comput. Syst.*, *44*(4), 561–589.
† Cited twice, page 159.

ten Cate, B. & Segoufin, L. (2010). Transitive closure logic, nested tree walking automata, and XPath. *J. ACM*, *57*(3), 251–260.
† Cited 7 times, pages 158, 159, 162, 164, and 197.

Unknown (Ed.). (1978). *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society.
† Cited twice, pages 211 and 223.

Vacher, C. (2010). *Tree automata with global constraints for the verification of security properties*. Ph.D. thesis, ENS Cachan.
† Cited 9 times, pages 107, 113, 193, 194, and 196.

van Oostrom, V. (Ed.). (2004). *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*. Springer.
† Cited twice, pages 213 and 223.

Vardi, M. Y. & Wolper, P. (1986). Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, *32*(2), 183–221.
† Cited page 12.

Warshall, S. (1962). A theorem on boolean matrices. *J. ACM*, *9*(1), 11–12.
† Cited page 172.

Williams, V. V. (2011). Breaking the Coppersmith-Winograd barrier. Unpublished manuscript.
† Cited page 171.

Wong, K. & Löding, C. (2007). Unranked tree automata with sibling equalities and disequalities. In Arge, L., Cachin, C., Jurdzinski, T., & Tarlecki, A. (Eds.), *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, (pp. 875–887). Springer.
† Cited page 110.

Index

U

V

W

Version of the document: **d361**,
dated *2013-12-30 05:08:42+01:00* ,
compiled on *December 30, 2013*.

Abstract:

Tree automata, and their applications to verification, form the common thread of this thesis. **In the first part**, we define a complete model-checking framework; the general problem which we solve is to verify that a given term rewriting system – encoding some program, circuit, protocol, or more generally any system of interest – satisfies a given specification expressed in linear temporal logic, dictating the order in which the transitions of the system may occur. Our methods are closely related to reachability analysis techniques. In a first step, translation rules supporting a fragment of LTL sufficiently expressive to describe common security properties reformulate the verification problem into an equivalent expression of propositional logic whose atoms are comparisons of languages obtained through rewriting; we call such a formula a rewrite proposition. In the second step, the rewrite proposition is given a concrete representation in terms of tree automata with or without constraints. Since the general problem is undecidable, these representations are sometimes approximations, for which we use constructions studied for reachability analysis; the end result is a set of semi-decision procedures for the general problem. **The second part** focuses on an important aspect of the automata involved: constraints. We study their role in the complexity of several decision problems, in particular when bounding the number of constraints. **Finally**, we also study the very different variety of tree-walking automata, which have tight connections with navigational languages on semi-structured documents. We improve their conversion into branching models, and develop an efficient and accurate semi-decision procedure for emptiness testing.

Keywords: Tree automata, constraints, approximations, semi-decision procedures, tree model-checking

Résumé :

Les automates d'arbres et leurs applications à la vérification forment le tronc commun de cette thèse. **Dans sa première partie**, nous définissons une plate-forme de model-checking complète; le problème général que nous résolvons est de vérifier qu'un système de réécriture – codant quelque programme, circuit, protocole, ou tout autre système à vérifier – suit une spécification donnée, exprimée en logique temporelle linéaire, imposant l'ordre dans lequel les transitions du système doivent s'enchaîner. Nos méthodes se rapprochent fortement des techniques d'analyse d'accessibilité. Dans une première étape, des règles de traduction supportant un fragment de LTL assez expressif pour décrire des propriétés de sécurité usuelles reformulent le problème de vérification en une expression de logique propositionnelle dont les atomes sont des comparaisons de langages obtenus par réécriture; nous appelons une telle formule une proposition de réécriture. La deuxième étape consiste à donner à cette proposition de réécriture une représentation concrète en termes d'automates d'arbres avec et sans contraintes. Étant donné que le problème général est indécidable, ces représentations sont occasionnellement approximées, ce pour quoi nous utilisons des constructions étudiées pour l'analyse d'accessibilité; le résultat final est un ensemble de procédures de semi-décision pour le problème général. **La seconde partie** se penche sur un aspect important des automates que nous utilisons: leurs contraintes. Nous étudions leur contribution à la complexité de plusieurs problèmes de décision, en particulier lorsque le nombre de contraintes est borné. **Finalement**, nous étudions également les automates d'arbres cheminants, une variété très différente, dont les connexions aux langages de navigation sur les documents semi-structurés sont fortes. Nous améliorons leur conversion en automates parallèles, et nous développons une procédure de semi-décision de leur vacuité, à la fois efficace et précise.

Mots-clés : Automates d'arbres, contraintes, approximations, semi-décision, vérification de modèles à arbres

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ