Decidability and complexity of simulation preorder for data-centric Web services

Lakhdar Akroun

To cite this version:

Lakhdar Akroun. Decidability and complexity of simulation preorder for data-centric Web services. Other [cs.OH]. Université Blaise Pascal - Clermont-Ferrand II, 2014. English. NNT : 2014CLF22523 . tel-01155513

N° d'ordre : D. U : 2523
E D S P I C : 676

UNIVERSITÉ Blaise Pascal Clermont-Ferrand II

U.F.R. Sciences

ÉCOLE DOCTORALE
SCIENENCES POUR L'INGENIEUR DE CLERMONT FERRAND

T H È S E

Présentée par
M. Lakhdar AKROUN
en vue de l'obtention du grade de

DOCTEUR de l'UNIVERSITÉ BLAISE PASCAL

Spécialité : Informatique

Titre :

Decidability and Complexity of Simulation Preorder for Data-Centric Web Services.

MEMBRES du JURY :

M. Gwen Salaün	HDR	Grenoble INP	(***Rapporteur***)
M. Kais Klai	HDR	Université Paris Nord	(***Rapporteur***)
M. Michael Rusinowitch	PR	Université de Lorraine	(***Examinateur***)
Mme. Véronique Benzaken	PR	Université Paris-Sud 11	(***Examinateur***)
M. Farouk Toumani	PR	Université Blaise Pascal	(***Directeur de thèse***)
M. Lhouari Nourine	PR	Université Blaise Pascal	(***Encadrant***)

$N°$ d'ordre : D. U : 2523
E D S P I C : 676

UNIVERSITÉ Blaise Pascal Clermont-Ferrand II

U.F.R. Sciences

ÉCOLE DOCTORALE
SCIENENCES POUR L'INGENIEUR DE CLERMONT FERRAND

T H È S E

Présentée par

M. Lakhdar AKROUN

en vue de l'obtention du grade de

DOCTEUR de l'UNIVERSITÉ BLAISE PASCAL

Spécialité : Informatique

Titre :

Decidability and Complexity of Simulation Preorder for Data-Centric Web Services.

MEMBRES du JURY :

Acknowledgments

Special thanks are directed to Gwen SALAÜN and Kais KLAI for having accepted to review my thesis manuscript. My thanks go as well to Michael RUSINOWITCH for having accepted to act as examiner of my thesis.

This thesis would not have been possible without my advisors, Farouk TOUMANI, Lhouari NOURINE and Boualem BENATALLAH.

Thanks to Mr. Alain Quillot for having hosted me at the LIMOS.

Abstract

In this thesis we address the problem of analyzing specifications of data-centric Web service interaction protocols (also called data-centric business protocols). Specifications of such protocols include data in addition to operation signatures and messages ordering constraints. Analysis of data-centric services is a complex task because of the inherently infinite states of the underlying service execution instances. Our work focuses on characterizing the problem of checking a refinement relation between service interaction protocol specifications. More specifically, we consider the problem of checking the simulation preorder when service business protocols are represented using data-centric state machines. First we study the Colombo model [BCG+05]. In this framework, a service (i) exchanges messages using variables; (ii) acts on a shared database; (iii) has a transition based behavior. We show that the simulation test for unbounded Colombo is undecidable. Then, we consider the case of bounded Colombo where we show that simulation is (i) EXPTIME-complete for Colombo services without any access to the database (noted $Colombo^{DB=\emptyset}$), and (ii) 2EXPTIME-complete when only bounded databases are considered (the obtained model is noted $Colombo^{bound}$). In the second part of this thesis, we define a *generic model* to study the impact of various parameters on the simulation test in the context of data-centric services. The generic model is a guarded transition system acting (i.e., read and write) on databases (i.e., local and shared) and exchanging messages with its environment (i.e., other services or users). The model was designed with a database theory perspective, where all actions are viewed as queries (i.e modification of databases, messages exchanges and guards). In this context, we obtain the following results (i) for *update free guarded* services (i.e., generic services with guards and only able to send empty messages) the decidability of simulation is fully characterized w.r.t decidability of satisfiability of the query language used to express the guards augmented with a restrictive form of negation, (ii) for *update free send* services (i.e., generic services without guards and able to send as messages the result of queries over local and shared database), we exhibit sufficient conditions for both decidability and undecidability of simulation test w.r.t the language used to compute messages payloads, and (iii) we study the case of *insert* services (i.e., generic services without guards and with the ability of insert the result of queries into the local and the shared database). In this case, we study the simulation as well as the weak simulation relations where we show that: (i) the weak simulation is undecidable when the insertions are expressed as conjunctive queries, (ii) the simulation is undecidable if satisfiability of the query language used to express the insertion augmented with a restricted form of negation is undecidable. Finally, we study the interaction between the queries used as guards and the ones used as insert where we exhibit a class of services where satisfiability of both languages is decidable while simulation is undecidable.

Keywords

Formal verification, data-centric Web services, database theory

Resumée

Dans cette thèse nous nous intéressons au problème d'analyse des spécifications des protocoles d'interactions des services Web orientées données. La spécification de ce type de protocoles inclue les données en plus de la signature des opérations et des contraintes d'ordonnancement des messages. L'analyse des services orientés données est complexe car l'exécution d'un service engendre une infinité d'états.

Notre travail se concentre autour du problème d'existence d'une relation de simulation quand les spécifications des protocoles des services Web sont représentés en utilisant un system a transition orienté données. D'abords nous avons étudié le model Colombo [BCG$^+$05]. Dans ce modèle, un service (i) échange des messages en utilisant des variables ; (ii) modifie une base de donnée partagée ; (iii) son comportement est modélisé avec un système a transition. Nous montrons que tester l'existence de la relation de simulation entre deux services Colombo non bornée est indécidable. Puis, nous considérons le cas où les services sont bornés. Nous montrons pour ce cas que le test de simulation est (i) EXPTIME-complet pour les services Colombo qui n'accèdent pas a la base de donnée (noté $Colombo^{DB=\emptyset}$), et (ii) 2EXPTIME-complet quand le service peut accéder a une base de donnée bornée ($Colombo^{bound}$). Dans la seconde partie de cette thèse, nous avons définie un modèle générique pour étudier l'impacte de différents paramètres sur le test de simulation dans le contexte des services Web orientés données. Le modèle générique est un systeme a transition gardé qui peut lire et écrire a partir d'une base de donnée et échanger des messages avec son environnement (d'autres services ou un client). Dans le modèle générique toutes les actions sont des requêtes sur des bases de données (modification de la base de données, messages échangés et aussi les gardes). Dans ce contexte, nous avons obtenue les résultats suivant : (i) pour les services gardés sans mise a jour, le test de simulation est caractérisé par rapport à la décidabilité du test de satisfiabilité du langage utilise pour exprimer les gardes augmenté avec une forme restrictive de négation, (ii) pour les services sans mise a jour mais qui peuvent envoyer comme message le résultat d'une requête, nous avons trouvé des conditions suffisantes d'indécidabilité et de décidabilité par rapport au langage utilise pour exprimer l'échange de messages, et (iii) nous avons étudié le cas des services qui ne peuvent que insérer des tuples dans la base de donnée. Pour ce cas, nous avons étudié la simulation ainsi que la weak simulation et nous avons montré que : (a) la weak simulation est indécidable quand les requêtes d'insertion sont des requêtes conjonctives, (b) le test de simulation est indécidable si la satisfiabilité du langage de requête utilisé pour exprimer les insertions augmenté avec une certaine forme de négation est indécidable. Enfin, nous avons étudié l'interaction entre le langage utilisé pour exprimer les gardes et celui utilisé pour les insertions, nous exhibons une classe de service où la satisfiabilité des deux langages est décidable alors que le test de simulation entre les services qui leurs sont associé ne l'est pas.

Mots clés

Vérification Formel, service Web orienté données, base de donnée

Table des matières

Chapitre 1

Introduction

Nowadays companies and organizations build (intra and inter) distributed information systems by integrating existing independent applications, also called legacy systems [Kra07]. These applications use proprietary tools and the cost of rewriting them from scratch would be unreasonable. The use of classical integration solutions (e.g., Enterprise Application Integration and Middlewares) requires a huge amount of resources and time to integrate the legacy systems [ACKM04]. *Web services* [W3C02] are gaining acceptance as a promising technology to deal with integration challenges. Web services are programs, that export their descriptions and make the functionality of an application available through standard web technologies. The use of such standards enable rapid, low-cost inter-operation and permit the definition of architectures and techniques to build new functionalities while integrating existing applications. Roughly speaking, a web service is a program that exports its behavior and can be invoked and executed by other programs via the web [ACKM04].

Different kinds of standards and models have been proposed to describe and reason on Web services [ACKM04]. Those standards focus on different levels of abstraction and target different aspects of Web services. In one extreme, a Web service is viewed as a black box, with its specification limited to the signatures of services operations. At the other extreme, the internal logic of the Web service is specified using workflow formalism and is made publicly available. Main stream service description languages such as WSDL[1] allow descriptions of low-level service operations. Semantic Web-based representation languages (e.g., OWL-S) investigated rich and machine-understandable descriptions of service properties and capabilities. Business protocol representation models and languages (e.g., state machines [BFHS03, BCT04b, BCT06], Petri-nets [NM02, HB03a, Loh08]) are description models which are used for specifying external behavior of services. Business protocols play an important role, since they provide to developers information on how to write a client (a service) to correctly interact with a given service. Business protocols record the intended behavior of the service [HLL$^+$12] and open the possibility to formally analyze and synthesis services. Recent approaches address the problem of checking similarity and compatibility of business protocols (e.g., [BCT04a, BSBM04, XDMNZ04, PF04, WMFN04]) as well as problems related to verification and synthesis of business protocols [NM02, HB03a, BFHS03, BCG$^+$03, GHIS04, DS05, BCG$^+$05, PTB05, FGG$^+$08, AP07, BCP08, FFM$^+$10, ARN12, MW07].
Business process specifications have recently evolved from process centric approaches to data-centric approaches. Process-centric models (i.e., state machines, Petri-nets) concen-

1. Web Services Description Language.

trate only on the flow of actions while the data and its modifications by the process operations are completely hidden. On the other hand, data-centric models for business process specification describe data and processes at the same level [AN03, Hul08, HSV13, CDM13]. The importance of explicitly representing data in business process specification comes from the fact that many decisions during the execution of a task depend on data values [MSW11, CDM13], and there is a strong interaction between the data of an information system and its operations. In fact, the operations modify the data, and data act as guards that control the execution of operations.

Incorporation of data into business process specification challenges formal system verification [CDM13]. The presence of data makes the formal models infinite. As a consequence, the direct use of classical algorithms to analysis them is often not possible. But the infiniteness of the models does not lead necessarily to undecidablity. For example, [AVFY98, AVFY00, DSV04, ABGM09, ASV08, ASV09a, ASV09b] consider the problem of verification of data-centric business processes using semantic or syntactic restrictions on the model (i.e., bounding the number of values in the database, limiting the access to the database) in order to obtain decidable fragments.

In this thesis, we consider the problem of analyzing specifications of data-centric business protocols using the simulation preorder [Mil71]. Simulation preorder is a relation between state transition systems which ensures that the behavior of a given system can be faithfully reproduced by a second one (in this case the first system is said to be *simulated* by the second one). Simulation stands out as the most well understood notion to compare behavior of programs [HHK95]. Checking if a transition system T_1 is simulated by a transition system T_2, can be viewed as a game between two players : T_1 is called *the spoiler* and T_2 *the duplicator*. The spoiler wins the simulation game if it can execute a move that the duplicator cannot reproduce. In this case, T_1 is not simulated by T_2. The duplicator wins the simulation game if it can reproduce each move of the spoiler. In this case, T_1 is simulated by T_2.

The relation of simulation has been used to study business protocol compatibility and substitution problems [BCT06] as well as business protocol synthesis [MW07, BCGP08]. Those problems are reducible to simulation between finite state machines when the business protocol is modeled with a finite transition system. The test of simulation between two finite state machines can be achieved in a polynomial time [HHK95]. The relation of simulation was also studied in the context of infinite transition systems. For example in [HNT08] it is shown that the unbounded variant of the protocol synthesis problem is decidable, i.e., when the number of instances of an available service that can be involved in a composition is not bounded a priori. This problem has been recasted in [HNT08] as a problem of deciding simulation between a finite state machine and an infinite state machine representing a shuffle closure of existing services. When the service is represented using Petri net [Pet73], the simulation test is ranged from EXPTIME-complete [KM02b] to undecidable [KM02a]. The problem of simulation is known to be decidable for one-counter nets [AC98].

The study of simulation when services incorporate a database in their specifications was addressed only in few works [PG09, BCG$^+$05]. For example, [PG09] studies the simulation between data-centric services in a restricted framework, where : (i) the language used to updates the databases is very restrictive, (ii) the size of the allowed database instances is bounded. The authors prove that the simulation is decidable in this setting. In [BCG$^+$05], the authors study the composition problem where the decidability is obtained by bounding the number of new values introduced during an execution of a service as well as the number of accesses to the database. In this context, service composition has been shown to be in 2-EXPTIME.

In this thesis, we study the relation of simulation between data-centric business protocols. We focus our attention first on the decidability and the complexity issues for an existing model, namely the Colombo model [BCG$^+$05]. Then, we define a *generic model* to study the impact of various parameters on the simulation test in the context of data-centric services. The generic model is a guarded transition system acting (i.e., read and write) on databases (i.e., local and shared) and exchanging messages with its environment (i.e., other services or users). The generic model is inspired from data-centric models proposed in the literature [ABGM09, BLP11, AD07, AVFY98]. The model was designed with a database theory perspective, where all actions are viewed as queries (i.e modification of databases, messages exchanges and guards). With this optic, existing results and tools developed in the area of database theory provide a great help to understand the impact of including data into specifications of web services and its effects on decidability of simulation.

Main contributions

We summarize below, the main contributions of this thesis.

Colombo model. Colombo is a pioneer data-centric service model that has been used to investigate the service composition problem. A Colombo service is specified as a guarded transition system, augmented with a shared (with other services of the system) database as well as a set of variables that are used to send and receive messages. The modification of the database and the variables is achieved through *atomic processes*. An atomic process describes actions in terms of its inputs, outputs, preconditions and postconditions. Two sources of infiniteness make the simulation test difficult in this context :

- the variables take their values from an infinite domain and hence the number of potential messages (and hence values) that can be received by a service in a given state may be infinite. As a consequence, the number of successor of a state may be infinite as well as the number of configurations of a service (this is because a service execution may visit an infinite number of databases simply by inserting the received values in the shared database).
- the number of possible initial instances of the shared database is infinite which makes the number of initial configurations of a service infinite.

At first glance, the Colombo model appears to have a limited expressivity since :

- it restricts accesses to the database only through atomic processes, and
- it supports a very limited database *'query'* language which consists in simple key-based access functions.

The table 1.1 summarizes the results of decidability and complexity obtained for different classes of the Colombo model. More precisely, we show that, checking simulation in a Co-

TABLE 1.1 – Results of simulation for the Colombo model.

Class of services	Simulation
$Colombo^{unb}$	Undecidable
$Colombo^{DB=\emptyset}$	EXPTIME-complete
GVA	EXPTIME-complete
$Colombo^{bound}$	2-EXPTIME-complete

lombo model with unbounded accesses to the database, called $Colombo^{unb}$, is undecidable. The proof is based on a reduction from the halting problem of a two counter machine (a

Minsky machine) [Min67] to the state reachability problem in *Colombo*^{unb}. Even worse, the way the proof is constructed enables to derive that the reachability and the simulation problems remain undecidable even in the case of non-communicating *Colombo*^{unb} services with read-only accesses to the database (i.e., services that cannot send or receive messages nor update the shared database). Then, we study the simulation problem in the case of Colombo services with a bounded database (i.e. the class of Colombo services having shared database with a number of tuples that cannot exceed a given constant k). Such a class is called *Colombo*^{bound}. We show that the simulation is 2-EXPTIME-complete for *Colombo*^{bound}. The proof is achieved in two steps :

- First we show that checking simulation is EXPTIME-complete for Colombo services without any access to the database (namely DB-less services *Colombo*$^{DB=\emptyset}$). *Colombo*$^{DB=\emptyset}$ services are also infinite-state systems, because they manipulate variables which take their values from an infinite domain. A finite symbolic representation of such services can be obtained by partitioning the original infinite state space (here a state is characterized by the control state of the transition system and a valuation of the variables) into a finite number of equivalence classes. Then, a simulation algorithm can be designed using a symbolic procedure that manipulates finite sets of states (i.e., the equivalence classes) instead of infinite individual states. The complexity is obtained by a reduction from the existence of infinite execution of an *alternating Turing machine* working on a space polynomially bounded by the size of its input.
- As a side effect of this work, we establish a correspondence between *Colombo*$^{DB=\emptyset}$, restricted to equality, and Guarded Variable Automata (GVA) [BCR14]. As a consequence, we derive EXPTIME-completeness of simulation for GVA. Note that, an EXPTIME upper bound of simulation in GVA is provided in [BCR14].
- Then we show that checking the simulation for *Colombo*^{bound} services can be rewritten into equivalent *Colombo*$^{DB=\emptyset}$ while preserving the simulation preorder. The 2-EXPTIME-hardness of checking the simulation for *Colombo*^{bound} services is obtained by a reduction from the existence of infinite execution of an *alternating Turing machine* working on a space exponentially bounded by the size of its input.

Generic model. We define a generic data-centric service as :

- a guarded transition system augmented with the ability of updating (i.e., read and write) databases (i.e., local and shared). The notion of local database is introduced to express the fact that some parts of the information are private to a service and hence are not visible to other services. Operations over a local database are defined as *silent* transitions [HB03b, vdADO+08]. For example, modifying the local database is a silent transition (i.e., not observable from an external point of view). In the context of the simulation preorder, this notion of observable transitions and non-observable transitions is captured with *weak simulation*,
- the service modifies the databases through update queries expressed in the language $\mathcal{L}_U$,
- the guards are boolean queries over the databases, expressed in a language $\mathcal{L}_T$,
- communication between web services is captured with incoming and outgoing messages. The incoming messages are databases, and the outgoing messages are expressed using queries over the *local* and the *shared* databases, in a language $\mathcal{L}_S$.

Example 1. Figure 1.1 depicts an example of services specified using our generic model. Each service (A and B) has its own local database as well as a shared database. Services communicate through messages. Note that an outgoing message is the result of a query

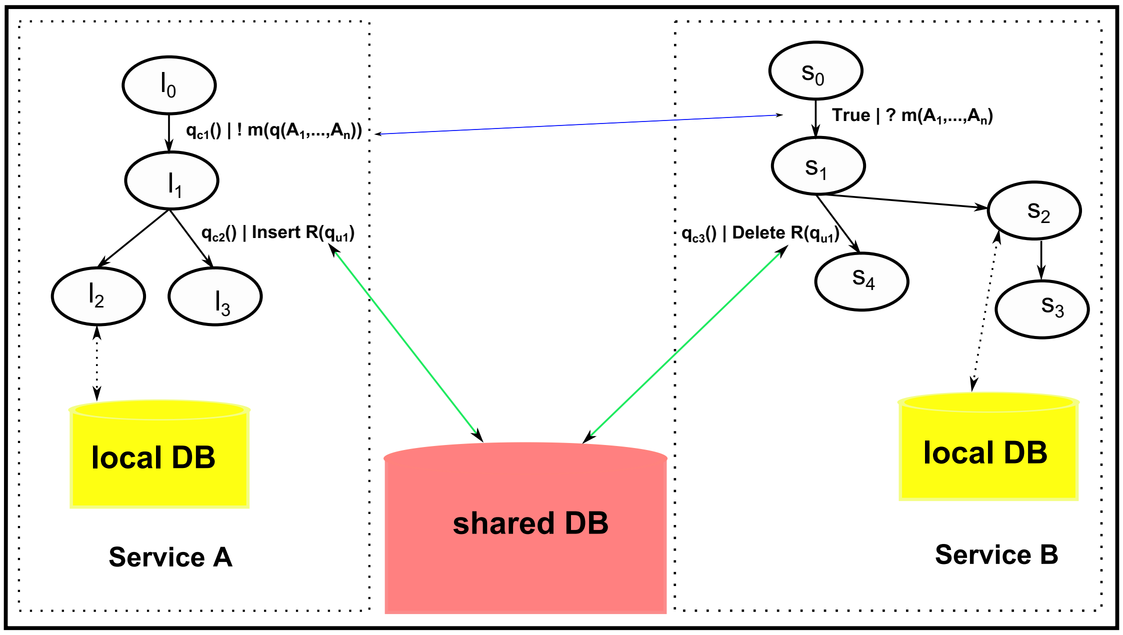

l_0
$q_{c1}()$ | ! $m(q(A_1,...,A_n))$
l_1
$q_{c2}()$ | Insert $R(q_{u1})$
l_2
l_3
local DB
Service A
shared DB
s_0
True | ? $m(A_1,...,A_n)$
s_1
$q_{c3}()$ | Delete $R(q_{u1})$
s_2
s_4
s_3
local DB
Service B

FIGURE 1.1 – Generic web service framework.

(e.g., the service A sends a message m, which contains a result of the query q). A query q can be defined over the local as well as the shared database. The transitions are guarded by boolean queries (q_{ci}). Finally, services can modify the databases using *update queries* (e.g., the service A inserts the result of the query q_{u1} into the relation R). In this thesis, we focus our attention on insert queries and we do not consider delete and modify queries.

In order to isolate and study the impact of the different parameters of the generic model on the simulation preorder, we investigate the decidability and complexity issues of the simulation for various classes of our generic model. Each class is characterized by :

- the type of actions supported by the model, e.g., the service can only send messages, or only insert in the database, ... etc,
- the languages used to instantiate respectively $\mathcal{L}_T$, $\mathcal{L}_U$ and $\mathcal{L}_S$,
- the presence or not of the local database (i.e., in the presence of local database, we study weak simulation).

Table 1.2 summarizes the considered sub-classes of the generic model as well as the obtained results. We consider more precisely the following classes :

- Update-free services. This class represents services which are not able to make modifications over the databases. The class of update-free service is decomposed into two sub-classes :
  - Guarded services, this class enables to focus on the role played by the language of guards ($\mathcal{L}_T$) on the decidability of the simulation relation. Our main result regarding this class lies in a full characterization of the decidability of simulation in terms of the decidability of checking satisfiability of formulas expressed in the language $\mathcal{L}_T$ augmented with a restricted form of negation. We denote this language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ (i.e., the conjunction and negation is applied on boolean $\mathcal{L}_T$ formulas). As for the case of $Colombo^{DB=\emptyset}$, we use a finite symbolic representation of update-free services by partitioning the original infinite state space into a finite number of equivalence classes.
  - Send services. This class represents update-free services which send the results of

TABLE 1.2 – Summarization of results.

Class of services	Restrictions	Simulation
Guarded services		decidable iff satisfiablity of $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is decidable
Send services		undecidable if satisfiability of $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable
Send services		decidable if satisfiability of a partition is decidable
Insert services	-insertion	Undecidable if satisfiability of the language of insertion $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ is undecidable
Insert services	-insertion -$\mathcal{L}_U = GNCQ$	Undecidable
Class of services	**Restrictions**	**Weak simulation**
Insert services	-insertion -$\mathcal{L}_U = CQ$ -local database	Undecidable

queries expressed in the language $\mathcal{L}_S$ as messages. We focus on the role played by the language $\mathcal{L}_S$. As a result, we show that the test of simulation for send services is undecidable if satisfiability of formulas in the language $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable (i.e., the conjunction and negation is applied on boolean $\mathcal{L}_S$ formulas). We extend the symbolization framework used in the case of guarded services to obtain decidability of simulation between send services. In this case, the simulation is decidable if testing the satisfiability of a partition is decidable. Note that, in current state of affairs we are not able to provide a full characterization of simulation in this class since we are only able to provide sufficient conditions for both decidability and undecidability of simulation in this context.

– Insert services. This class describes services without guards. The considered services are able to insert data in the shared database. In this context, we study the simulation as well as the weak simulation relations. This later one is considered when the insertion in the local database is allowed. As a result, the test of simulation for insert services is undecidable if checking the satisfiability of formulas in the language $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ is undecidable (i.e., the conjunction and negation is applied on boolean $\mathcal{L}_I$ formulas) . We are not able to provide a full characterization of the decidability of simulation for insert services. We exhibit a language $GNCQ$ (*Guarded Negation Conjunctive Query*[2]) where testing the satisfiability for boolean $GNCQ$ queries augmented with a restricted form of negation is decidable but the simulation for insert services using the language $GNCQ$ as insertion query language is undecidable. The problem remains open when $\mathcal{L}_I$=CQ. Finally, we prove that the weak simulation is undecidable when the language of insertion $\mathcal{L}_I = CQ$.
– We also study the interaction of the languages used to express the guards with the updates. More precisely we show that, testing the simulation relation is undecidable when generic services use $GNCQ$ as guards and CQ as insertion query language.

2. The *Guarded Negation Conjunctive Query* ($GNCQ$) language is included in *Guarded Negation First Order* language $GNFO$ [BtCS11]. $GNCQ$ queries are conjunctive queries with guarded negations (i.e., all variables appearing in negative atoms must appears in a positive atom)

Structure of the thesis

This thesis is structured as follows. In Chapter 2 we give an overview of the Colombo model and present our result regarding the undecidability of simulation for $Colombo^{unb}$. This chapter addresses then the complexity issue for $Colombo^{DB=\emptyset}$ and $Colombo^{bound}$. In Chapter 3, we define our generic model and the relation of (weak)simulation. Then, we study decidability and complexity issues for *Update free* services. After that, we focus on *Insert* services. Finally, we review related works and conclude in Chapter 4. Additional proofs and technical details are given in appendix A.

Chapitre 2

Checking Simulation Preorder in the Colombo Model

This chapter is organized as follows : we start by some preliminaries in section 2.1. In section 2.2 we overview the *Colombo* model and defines the associated simulation problem. Section 2.3 describes our results regarding undecidability of unbounded Colombo. Section 2.4 considers the case of DB-less $Colombo^{DB=\emptyset}$ services (i.e., Colombo services which are not able to access to databases) and show decidability and complexity results of simulation in this context. Section 2.5 is devoted to $Colombo^{bound}$ case (i.e., Colombo services with bounded database). Section 2.6 concludes this chapter.

2.1 Preliminaries

Relational database We assume some familiarity with relational database concepts (e.g., see [AHV95]). Let $\mathcal{U}$ be an infinite set of attributes, $\mathcal{V}$ a possibly infinite set of variables and $\mathcal{D}$ an infinite set of constants (values). The sets $\mathcal{U}$, $\mathcal{V}$ and $\mathcal{D}$ are pairwise disjoint. Associated with every attribute $A \in \mathcal{U}$ an attribute domain $Dom(A) \subseteq \mathcal{D}$. A relational schema $\mathcal{R}$ is a set $\{R_1, \ldots, R_n\}$ of relation schemas, where each R_i is defined over a finite set $X_i \subset \mathcal{U}$ of attributes, X_i=$\{A_1, \ldots, A_n\}$ and arity(R_i)=n. We write $schema(R_i) = X_i$. An instance r of R_i over the set of attributes $\{A_1, \ldots, A_n\}$ is a finite subset of the Cartesian product $Dom(A_1) \times \ldots \times Dom(A_n)$. We denote by $|r|$ the cardinality of the relation r (i.e., the total number of tuples in the instance r). An instance I (database) of the relational schema $\mathcal{R}$ is the set $\{r_1, \ldots, r_n\}$ where each r_i is an instance of $R_i \subset \mathcal{R}$. The set of all possible database of $\mathcal{R}$ is denoted $\mathcal{I}_{\mathcal{R}}$.

Finite state machine A finite state machine M is a tuple $\langle \Sigma_M, Q_M, F_M, q_M^0, \delta_M \rangle$ where :

- Σ_M is a finite alphabet,
- Q_M is a set of states,
- $F_M \subseteq Q_M$ is the set of final states,
- q_M^0 is the initial state,
- $\delta_M \subseteq Q_M \times \Sigma_M \times Q_M$ is the set of transitions.

Simulation preorder for finite state machines Let $M = \langle \Sigma_M, Q_M, F_M, q_M^0, \delta_M \rangle$ and $M^{'} = \langle \Sigma_{M'}, Q_{M'}, F_{M'}, q_{M'}^0, \delta_{M'} \rangle$ be two finite state machines. A state $q_1 \in Q_M$ is simulated by a state $q_1^{'} \in Q_{M'}$ noted $q_1 \preceq q_1^{'}$ iff the following conditions hold :

- $\forall a \in \Sigma_M$ and $\forall q_2 \in Q_M$ such that $(q_1, a, q_2) \in \delta_M$ there exists a transition $(q_1^{'}, a, q_2^{'}) \in \delta_{M^{'}}$ such that $q_2 \preceq q_2^{'}$, and
- if $q_1 \in F_M$ then $q_2^{'} \in F_{M^{'}}$.

$M \preceq M^{'}$ iff $q_M^0 \preceq q_{M^{'}}^0$.
M and $M^{'}$ are simulation equivalent, noted $M \cong M^{'}$ iff $M \preceq M^{'}$ and $M^{'} \preceq M$.

2.2 Overview on the Colombo model

We present below a simplified version of the Colombo model which is sufficient to present our results [1]. A detailed description of the Colombo model is given in [BCG+05].

A *world* database schema, denoted $\mathcal{W}$, is a finite set of relation schemas having the form $R_k(A_1, \ldots, A_k; B_1, \ldots, B_n)$, where A_is, B_js are attributes and the A_is form a key for R_k. A world database is an instance over the schema $\mathcal{W}$. Let $R(A_1, \ldots, A_k; B_1, \ldots, B_n)$ be a relation schema in $\mathcal{W}$, then $f_j^R(A_1, \ldots, A_k)$ is an access function that returns the $k+j$-*th* element of the tuple t in R identified by the key $(A_1, \ldots, A_k)$(i.e., the j-*th* element of the tuple t after the key). Given a set of constants C and variables V, the set of accessible terms over C and V is defined recursively to include all the terms constructed using C, V and the f_j^R functions.

Example 2. Figure 2.1(c) depicts an example of a world database schema while figure 2.2 shows an instance of such a schema. For example, access to the relation Inventory(code, available, warehouse, price) is only possible through the access function $f_j^{Inventory}(code)$ with $j \in [1, 3]$. For instance, using the world database depicted at figure 2.2, the function $f_2^{Inventory}(\text{“}HP15\text{”})$ returns the value “NGW”, corresponding to the value of the second attribute (i.e., the attribute warehouse) of the tuple identified by the code “$HP15$” in the relation Inventory.

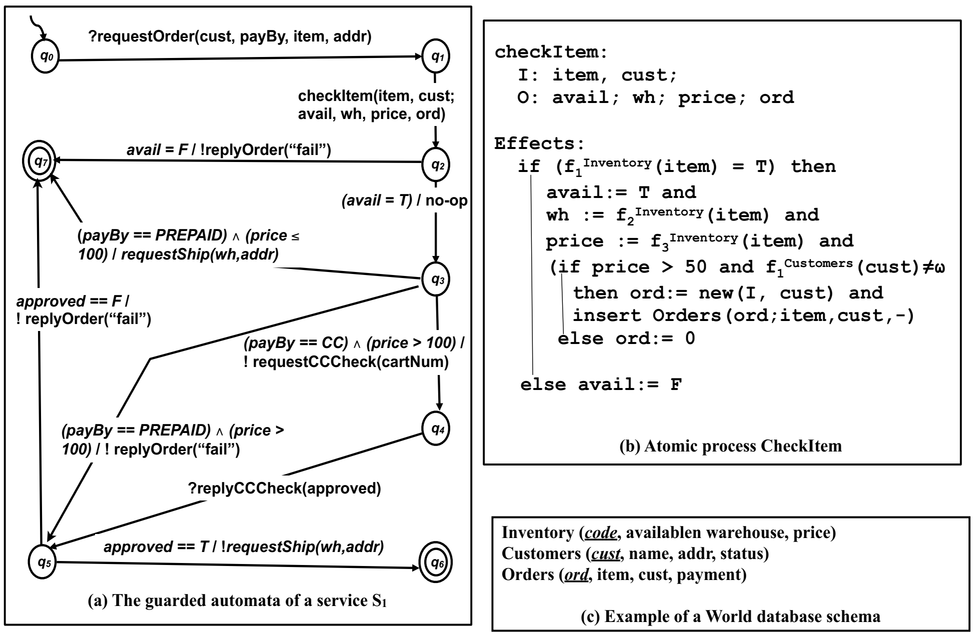
2.2.1 Atomic processes

In the Colombo model, services *actions* are achieved using the notion of atomic processes. An atomic process is a triplet $p = (I, O, CE)$ where : I and O are respectively input and output signatures (i.e., sets of typed variables) and $CE = \{(\theta, E)\}$, is a set of conditional effects, with :

- Condition θ is a boolean expression over atoms over accessible terms over some family of constants and the input variables $u_1, \ldots, u_n$ in I,
- A set of effects E where each effect $e \in E$ is a pair (es, ev) with :
  - es, effect on world state, is a set of modifications on the global database, i.e., expressions of the form
    - $insert\ R(t_1, \ldots, t_k, s_1, \ldots, s_l)$,
    - $delete\ R(t_1, \ldots, t_k)$,
    - $modify\ R(t_1, \ldots, t_k, r_1, \ldots, r_l)$,

    where each t_i, with $i \in [1, k]$, (respectively, s_j with with $j \in [1, l]$) is an accessible term over some set of constants and input variables $u_1, \ldots, u_n$ in I, and where each r_j, with $j \in [1, l]$, is either an accessible term or the special symbol “_” which indicates a position of the identified tuple in R which should remain unchanged.
  - ev, effects on output variables, is a set of expressions of the forms : $v_j := t, \forall v_j \in O$ such that either $t = \omega$ or t is an accessible term over some set of constants and

1. In particular, we omit notions like *QStore*, linkage, ..., which are not relevant for our purposes.


?requestOrder(cust, payBy, item, addr)
checkItem(item, cust; avail, wh, price, ord)
avail = F / !replyOrder("fail")
(avail = T) / no-op
(payBy == PREPAID) ∧ (price ≤ 100) / requestShip(wh,addr)
approved == F / ! replyOrder("fail")
(payBy == CC) ∧ (price > 100) / ! requestCCCheck(cartNum)
(payBy == PREPAID) ∧ (price > 100) / ! replyOrder("fail")
?replyCCCheck(approved)
approved == T / !requestShip(wh,addr)
q0
q1
q2
q3
q4
q5
q6
q7
(a) The guarded automata of a service S1

```
checkItem:
  I: item, cust;
  O: avail; wh; price; ord

Effects:
  if (f1^Inventory(item) = T) then
     avail:= T and
     wh := f2^Inventory(item) and
     price := f3^Inventory(item) and
     (if price > 50 and f1^Customers(cust)≠ω
       then ord:= new(I, cust) and
       insert Orders(ord;item,cust,-)
      else ord:= 0

  else avail:= F
```

(b) Atomic process CheckItem

Inventory (<u>*code*</u>, availablen warehouse, price)
Customers (<u>*cust*</u>, name, addr, status)
Orders (<u>*ord*</u>, item, cust, payment)

(c) Example of a World database schema

FIGURE 2.1 – Example of Colombo service (inspired from [BCG⁺05]).

over the input variables $u_1, \ldots, u_n$. The symbol ω is used to denote an *undefined* (or null) value.

Example 3. Figure 2.1(b) shows an example of a specification of an atomic process (the atomic process CheckItem). This process takes as input an item code (item) and a customer number (cust) and checks first if the requested item is available in the relation inventory (condition if $\mathsf{f}_1^{\mathsf{inventory}}(\mathsf{item}) = \mathsf{T}$). If the requested item is not available, the process CheckItem simply returns the output parameter *avail* $= F$. Otherwise, if the requested item is available, the process returns the warehouse where the item is stocked and the price. Moreover, if the price of the requested item is greater than 50 and the status of the current customer is defined (condition if price > 50 and $\mathsf{f}^1_{\mathsf{Customers}}(\mathsf{cust}) \neq \omega$), then a new order id is created and inserted in the relation Orders. Otherwise, the process returns the output parameter ord $= 0$. Note that, the new order id is created with the function new, this is just for simplifying the example. In fact, the new value of ord is obtained through a message, then the service verify if this value is not the value of an existing order id.

2.2.2 Guarded automata

The behavior of a Colombo service is given by the notion of *guarded automata* as defined below.

Definition 1. ***guarded automaton (GA)***
A guarded automaton of a service S is a tuple $GA(S) = \langle Q, \delta, q_0, F, LStore(S)\rangle$, where :
- *Q is a finite set of control states with $q_0 \in Q$ the initial state,*
- *$F \subseteq Q$ is a set of final states,*
- *$LStore(S)$ is a finite set of typed variables,*
- *the transition relation δ contains tuples (q, θ, μ, q') where $q, q' \in Q$, θ is a condition over LStore (no access to world instance), and μ has one of the following forms :*

- *(incoming message) $\mu = ?m(v_1, \ldots, v_n)$ where m is a message having as signature $m(p_1, \ldots, p_n)$, and $v_i \in LStore(S), \forall i \in [1, n]$, or*
- *(send message) $\mu = !m(b_1, \ldots, b_n)$ where m is a message having as signature $m(p_1, \ldots, p_n)$, and $\forall i \in [1, n]$, each b_i is either a variable of $LStore(S)$ or a constant, or*
- *(atomic process invocation) $\mu = p(u_1, \ldots, u_n; v_1, \ldots, v_m, CE)$ with p an atomic process having n inputs, m outputs and CE as conditional effects, and $\forall i \in [1, n]$, each u_i (respectively, v_i) is either a variable of $LStore(S)$ or a constant.*

A message type has the form $m(p_1, \ldots, p_n)$ where m is the message name and $p_1, \ldots, p_n$ are message parameters. Each parameter p_i is defined over a domain $\mathcal{D}$.
$LStore(S)$ can be viewed as a working area of a service. The variables of $LStore(S)$ are used to (i) capture the values of incoming messages, (ii) capture the output values of atomic processes, (iii) populate the parameters of outgoing messages, and (iv) populate the input parameters of atomic processes.

Example 4. Figure 2.1(a), inspired from [BCG+05], shows the guarded automata of a Warehouse service. The states of the automata represent the different phases that the service may go through during its execution. Transitions are associated with a send or a receive message or with an atomic process. The Warehouse service is initially at its initial state (i.e., the state indicated in the figure by an unlabeled entering arrow). The service starts its execution upon receiving a requestOrder message. Then, depending on the requested payment mode and the price, respectively given by the values of the received message parameters payBy and price, the service can make two possible moves : (i) if the payment mode is CC (credit card) or the price > 10, the service sends a requestCCCheck message, for example to a bank, in order the check whether the credit card can be used to make the payment, or (ii) if the payment mode is $PREPAID$ and the price ≤ 10, the service will execute the atomic process charge in order to achieve the payment. The service ends its execution at a final state, depicted in the figure by double-circled states.

If a given guarded automaton $GA(S)$ uses only transitions of the form (q, θ, μ, q') with μ is an atomic process, in this case the corresponding service S is called a *non-communicating* service (since S cannot exchange messages with its environment). Moreover, if all the atomic processes used in a guarded automaton $GA(S)$ have no effects on world states (i.e., the set es of each atomic process is empty), in this case the service S is called a *read-only* Colombo service.

2.2.3 Service runs

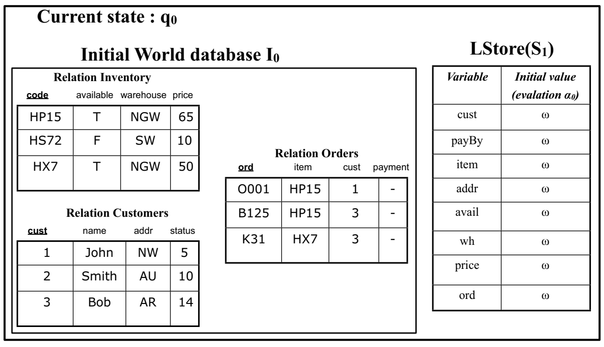
We use the notion of an extended automata to define the semantics of a Colombo service. At every point in time, the behavior of an instance of a Colombo service S is determined by its *instantaneous description (or simply, configuration).* A configuration of a service is given by a triplet $id = (l, \mathcal{I}, \alpha)$ where l is its current control state, $\mathcal{I}$ a world database instance and α is a valuation over the variables of $LStore$.

Definition 2. ***(service runs)***
Let $GA(S) = \langle Q, \delta, l_0, F, LStore(S) \rangle$ be a guarded automata of a service S. A run σ of S is a finite sequence $\sigma = id_0 \xrightarrow{\mu_0} id_1 \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_{n-1}} id_n$ wich satisfy the following conditions :

- *(Initiation) $id_0 = (l_0, \mathcal{I}_0, \alpha_0)$ is an initial configuration of the run with I_0 is an arbitrary database over $\mathcal{W}$ and $\alpha_0(x) = \omega$, $\forall x \in LStore(S)$.*

– *(Consecution)* $\forall i \in [1,n]$, $id_i = (l_i, \mathcal{I}_i, \alpha_i)$ *and there is a transition* $(l_i, \theta, \mu, l_{i+1}) \in \delta$ *such that* $\alpha_i(\theta) \equiv \mathsf{true}$ *and one of the following conditions holds :*
  – $\mu = ?m(v_1, \ldots, v_n)$ *and* $\mu_i = ?m(c_1, \ldots, c_n)$, *with* c_j *a constant* $\forall j \in [1,n]$, *then* $\mathcal{I}_{i+1} = \mathcal{I}_i$ *and* $\alpha_{i+1}(v_j) = c_j$ *and* $\forall v \in LStore(S) \setminus \{v_1, \ldots, v_n\}, \alpha_{i+1}(v) = \alpha_i(v)$,
  – $\mu = !m(b_1, \ldots, b_n)$ *and* $\mu_i = !m(\alpha_i(b_1), \ldots, (\alpha_i(b_n)))$ *then* $\mathcal{I}_{i+1} = \mathcal{I}_i$ *and* $\forall v \in LStore(S), \alpha_{i+1}(v) = \alpha_i(v)$, *and*
  – $\mu = p(u_1, \ldots, u_n; v_1, \ldots, v_m, CE)$ *and* $\mu_i = p(\alpha_i(u_1), \ldots, \alpha_i(u_n); \alpha_{i+1}(v_1), \ldots, \alpha_{i+1}(v_m), CE)$ *then*
    – *if there is no* $(c, E) \in CE$ *s.t.* $\alpha_i(c) \equiv \mathsf{true}$ *(or there is more than one such* (c, E)*) then* $\mathcal{I}_{i+1} = \mathcal{I}_i$ *and* $\forall v \in LStore(S), \alpha_{i+1}(v) = \alpha_i(v)$, *or*
    – *let* (c, E) *be the unique conditional effects in* CE *s.t.* $\alpha_i(c) \equiv \mathsf{true}$, *and let* (es, ev) *be a non-deterministically chosen element of* E, *then :*
      – *for each statement insert* $R(t_1, \ldots, t_k, s_1, \ldots, s_l)$, *delete* $R(t_1, \ldots, t_k)$, *or modify* $R(t_1, \ldots, t_k, s_1, \ldots, s_l)$ *in es, apply the corresponding modification obtained by replacing* t_i *(respectively,* s_i*) by* $\alpha_i(t_i)$ *(respectively,* $\alpha_i(s_i)$*) on the instance* $\mathcal{I}_i$*. The obtained instance is the database* $\mathcal{I}_{i+1}$.
      – $\forall v_j := t \in ev$, $\alpha_{i+1}(v_j) = \alpha_i(t)$ *and* $\alpha_{i+1}(v) = \alpha_i(v)$ *for all the other variables* v *of* $LStore(S)$.

An execution of a service S starts at an initial configuration $id_0 = (l_0, \mathcal{I}_0, \alpha_0)$, with l_0 the initial control state of $GA(S)$, $\mathcal{I}_0$ an arbitrary database over $\mathcal{W}$ and $\alpha_0(x) = \omega$, $\forall x \in LStore(S)$. Then, a service moves from an id_i to id_j according to the mechanics defined by the set of transitions of $GA(S)$. If $id_i \xrightarrow{\mu_i} id_j$ satisfies the consecution condition above, we say that μ_i is allowed from id_i. More specifically, we have the following cases :
– $\mu = ?m(v_1, \ldots, v_n)$ then only $(v_1, \ldots, v_n)$ receive new values. The other variables and the database do no change.
– $\mu = !m(b_1, \ldots, b_n)$ then there is no modification on the variables nor the database.
– $\mu = p(u_1, \ldots, u_n; v_1, \ldots, v_m, CE)$ then
  – if there is no $(c, E) \in CE$ where c is verified (or there is more than one) then there is no modification of the variables nor the database.
  – let (c, E) be the unique conditional effects in CE s.t c is verified, and let (es, ev) be a non-deterministically chosen element of E, then :
    – for each statement *insert* $R(t_1, \ldots, t_k, s_1, \ldots, s_l)$, *delete* $R(t_1, \ldots, t_k)$, or *modify* $R(t_1, \ldots, t_k, s_1, \ldots, s_l)$ in es, apply the corresponding modifications. The obtained instance is the database $\mathcal{I}_{i+1}$.
    – for all $v_j := t$ in ev, execute the assignment, all the other variables v of $LStore(S)$ do not change.


Current state : q_0

Initial World database I_0

Relation Inventory

code	available	warehouse	price
HP15	T	NGW	65
HS72	F	SW	10
HX7	T	NGW	50

Relation Customers

cust	name	addr	status
1	John	NW	5
2	Smith	AU	10
3	Bob	AR	14

Relation Orders

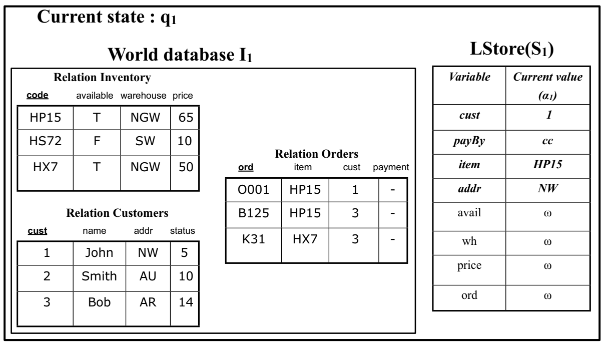
ord	item	cust	payment
O001	HP15	1	-
B125	HP15	3	-
K31	HX7	3	-

LStore(S_1)

Variable	*Initial value (evalation α_0)*
cust	ω
payBy	ω
item	ω
addr	ω
avail	ω
wh	ω
price	ω
ord	ω

Figure 2.2 – Example of an initial configuration $id_0 = (l_0, \mathcal{I}_0, \alpha_0)$.

Example 5. We illustrate in this example a run of our sample Warehouse service S_1 depicted at figure 2.1. Figure 2.2 shows a possible initial configuration of the Warehouse service S_1. This configuration is made of : (i) the initial state q_0 of the guarded automaton of S_1, (ii) an initial world database over the relation schemas Inventory, Customers and Orders, and (iii) the local store LStore(S_1) having all its variables set to ω (i.e., the variables are initially undefined). Upon the reception of the message requestOrder(cust := '1", payBy := "cc", item := "HP15", addr := "NW") the service S_1 moves from configuration $id_0 = (l_0, \mathcal{I}_0, \alpha_0)$ to the configuration $id_1 = (l_1, \mathcal{I}_1, \alpha_1)$ depicted at figure 2.3. Note that at configuration id_1, the world database is left unchanged while the values conveyed by the message requestOrder are stored in the corresponding variables in LStore(S_1). Then, upon the execution of the atomic process CheckItem, the service moves from configuration id_1 to the configuration $id_2 = (l_2, \mathcal{I}_2, \alpha_2)$ depicted at figure 2.4. As explained in the previous example, the atomic process CheckItem (c.f., figure 2.1(b)), takes as input parameter the variable item whose value at configuration id_1 is $\alpha_1(\text{item}) =$ "HP15". Hence, the condition (if $f_1^{\text{Inventory}}(\text{item}) = T$) in the specification of the *effects* of the CheckItem process is evaluated to true. Therefore, the output parameters avail, wh and price of the CheckItem process are updated as follows : avail := T, wh := $f_2^{\text{Inventory}}$("HP15") = "NGW" and price := $f_3^{\text{Inventory}}$("HP15") = "65". Moreover, the condition (if price > 50 and $f^1_{\text{Customers}}(\text{cust}) \neq \omega$) is also evaluated to true at configuration id_1. Hence, a new order id is generated (e.g., the order L021) and inserted in the relation Orders.


Current state : q₁

World database I₁

Relation Inventory

code	available	warehouse	price
HP15	T	NGW	65
HS72	F	SW	10
HX7	T	NGW	50

Relation Customers

cust	name	addr	status
1	John	NW	5
2	Smith	AU	10
3	Bob	AR	14

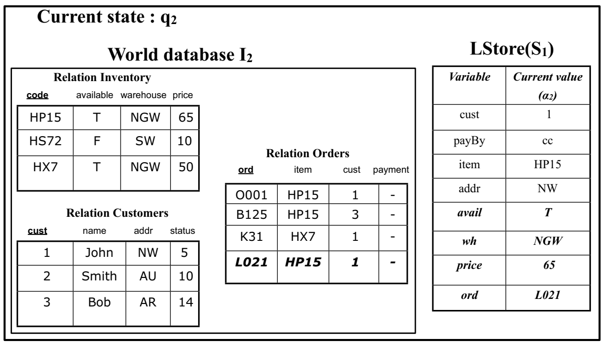
Relation Orders

ord	item	cust	payment
O001	HP15	1	-
B125	HP15	3	-
K31	HX7	3	-

LStore(S₁)

Variable	*Current value (α_1)*
cust	***1***
payBy	***cc***
item	***HP15***
addr	***NW***
avail	ω
wh	ω
price	ω
ord	ω

Figure 2.3 – The configuration $id_1 = (l_1, \mathcal{I}_1, \alpha_1)$ after reception of the message requestOrder.


Current state : q₂

World database I₂

Relation Inventory

code	available	warehouse	price
HP15	T	NGW	65
HS72	F	SW	10
HX7	T	NGW	50

Relation Customers

cust	name	addr	status
1	John	NW	5
2	Smith	AU	10
3	Bob	AR	14

Relation Orders

ord	item	cust	payment
O001	HP15	1	-
B125	HP15	3	-
K31	HX7	1	-
L021	***HP15***	***1***	**-**

LStore(S₁)

Variable	*Current value (α_2)*
cust	1
payBy	cc
item	HP15
addr	NW
avail	***T***
wh	***NGW***
price	***65***
ord	***L021***

Figure 2.4 – The configuration $id_2 = (l_2, \mathcal{I}_2, \alpha_2)$ after the execution of the checkItem process.

2.2.4 Extended state machine

The semantics of a Colombo service can be captured by the following notion of an extended infinite state machine.

Definition 3. *(extended state machine) Let $GA(S) = \langle Q, \delta, l_0, F, LStore(S)\rangle$ be a guarded automata of a service S. The associated infinite state machine, noted $E(S)$, is a tuple $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ where :*

- *$\mathbb{Q} = \{(l, \mathcal{I}, \alpha)\}$ with $l \in Q$, $\mathcal{I}$ a database over $\mathcal{W}$ and α a valuation over the variables of LStore. The set $\mathbb{Q}$ contains all the possible configurations of $E(S)$.*
- *$\mathbb{Q}_0 = \{(l_0, \mathcal{I}_0, \alpha_0)\}$, with $\mathcal{I}_0$ an arbitrary database over $\mathcal{W}$ and $\alpha_0(x) = \omega$, $\forall x \in LStore(S)$. $\mathbb{Q}_0$ is the infinite set of initial configurations of $E(S)$.*
- *$\mathbb{F} = \{(l_f, \mathcal{I}, \alpha) \mid l_f \in F\}$. F is the set of final configurations of $E(S)$.*
- *Δ is an (infinite) set of transitions of the form $\tau = (l_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (l_j, \mathcal{I}_j, \alpha_j)$ such that μ_i is allowed from $(l_i, \mathcal{I}_i, \alpha_i)$ (i.e., τ satisfies the* consecution *condition of definition 2).*

Any configuration of the extended state machine belongs in a path from an initial configuration to a final configuration. A run of $E(S)$ is any finite path from an initial configuration of $E(S)$ to one of its final configurations. Given an initial configuration id_0 of $E(S)$, all the possible runs of $E(S)$ starting from id_0 form an (infinite) execution tree having id_0 as its root. Hence, due to the infinite number of initial databases, all the runs of service S are captured in an (infinite) forest, that contains all possible execution trees of $E(S)$ (i.e., the set of trees having as a root an initial configuration id with $id \in \mathbb{Q}_0$).

2.2.5 Simulation relation

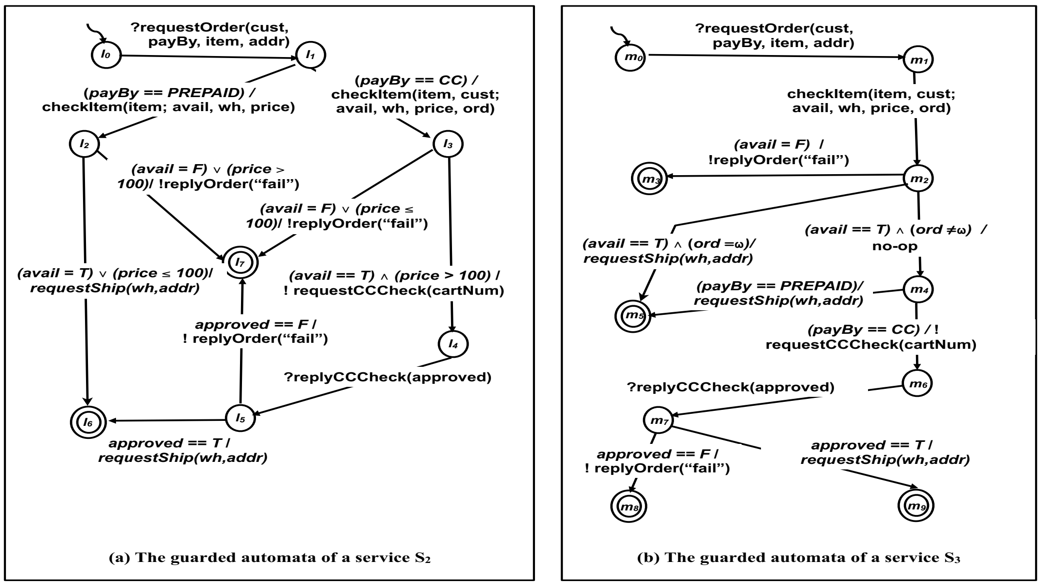
We define now the notion of simulation between two Colombo services.

Definition 4. ***(Simulation)*** *Let S and S' be two Colombo services and let $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ and $E(S') = (\mathbb{Q}', \mathbb{Q}'_0, \mathbb{F}', \Delta')$ be respectively there associated extended state machines.*

- *Let $(id, id') \in \mathbb{Q} \times \mathbb{Q}'$. The configuration $id = (l, \mathcal{I}, \alpha)$ is simulated by $id' = (l', \mathcal{I}', \alpha')$, noted $id \preceq id'$, iff :*
  - *if $id \in \mathbb{F}$ then $id' \in \mathbb{F}'$ and*
  - *$\mathcal{I} = \mathcal{I}'$, and*
  - *$\forall id \xrightarrow{\mu} id_j \in \Delta$, there exists $id' \xrightarrow{\mu'} id'_l \in \Delta'$ such that $\mu = \mu'$ and $id_j \preceq id'_l$*
- *The extended state machine $E(S)$ is simulated by the extended state machine $E(S')$, noted $E(S) \preceq E(S')$, iff $\forall id_0 \in \mathbb{Q}_0, \exists id'_0 \in \mathbb{Q}'_0$ such that $id_0 \preceq id'_0$*
- *A Colombo service S is simulated by a Colombo service S', noted $S \preceq S'$, iff $E(S) \preceq E(S')$.*

Informally, if $S \preceq S'$, this means that S' is able to faithfully reproduce the external visible behavior of S. The external visible behavior of a service is defined here with respect to the content of the world database as well as the exchanged *concrete* messages (i.e., message name together with the values of the message parameters). The existence of a simulation relation ensures that each execution tree of S is also an execution tree of S' (in fact, a subtree of S'), modulo a relabeling of control states.

Example 6. Consider the Colombo services S_2 and S_3 depicted at figure 2.5. We assume that these services use the same world database schema as the service S_1 of figure 2.1. An


?requestOrder(cust, payBy, item, addr)
l0
l1
(payBy == PREPAID) / checkItem(item; avail, wh, price)
(payBy == CC) / checkItem(item, cust; avail, wh, price, ord)
l2
l3
(avail = F) ∨ (price > 100)/ !replyOrder("fail")
(avail = F) ∨ (price ≤ 100)/ !replyOrder("fail")
l7
(avail = T) ∨ (price ≤ 100)/ requestShip(wh,addr)
(avail == T) ∧ (price > 100) / ! requestCCCheck(cartNum)
approved == F / ! replyOrder("fail")
l4
?replyCCCheck(approved)
l6
l5
approved == T / requestShip(wh,addr)
?requestOrder(cust, payBy, item, addr)
m0
m1
checkItem(item, cust; avail, wh, price, ord)
(avail = F) / !replyOrder("fail")
m3
m2
(avail == T) ∧ (ord =ω)/ requestShip(wh,addr)
(avail == T) ∧ (ord ≠ω) / no-op
(payBy == PREPAID)/ requestShip(wh,addr)
m4
m5
(payBy == CC) / ! requestCCCheck(cartNum)
?replyCCCheck(approved)
m6
m7
approved == F / ! replyOrder("fail")
approved == T / requestShip(wh,addr)
m8
m9

(a) The guarded automata of a service S_2

(b) The guarded automata of a service S_3

Figure 2.5 – Examples of Colombo services.

interesting question is to compare the three services with respect to there external visible behaviours. For example, although the automata of the services S_1 and S_2 look different, service S_1 is in fact simulated by service S_2 (i.e., $\mathsf{S}_1 \preceq \mathsf{S}_2$) which means that any behaviour of S_1 can be reproduced by S_2.

In contrast, even if service S_3 looks more general than S_1, the two services are in fact not comparable w.r.t. simulation relation (i.e., $\mathsf{S}_1 \not\preceq \mathsf{S}_3$ and $\mathsf{S}_3 \not\preceq \mathsf{S}_1$). One can see that S_1 does not simulate S_3 because S_3 allows the $\mathsf{PREPAID}$ payment mode for any item while S_1 accepts the $\mathsf{PREPAID}$ payment mode only for items having a price less than $\mathsf{100}$. The service S_3 does not simulate S_1 because if a payment by credit card (payment mode CC) is approved, the service S_1 sends a message $!\mathsf{requestShip}(\mathsf{wh}, \mathsf{addr})$ before terminating the execution while service S_3 never sends such a message.

2.3 Undecidability of simulation in unbounded Colombo

We shall show that the simulation problem is undecidable for Colombo services.

Problem 1. Let S and S' be two Colombo services. The simulation problem, noted $\mathsf{CheckSim}(\mathsf{S}, \mathsf{S}')$, is the problem of deciding whether $S \preceq S'$.

We start by establishing a connection between the problems of *state reacheability* and checking simulation between services. We exploit then this connection to establish undecidability of simulation.

Let us first define the state reachability problem for *Colombo* services.

Problem 2. Let S be a Colombo service and $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ its extended state machine. Let $l \in Q$ be a control state in $GA(S)$. The reachability problem, noted $\mathsf{reach}(E(S), l)$, is the following : Is there a database $\mathcal{J}$ over the scheme $\mathcal{W}$ and a valuation α over $LStore(S)$ such that the configuration $(l, \mathcal{J}, \alpha)$ appears in a run of $E(S)$?

Example 7. An example of a reachability problem is to ask whether the configuration $id_2 = (l_2, \mathcal{I}_2, \alpha_2)$ depicted at figure 2.4 is reachable by our simple $\mathsf{Warehouse}$ service S_1. The answer in this case is yes since, as illustrated in the previous example, the configuration id_2 can be reached from the initial configuration id_0 shown at figure 2.2.

We exhibit the following straightforward link between simulation and reachability.

Theorem 1. *If the reachability problem for a given class of Colombo service is undecidable so the simulation is also undecidable in this class.*

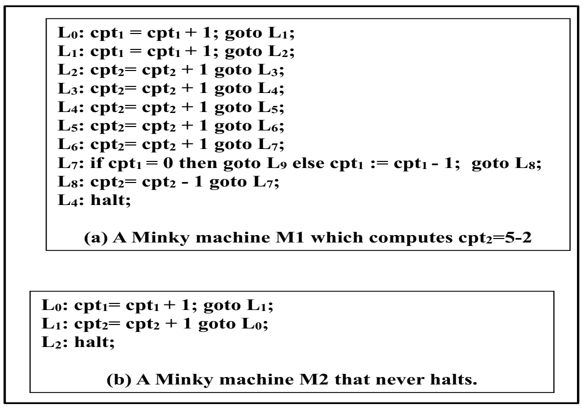
Démonstration. (sketch)
Let S be a Colombo service and l be a state in $GA(S)$. w.l.o.g., we assume that for any transition (l', c, μ, l) of $GA(S)$, the label μ is unique (i.e., μ do not appear in any another transition of $GA(S)$). Then, given the reachability problem $\mathsf{reach}(E(S), l)$, we build a new service S', such that $GA(S')$ is obtained from $GA(S)$ by deleting the state l. Consider now the simulation problem $CheckSim(S, S')$. Hence in this case, it is easy to prove that $S \preceq S'$ iff l is not reachable in $E(S)$.

$\square$

Let us consider now the reachability problem in Colombo.

Lemma 1. *The reachability problem in Colombo is undecidable.*

The proof of this lemma is achieved by a reduction from halting problem of a Minsky machine [Min67]. A Minsky machine M consists of two nonnegative *counters*, cpt_1 and



```
L0: cpt1 = cpt1 + 1; goto L1;
L1: cpt1 = cpt1 + 1; goto L2;
L2: cpt2= cpt2 + 1 goto L3;
L3: cpt2= cpt2 + 1 goto L4;
L4: cpt2= cpt2 + 1 goto L5;
L5: cpt2= cpt2 + 1 goto L6;
L6: cpt2= cpt2 + 1 goto L7;
L7: if cpt1 = 0 then goto L9 else cpt1 := cpt1 - 1;  goto L8;
L8: cpt2= cpt2 - 1 goto L7;
L4: halt;
```

(a) A Minky machine M1 which computes cpt_2=5-2

```
L0: cpt1= cpt1 + 1; goto L1;
L1: cpt2= cpt2 + 1 goto L0;
L2: halt;
```

(b) A Minky machine M2 that never halts.

Figure 2.6 – Example of two Minsky machines.

cpt_2, and a sequence of labelled instructions :

$$\mathsf{L}_0 : \ \mathsf{instr}_0;\ \mathsf{L}_1 : \mathsf{instr}_1; \ldots \mathsf{L}_{n\text{-}1} : \mathsf{instr}_{n\text{-}1};\ \mathsf{L}_n : \mathsf{halt}$$

where each of the first n instructions has one of the following forms :

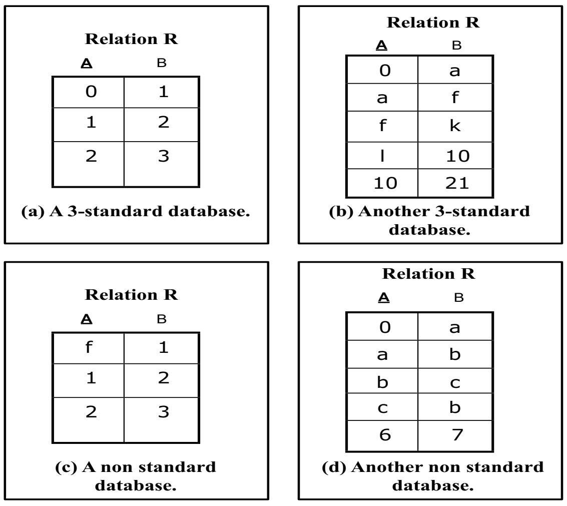
1. $\mathsf{L}_i : \mathsf{cpt}_k := \mathsf{cpt}_k + 1;\ \mathsf{goto}\ L_j$, or
2. $\mathsf{L}_i : \mathsf{if}\ \mathsf{cpt}_k = 0\ \mathsf{then}\ \mathsf{goto}\ L_j\ \mathsf{else}\ cpt_k := cpt_k\text{-}1;\ \ \mathsf{goto}\ L_l$.
   with $k \in \{1, 2\}$, $i \in [0, n\text{-}1]$ and $j, l \in [0, n]$.

A machine M starts its execution with counters $cpt_1 = cpt_2 = 0$ and the control at label L_0. Then, when the control is at a label $\mathsf{L}_i, i \in [0, n\text{-}1]$, the machine executes the instruction instr_i and jumps to the appropriate label as specified in this instruction. The machine M halts if the control reaches the halt instruction at label L_n.

Example 8. Figure 2.6(a) shows an example of a Minsky machine $\mathsf{M1}$ which computes the difference operation $5-2$ at the counter cpt_2. The seven first lines L_0 to L_6 of $\mathsf{M1}$ are used to initialize the counters $\mathsf{cpt}_1 := 2$ and $\mathsf{cpt}_2 = \mathsf{5}$. Then, the machine $\mathsf{M1}$ loops on the lines L_7 and L_8 to compute the difference $\mathsf{cpt}_2 - \mathsf{cpt}_1$ and halts. Figure 2.6(b) shows a Minsky machine $\mathsf{M2}$ that never halts. An execution of such a machine leads to an infinite sequence : $(\mathsf{L}_0, \mathsf{cpt}_1 = \mathsf{0}, \mathsf{cpt}_2 = \mathsf{0}), (\mathsf{L}_1, \mathsf{cpt}_1 = \mathsf{1}, \mathsf{cpt}_2 = \mathsf{0}), (\mathsf{L}_0, \mathsf{cpt}_1 = \mathsf{1}, \mathsf{cpt}_2 = \mathsf{1}), \ldots$

It is known that the halting problem of Minsky machines, i.e., whether the execution of a given machine halts, is undecidable even in the case when the two counters are initialized to zero [Min67].

Given a Minsky machine M, we construct a Colombo service S_M that captures the execution of M. S_M uses a world database schema containing a single binary relation schema (i.e., $\mathcal{W} = \{R(A;B)\}$). The main idea to simulate a machine M is to make S_M



Relation R

A	B
0	1
1	2
2	3

(a) A 3-standard database.

Relation R

A	B
0	a
a	f
f	k
l	10
10	21

(b) Another 3-standard database.

Relation R

A	B
f	1
1	2
2	3

(c) A non standard database.

Relation R

A	B
0	a
a	b
b	c
c	b
6	7

(d) Another non standard database.

FIGURE 2.7 – Examples of standard and non-standard world databases.

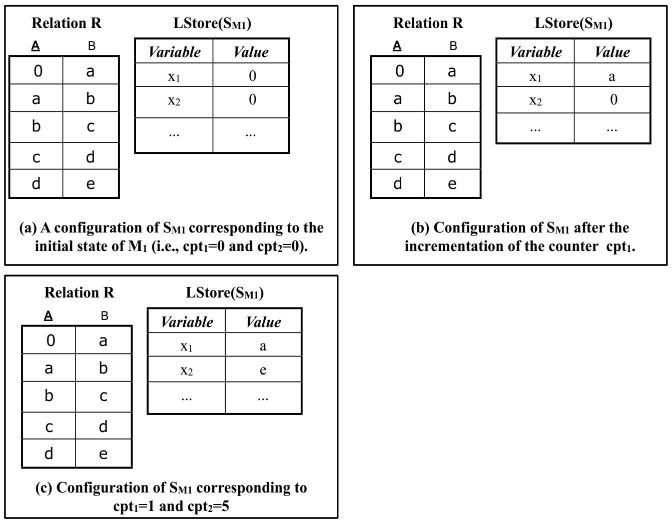
working only on parts of instances of R that form a *chain* having the constant 0 as a root. A chain of length k is any set $\Upsilon_k = \{(c_0, c_1), \ldots, (c_{k-1}, c_k) \mid \forall i \in [0, k\text{-}1], c_i \text{ is a constant}\}$. The constant c_0 is called the root of Υ_k. For a pair $(c_{l\text{-}1}, c_l) \in \Upsilon_k$, we note by $d(c_l) = l$ the *distance* of c_l with respect to the root c_0 in the chain Υ_k. An instance I of R is said k-standard if there exists a chain Υ_k such that $\Upsilon_k \subseteq I$ and $c_0 = 0$. Hence, a k-standard instance contains a chain of length k that starts with pair $(0, c_1)$.

Example 9. Figures 2.7(a) and (b) show examples of two 3-standard databases. Each of the relations of these figures contains a chain of length 3 starting from the root $\mathsf{0}$: the chain $\Upsilon_3 = \{(\mathsf{0},\mathsf{1}), (\mathsf{1},\mathsf{2}), (\mathsf{2},\mathsf{3})\}$ of figure 2.7(a) and the chain $\Upsilon_3 = \{(\mathsf{0},\mathsf{a}), (\mathsf{a},\mathsf{f}), (\mathsf{f},\mathsf{k})\}$ of figure 2.7(b). Note that, the relation R of figure 2.7(b) contains two additional tuples $(\mathsf{l},\mathsf{10})$ and $(\mathsf{10},\mathsf{20})$ that do not belong to the chain $\Upsilon_3 = \{(\mathsf{0},\mathsf{a}), (\mathsf{a},\mathsf{f}), (\mathsf{f},\mathsf{k})\}$. These two tuples will never be accessed by the constructed Colombo services (i.e., the constructed Colombo services can *see* only the elements of a chain rooted at the constant 0). Figures 2.7(c) and (d) show examples of non-standard databases. The database at figure 2.7(c) is non 1-standard because it does not contain the constant $\mathsf{0}$ while the database at figure 2.7(d) is non-standard because it includes a chain with a cycle (i.e., the chain $\Upsilon_3 = \{(\mathsf{0},\mathsf{a}), (\mathsf{a},\mathsf{b}), (\mathsf{b},\mathsf{c}), (\mathsf{c},\mathsf{b})\}$).

To simulate the counters *cpt*$_1$ and *cpt*$_2$ during an execution of M, S_M uses respectively two variables, namely x_1 and x_2 (hereafter called counter variables), of its LStore. The variables x_1 and x_2 are initially set to 0. Intuitively, a value of a counter cpt_j, with $j \in \{1, 2\}$, is captured by the *distance* between the current value of the variable x_j w.r.t. to the root 0 of the chain (i.e., $cpt_j = d(x_j)$). Hence, a given counter cpt_j of a minsky machine M is equal to 0 iff its corresponding counter variable x_j is equal to 0 (with $j \in \{1, 2\}$). Incrementing a counter cpt_j is captured in S_M by moving forward the corresponding variable x_j in the chain Υ_k while decreasing a counter amounts to moving x_k backward in the chain.

Example 10. Figure 2.8 shows some configurations of a Colombo service $\mathsf{S_{M1}}$ used to simulate the Minsky machine $M1$ of figure 2.6(a). The local store of $\mathsf{S_{M1}}$ includes among others the variables $\mathsf{x_1}$ and $\mathsf{x_2}$ which are respectively used to simulate the counters *cpt*$_1$ and *cpt*$_2$ of $M1$. The initial state of $M1$, i.e., $cpt_1 = cpt_2 = 0$, corresponds to the configuration of $\mathsf{S_{M1}}$ depicted at figure 2.8 (a). In this configuration, both $\mathsf{x_1}$ and $\mathsf{x_2}$ are set to $\mathsf{0}$. Figure 2.8 (b) shows the configuration of $\mathsf{S_{M1}}$ after the incrementation of the counter *cpt*$_1$ of $M1$ while figure 2.8 (c) shows a configuration of $\mathsf{S_{M1}}$ corresponding to a state of $M1$ where $cpt_1 = 1$ and $cpt_2 = 5$.

Moreover, to be able to simulate correctly an execution of a Minsky machine M, a service S_M requires an input database which is at least k_{max}-standard where k_{max} is the maximum value reached by the counters *cpt*$_1$ and *cpt*$_2$ of M in the considered execution. Hence, during its execution a service S_M needs to continuously check that


Relation R

A	B
0	a
a	b
b	c
c	d
d	e

LStore(S$_{M1}$)

Variable	Value
x$_1$	0
x$_2$	0
...	...

(a) A configuration of S$_{M1}$ corresponding to the initial state of M$_1$ (i.e., cpt$_1$=0 and cpt$_2$=0).

Relation R

A	B
0	a
a	b
b	c
c	d
d	e

LStore(S$_{M1}$)

Variable	Value
x$_1$	a
x$_2$	0
...	...

(b) Configuration of S$_{M1}$ after the incrementation of the counter cpt$_1$.

Relation R

A	B
0	a
a	b
b	c
c	d
d	e

LStore(S$_{M1}$)

Variable	Value
x$_1$	a
x$_2$	e
...	...

(c) Configuration of S$_{M1}$ corresponding to cpt$_1$=1 and cpt$_2$=5

Figure 2.8 – Examples of configurations of a service S_{M_1} which simulates the Minsky Machine $M1$.

the current database is k_{max}-standard. Due to the limited expressivity of the Colombo model, the implementation of such verification operations as well as the incrementation and decrementation of the counter variables x_1 and x_2 are not straightforward. We explain below in more details how the service S_M is constructed.

Let M be a Minsky machine defined as above. We associate to M, a Colombo service S_M, called the corresponding service of M, with the guarded automata $GA(S_M) = \langle Q, \delta, q_{start}, F, LStore(S)\rangle$. The set of states Q contains among other states, a state q_{L_i} for each label L_i in M, with $i \in [1, n\text{-}1]$, the initial state q_{start} and two final states q_{fail} and

q_{halt}. The state q_{halt} corresponds to the label L_n of the halt instruction of M. An execution of S_M ends at the final state q_{halt} if the corresponding Minsky machine execution halts. An execution of S_M reaches the final state q_{fail} every time it is given as input an initial database which is not k_{max}-standard. To achieve this task, the service S_M uses a boolean variable noted x_{flag} to control the conformity of the current database : x_{flag} is initialized to true and then it is set to false if during a given execution the service finds out that the current database is not k_{max}-standard. Setting the boolean variable x_{flag} to false, will make the execution moving to the final state q_{fail}.

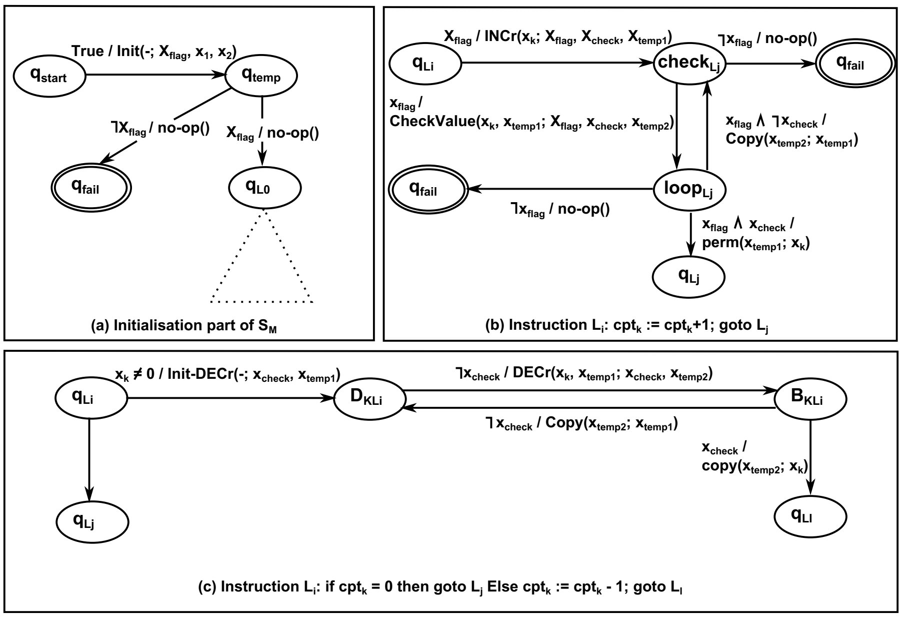
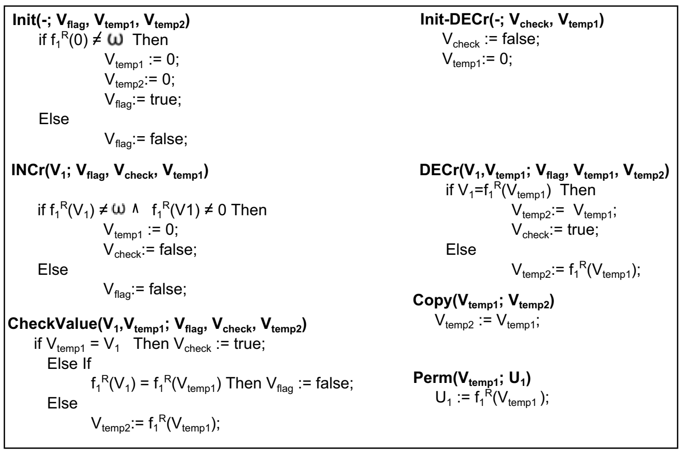

True / Init(-; X_{flag}, x_1, x_2)
q_{start}
q_{temp}
$\neg X_{flag}$ / no-op()
X_{flag} / no-op()
q_{fail}
q_{L0}
(a) Initialisation part of S_M
X_{flag} / INCr(x_k; X_{flag}, X_{check}, X_{temp1})
$\neg x_{flag}$ / no-op()
q_{Li}
$check_{Lj}$
q_{fail}
x_{flag} / CheckValue(x_k, x_{temp1}; X_{flag}, x_{check}, x_{temp2})
$x_{flag} \wedge \neg x_{check}$ / Copy(x_{temp2}; x_{temp1})
q_{fail}
$loop_{Lj}$
$\neg x_{flag}$ / no-op()
$x_{flag} \wedge x_{check}$ / perm(x_{temp1}; x_k)
q_{Lj}
(b) Instruction L_i: $cpt_k := cpt_k+1$; goto L_j
$x_k \neq 0$ / Init-DECr(-; x_{check}, x_{temp1})
$\neg x_{check}$ / DECr(x_k, x_{temp1}; x_{check}, x_{temp2})
q_{Li}
D_{KLi}
B_{KLi}
$\neg x_{check}$ / Copy(x_{temp2}; x_{temp1})
x_{check} / copy(x_{temp2}; x_k)
q_{Lj}
q_{Ll}
(c) Instruction L_i: if $cpt_k = 0$ then goto L_j Else $cpt_k := cpt_k - 1$; goto L_l

Figure 2.9 – Sub-processes of S_M.



```
Init(-; $V_{flag}$, $V_{temp1}$, $V_{temp2}$)
    if $f_1^R(0) \neq \omega$ Then
            $V_{temp1} := 0$;
            $V_{temp2} := 0$;
            $V_{flag} :=$ true;
    Else
            $V_{flag} :=$ false;

INCr($V_1$; $V_{flag}$, $V_{check}$, $V_{temp1}$)

    if $f_1^R(V_1) \neq \omega \wedge f_1^R(V1) \neq 0$ Then
            $V_{temp1} := 0$;
            $V_{check} :=$ false;
    Else
            $V_{flag} :=$ false;

CheckValue($V_1$,$V_{temp1}$; $V_{flag}$, $V_{check}$, $V_{temp2}$)
    if $V_{temp1} = V_1$ Then $V_{check} :=$ true;
      Else If
            $f_1^R(V_1) = f_1^R(V_{temp1})$ Then $V_{flag} :=$ false;
      Else
            $V_{temp2} := f_1^R(V_{temp1})$;

Init-DECr(-; $V_{check}$, $V_{temp1}$)
    $V_{check} :=$ false;
    $V_{temp1} := 0$;

DECr($V_1$,$V_{temp1}$; $V_{flag}$, $V_{temp1}$, $V_{temp2}$)
    if $V_1 = f_1^R(V_{temp1})$ Then
            $V_{temp2} := V_{temp1}$;
            $V_{check} :=$ true;
    Else
            $V_{temp2} := f_1^R(V_{temp1})$;

Copy($V_{temp1}$; $V_{temp2}$)
    $V_{temp2} := V_{temp1}$;

Perm($V_{temp1}$; $U_1$)
    $U_1 := f_1^R(V_{temp1})$;
```

Figure 2.10 – Atomic processes of the Colombo service S_M.

Figure 2.9 shows fragment of a Colombo service used to model the two kinds of instructions used by Minsky machines while figure 2.10 describes the associated atomic processes. Figure 2.9 (a) depicts the initialisation of a service S_M. An execution of such a service starts by executing the atomic process init and moves to the state q_{temp}. The init process checks that the initial database is 1-standard (i.e., it contains a tuple $(0, c_1)$) and in this case sets the counter variables to 0 and the boolean variable x_{flag} to true. In case the initial database is not 1-standard, the variable x_{flag} is set to false which will make the execution moving from state q_{temp} to the final state q_{fail}.

Example 11. Consider again a Colombo service $\mathsf{S_{M_1}}$ which simulates the Minsky Machine $M1$. By construction, the guarded automaton of such a service includes the initialisation part depicted at figure 2.9(a). Therefore, if the service $\mathsf{S_{M_1}}$ is given as initial database the non 1-standard database of figure 2.7(c), the service starts by executing the atomic process Init of figure 2.10 and moves to the state $\mathsf{q_{temp}}$. As an effect of the execution of the atomic process Init, the variable $\mathsf{V_{flag}}$ is set to false during this transition. Indeed, when evaluated over the non 1-standard database of figure 2.7(c), the condition $(\mathsf{f_1^R(0)} \neq \omega)$ of the Init process returns false and hence the effect $\mathsf{V_{flag}} := \mathsf{false}$ specified in the Else branch is applied. At state $\mathsf{q_{temp}}$, the only possible transition for service $\mathsf{S_{M_1}}$ is to move to the final state $\mathsf{q_{fail}}$ and terminate the execution. Hence, when given any non 1-standard initial database, the service $\mathsf{S_{M_1}}$ always terminates at state $\mathsf{q_{fail}}$ (and can never reach the state $\mathsf{q_{halt}}$).

Figure 2.9(b) depicts part of a service that implements Minsky machine instructions of type $1 : \mathsf{L_i} : \mathsf{cpt_k} := \mathsf{cpt_k} + 1;\ \mathsf{goto}\ L_j$ (i.e., incrementation of a counter cpt_k, with $k \in \{1, 2\}$). As explained above, incrementation amounts to moving forward in the chain the corresponding counter variable x_k. Assume that the current value of the variable x_k is $x_k = c_l$, with c_l a constant. The incrementation of x_k requires to : (i) first check that $f_1^R(x_k) \neq \omega$ (i.e., the chain is long enough to handle the new value of the counter), and (ii) check that $f_1^R(c_l)$ is a new value which has not already appeared in the chain. These two conditions ensure that the considered database is k-standard (with $k = d(c_l)+1$). The first condition is easy to check (c.f., atomic process INCr) while the second one is handled by reading the chain starting from the root until the tuple (c_{l-1}, c_l) and checking at each step whether the value $f_1^R(c_l)$ has already appeared or not. To achieve this task, an execution of S_M enters the state $check_{L_j}$ and then recursively calls the atomic process $\mathsf{CheckValue}$ starting from the root $(0, c_1)$ of the chain (c.f., loop between the states $Check_{L_j}$ and $Loop_{L_j}$ in figure 2.9(b)). The execution exits from the loop in two cases : (i) either it reaches to tuple (c_{l-1}, c_l), which means that the current database is k-standard (with $k = d(c_l)+1$) and hence the service moves to the state q_{L_j} and continue the execution, or (ii) it reaches a tuple $(c_i, f_1^R(c_l))$ in the chain which means that the database is not k-standard (with $k = d(c_l)+1$) and hence the service moves to the final state q_{fail}.

Example 12. Let us illustrate the incrementation of a counter on the non-standard database of figure 2.7(d). Consider the state of the Minsky machine $M1$ of figure 2.6 after the execution of the lines $\mathsf{L0}$ to $\mathsf{L4}$: the current values of the counters are $cpt_1 = 2$ and $cpt_2 = 3$ and the current line is $\mathsf{L5}$. This state corresponds to a configuration of the $\mathsf{S_{M1}}$ service with a current state q_{L_5} and the counter variables having as values : $\mathsf{x_1} = \mathsf{b}$ and $\mathsf{x_2} = \mathsf{c}$. Note that, in the considered database, the distance of the constant b to the root is equal to 2 while the distance of c to the root is equal to 3 (i.e., $\mathsf{d(b)} = 2$ and $\mathsf{d(c)} = 3$). Hence, such a configuration corresponds to a state of the Minsky machine $M1$ with the counter cpt_1 equal to 2 and the counter cpt_2 equal to 3 . The line $\mathsf{L5}$ of $M1$ increments cpt_2 and moves to line $\mathsf{L6}$. Let us see how such an incrementation is implemented by the Colombo service $\mathsf{S_{M1}}$. Following the automaton of figure 2.9(b), $\mathsf{S_{M1}}$ calls the atomic process INCr and moves from state q_{L_5} to state $check_{L_6}$. The execution of the atomic process INCr checks that the current chain is long enough to handle the new value of the counter. This is the case in the considered database since the condition $(\mathsf{f_1^R(x_2)} \neq \omega) \wedge (\mathsf{f_1^R(x_2)} \neq 0)$ evaluates to true over the non-standard database of figure 2.7(d) (indeed, we have $(\mathsf{f_1^R(x_2)} = \mathsf{f_1^R(c)} = \mathsf{b})$. But before assigning the constant b to the variable $\mathsf{x_2}$, the service $\mathsf{S_{M1}}$ enters the state

$check_{L_6}$ and checks whether or not b is a new constant in the chain (i.e., b does not already appear in the chain). This verification is achieved by iterating on the chain from the root $\mathsf{0}$ to the current value x_2 (i.e., the constant c) and checking at each iteration that the constant b do not belong to the chain (loop between the states $check_{L_6}$ and $loop_{L_6}$ and call to the atomic process CheckValue in the automaton of figure 2.9(b)). In the considered database, the first iteration reads the tuple $(\mathsf{0}, \mathsf{a})$ of the chain while the second iteration reads the tuple (a, b). The service S_{M1} is then able to detect that there is cycle in the chain because the constant b appears twice and hence the considered database is not standard. Hence, the service will move to state q_{fail} and terminates the execution.

We consider now the implementation of instructions of type 2 : $\mathsf{L_i}$: if $\mathsf{cpt_k} = 0$ then goto L_j else cpt_k := cpt_k-1 then goto L_l (c.f. figure 2.9(c)). The main difficulty here lies in the implementation of the decrementation operation (which amounts to moving back the counter x_k in the chain). Assume that the current value of x_k is c_l. Decrementing x_k amounts to assigning to x_k the constant c such that $f_1^R(c) = c_l$. To find the constant c one needs to read again the chain starting from the root. In the service S_M this is implemented by first entering the state D_{kL_i}, by executing the Init-Decr process, and then recursively calling the atomic process DECRr (c.f., loop between the states D_{kL_i} and $B_{k_{L_l}}$ of figure 2.9(c)) to explore the chain starting from the root and stopping at the tuple (c, c_l) (we are sure that such a tuple exist because during the incrementation step to reach the value c_l, the database has been checked to be at least $d(c_l)$-standard).

Example 13. Consider again a configuration of the S_{M1} service with the database of figure 2.7(d) and the counter variable $\mathsf{x}_2 = \mathsf{c}$ (i.e., corresponding to the counter $cpt_2 = 3$). To decrement x_2, the service S_{M1} reads the chain from the root $\mathsf{0}$ and stops the tuple (b, c) (third tuple of the database) because we have $(\mathsf{f}_1^{\mathsf{R}}(\mathsf{b}) = \mathsf{c}$ (and hence $\mathsf{d}(\mathsf{b}) = \mathsf{d}(\mathsf{c}) - 1 = 2$). The constant b is then assigned as the new value for the variable x_1 (which corresponds to a counter $cpt_2 = 2$).

We give now the main property of the proposed construction that enables to prove lemma 1.

Lemma 2. *Let M be a Minsky machine and S_M the corresponding Colombo service, then : M halts iff* reach($E(S_M)$, q_{halt})

This result is obtained from the connection that exists between executions of M and the executions of S_M that use as input a k-standard databases. In particular, the different values taken by the counter cpt_1 and cpt_2 during an execution of M are captured by the distances of the counter variables x_1 and x_2 during the execution of S_M. Hence, it is possible to map any execution of M into an execution of S_M on a k-standard database and conversely. Moreover, it is easy to show that if there exists an execution of M that halts and in which k_{max} is the maximum value reached by the counters of M, then the execution of the corresponding service S_M using a k_{max}-standard initial database terminates at the final state q_{halt}. On the other side, by construction, S_M terminates at the final state q_{halt} iff it takes as initial database a k-standard database (which hence can be mapped into an execution of M that halts).

From theorem 1 and lemma 1, we obtain the following main result regarding simulation in the Colombo model.

Theorem 2. *Let S and S' be two Colombo services, then* $\mathsf{CheckSim}(\mathsf{S}, \mathsf{S}')$ *is undecidable.*

Finally, the following theorem can be straightforwardly derived from the previous proof since the constructed service S_M is a non-communicating read-only Colombo service.

Theorem 3. *Let S and S' be two non-communicating services with read-only accesses to the world database and let l be a control state in $GA(S)$, then both* $\mathsf{CheckSim(S,S')}$ *and* $\mathsf{reach(E(S)}$*, l) are undecidable.*

The reduction from the halting problem of Minsky machine is possible because a non-communicating Colombo services with read-only accesses can access to an unbounded number of tuples in the database. In the next sections, we will prove that when the number of tuples acceded is bounded the simulation for Colombo model is decidable. This is done by a semantic restriction on the Colombo model and the resulting model is $Colombo^{bound}$. A $Colombo^{bound}$ service has a shared database with a number of tuples that cannot exceed a given constant k). To prove the decidability of the simulation for $Colombo^{bound}$ services, we use an intermediary class named $Colombo^{DB=\emptyset}$. $Colombo^{DB=\emptyset}$ services or DB-less services are Colombo services without any access to the database.

2.4 Decidability of simulation in DB-less Colombo

We investigate in this section the simulation problem in the setting of a Colombo model without a global database (i.e., we assume the world schema $\mathcal{W} = \emptyset$).
Let S be a $Colombo^{db=\emptyset}$ service. The associated state machine is a tuple $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$. A configuration of $E(S)$ has the form $id = (l, \emptyset, \alpha)$ while there is only one initial configuration $id_0 = (l_0, \emptyset, \alpha_0)$ with $\alpha_0(x) = \omega$, $\forall x \in LStore(S)$. Moreover, in $Colombo^{db=\emptyset}$ services, atomic processes can only assign constants to variables of $LStore(S)$ or assign value of a variable to another. Note that $E(S)$ is still an infinite state system. This is due to the presence of input messages with parameters taking their values from a possibly infinite domain. We describe below a symbolization technique that allows to abstract from concrete values and hence turns extended machines associated with $Colombo^{db=\emptyset}$ services into finite state machines.

Notation and basic notions. Let X be a set of variables taking their values from an infinite domain $\mathcal{D} \cup \{\omega\}$. Let θ be a condition on a set of variables X and let α be a valuation over X. Then $\theta(\alpha)$ is the condition obtained by replacing each variable x appearing in θ by $\alpha(x)$. We say that α satisfies θ, noted $\alpha \models \theta$, if $\theta(\alpha) = \mathsf{true}$. A valuation α satisfies a set Θ of conditions, noted $\alpha \models \Theta$, if $\alpha \models \theta$, $\forall \theta \in \Theta$.

Let $K = \{c_1, \ldots, c_k\}$ with $c_1 < \ldots < c_k$ be a set of constants in $\mathcal{D}$. We define the set I_K of elementary intervals over K as $I_K = \{[\omega,\omega],]-\infty, c_1[,]c_k, +\infty[\} \cup \{[c_l, c_l], l \in [1,k]\} \cup \{]c_l, c_{l+1}[, l \in [1, k-1]\}$. Note that, a set of intervals I_K forms a partition of the domain $\mathcal{D} \cup \{\omega\}$ (i.e., intervals in I_K are pairwise disjoint).

Example 14. For $K = \{4, 10\}$, the associated set of elementary intervals is $I_K = \{[\omega,\omega],]-\infty, 4[, [4,4],]4,10[, [10,10],]10,+\infty[\}$

Let X be a set of variables and $op \in \{=, <\}$. We denote by ψ a set of conditions, where each condition is defined as follows :
$\forall x, y \in X$, ψ contains $\{x = \omega\}$ or $\{y = \omega\}$ or $\{x \; op \; y\}$. ψ is called a $\mathsf{v\text{-}order}$ over X. A v-order ψ is said consistent iff it exists at least one valuation α over the variables of X such that $\alpha \models \psi$. We note by $vo(X)$ the set of all v-orders on X.

Example 15. Let $X = \{x, y\}$, then examples of v-orders over X are :

- $\psi_0 = \{x = \omega, y = \omega\}$
- $\psi_1 = \{x = y\}$
- $\psi_2 = \{x = \omega\}$
- $\psi_3 = \{y = \omega\}$
- $\psi_4 = \{x < y\}$

We use below the notion of regions to extend intervals to a set of variables.

Definition 5. ***(Regions)*** *Let $X = \{x_1, ..., x_n\}$ be a set of variables and K a set of constants. We assume variables in X ordered according to the lexicographic order. A region of X w.r.t K is a tuple $r = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi)$ with $\psi \in vo(X)$ and $\tau_{x_i} \in I_K, \forall i \in [1, n]$.*

The set of all possible regions of X w.r.t. K is denoted $R_g(X, K)$.

Hence a region $r = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi)$ associates an elementary interval τ_{x_i} with each variable $x_i \in X$.

Example 16. Let us consider the set of constants $K = \{4, 10\}$ and the set of variables $X = \{x, y\}$, with their associated elementary intervals and v-orders. The set $R_g(X, K)$ includes the following regions :

- $r_\omega = ([\omega, \omega], [\omega, \omega], \psi_0)$
- $r_1 = (]4, 10[,]4, 10[, \psi_1)$
- $r_2 = ([\omega, \omega],]-\infty, 4[, \psi_2)$
- $r_3 = ([10, 10], [\omega, \omega], \psi_3)$

In the sequel, we abuse of notation and write $r \wedge \theta$ instead of $(\tau_{x_1}, \ldots, \tau_{x_n}, \psi \wedge \theta)$ where θ is a condition. We introduce below some notation regarding regions.

- A valuation α over X belongs to a region $r = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi)$ of X, denoted $\alpha \in r$, iff $\alpha(x_i) \in \tau_{x_i}, \forall i \in [1, n]$ and $\alpha \models \psi$. The set of valuations that belong to a region r is noted $val(r)$,
- A region r is inconsistent, noted $r \models \bot$, if $val(r) = \emptyset$. In the previous example, the region r_3 is inconsistent.
- Let $r = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi)$ be a region of X. A projection of r on a set $\{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$, noted $\pi_{\{x_{i_1},\ldots,x_{i_k}\}}(r)$, is the region $\pi_{\{x_{i_1},\ldots,x_{i_k}\}}(r) = (\tau_{xi_1}, \ldots, \tau_{xi_k}, \psi_{|x_{i_1},\ldots,x_{i_k}})$, where $\psi_{|x_{i_1},\ldots,x_{i_k}}$ is the subset of ψ that contains only the conditions over $\{x_{i_1}, \ldots, x_{i_k}\} \cup \mathcal{D} \cup \{\omega\}$.
- Let $r = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi)$ and $r' = (\tau_{x_1}, \ldots, \tau_{x_n}, \psi')$ be two regions of X. We say that r coincides with r' on a set of variables $\{x_1, \ldots, x_k\} \subseteq X$, noted $r \equiv_{\{x_1,\ldots,x_k\}} r'$, if $\pi_{\{x_{i_1},\ldots,x_{i_k}\}}(r) = \pi_{\{x_{i_1},\ldots,x_{i_k}\}}(r')$

In the sequel, w.l.o.g., we assume that the set $R_g(X, K)$ contains only consistent regions. Note that, if X and K are both finite sets so $R_g(X, K)$ is also a finite set.

Lemma 3. *Let X be a finite set of variables and K a finite set of constants in Θ. Let $r \in R_g(X, K)$, then : $\forall \alpha_1, \alpha_2 \in r, \forall \Theta' \subseteq \Theta, if \alpha_1 \models \Theta'$ then $\alpha_2 \models \Theta'$.*

Démonstration. Suppose $\alpha_1 \models \Theta'$ with $\Theta' = \theta_1 \wedge \theta_2 \ldots \theta_k$. It is sufficient to prove the property for a condition $\theta \in \Theta'$. Let ψ be the v-order of r. We distinguish 4 cases :

1. θ is the condition $x = y$, then $\alpha_1 \models x = y$ implies $(x = y) \in \psi$, otherwise r is not consistent (by construction of r). Thus $\alpha_2 \models (x = y)$ since $\alpha_2 \models \psi$.
2. θ is the condition $x > y$. Similar to case 1.
3. θ is the condition $x = c$, with $c \in K$, then $\alpha_1 \models x = c$ implies $\tau_x = [c, c]$ in r (by construction of the elementary intervals). Thus $\alpha_2 \models (x = c)$ since $\alpha_2 \in r$.

4. θ is the condition $x > c$, with $c \in K$, then $\alpha_1 \models x > c$ implies that $\forall c' \in \tau_x$, $c' > c$ (by construction of the elementary intervals). Thus $\alpha_2 \models (x > c)$ since $\alpha_2(x) \in \tau_x$.

$\square$

Canonic representation of Colombo$^{\mathbf{db}=\emptyset}$ services. Given a Colombo service S, the main idea is to use the notion of regions to group together extended states of $E(S)$. Interestingly, the obtained representation, called a Colombo region automaton (defined below), is a finite state machine. We define below such state machines and then we show how they can be used to test simulation between *Colombo*$^{db=\emptyset}$ services.

W.l.o.g., we consider in the sequel only Colombo services with atomic processes having :

- disjoint input and output variables (i.e., services S that use atomic processes of the form $p(u_1, \ldots, u_n; v_1, \ldots, v_m)$ with $\{u_1, \ldots, u_n\} \cap \{v_1, \ldots, v_n\} \cap LStore(S) = \emptyset$), and
- a unique conditional effects (c, E) with $E = \{(es, ev)\}$ s.t. $es = \emptyset$ (since there are no modification on the dabase) and $ev = \{v_i := t,$ with t is either a constant or ω or an input variable$\}$.

Definition 6. **(Colombo$^{\mathbf{db}=\emptyset}$ *region automata*)** *Let $GA(S) = \langle Q, \delta, q_0, F, LStore(S)\rangle$ be a guarded automata of a Colombo$^{db=\emptyset}$ service S with $X = LStore(S) = \{x_1, \ldots, x_n\}$, and let Θ be a set of atomic conditions in $GA(S)$. Let K be a set of constants appearing in Θ. The associated Colombo$^{db=\emptyset}$ region automaton is a finite state machine $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X, K))$ defined as follows :*

- *$Q^S \subseteq Q \times R_g(X, K)$, the set of states of R^S,*
- *$q_0^S = (q_0, r_\omega)$, the initial state, where $r_\omega = ([w, w], ..., [w, w], \{(x_i = \omega), i \in [1, n]\})$.*
- *$F^S \subseteq F \times R_g(X, K)$, the set of final states,*
- *Let $r \in R_g(X, K)$. For each state (q, r) of R^S and for each transition $(q, \theta, \mu, q') \in \delta$ such that $r \wedge \theta$ is consistent then :*
  - (a) *if $\mu = !m(v_1, \ldots, v_m)$, we have $((q, r), \mu, (q', r)) \in \delta^S$.*
  - (b) *if $\mu = ?m(v_1, \ldots, v_m)$ we have $((q, r), \mu, (q', r')) \in \delta^S$ for each $r' \in R_g(X, K)$ which coincides with r on the variables $LStore(S) \setminus \{v_1, \ldots, v_m\}$.*
  - (c) *If $\mu = p(u_1, \ldots, u_n; v_1, \ldots, v_m, \{c, E\})$, we have two cases :*
    - (c-1) *if $r \wedge \theta \wedge c$ is consistent then $((q, r), p(u_1, \ldots, u_n; v_1, \ldots, v_m), (q', r')) \in \delta^S$ where r' coincides with r on the variables $LStore(S) \setminus \{v_1, \ldots, v_m\}$ and :*
      for each $i \in [1, m]$
      - *If $v_i := c \in E$, then r' includes $\tau_{v_i} = [c, c]$.*
      - *If $v_i := u_j \in E$ then r' includes $\tau_{v_i} = \tau_{u_j}$ and ψ' of r' includes $v_i = u_j$.*
      - *If $v_i := \omega \in E$ then r' includes $\tau_{vi} = [\omega, \omega]$*
    - (c-2) *if $r \wedge \theta \wedge \neg\, c$ is consistent, we have $((q, r), p(u_1, \ldots, u_n; v_1, \ldots, v_m), (q', r)) \in \delta^S$.*

It is worth noting that a region automaton constructed according to definition 6 must be cleaned to remove states that are not included in a path from the initial state to a final state. We illustrate the construction of a region automaton on the simple Colombo service depicted at figure 2.11

Example 17. The service S of figure 2.11 uses :

- a set of variables $X = LStore(S) = \{x, y\}$,
- a set of conditions $\Theta = \{(x > 5), (y > 5)\}$ used as guards in transitions or condition in the atomic process Perm.
- a set $K = \{5\}$ of constants that appear in Θ.

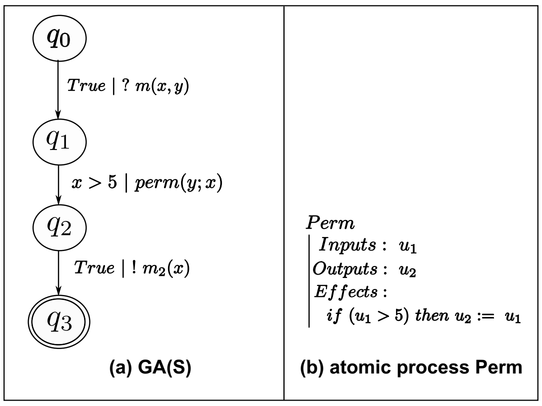


Figure 2.11 – A $Colombo^{db=\emptyset}$ service S.

Hence, the set of elementary intervals over K is :

$$I_K = \{[\omega,\omega],]-\infty, 5[, [5,5],]5,+\infty[\}$$

while the set $R_g(X,K)$ includes, among others, the following regions :
- $r_\omega = ([\omega],[\omega],\{x=\omega, y=\omega\}$
- $r_1 = ([\omega],]-\infty,5[, \{x=\omega, y=y\}$
- $r_2 = (]5,+\infty[,]5,+\infty[, \{y<x\}$
- $r_3 = (]5,+\infty[,]5,+\infty[, \{y=x\}$
- $r_4 = (]5,+\infty[,]5,+\infty[, \{x<y\}$
- $r_5 = (]5,+\infty[, [5,5], \{y<x\}$
- $r_6 = (]5,+\infty[,]-\infty,5[, \{y<x\}$
- ...

The corresponding region automaton R^S is depicted at figure 2.12. The initial state of R^S is made of the pair (q_0, r_ω). We illustrate below the cases (a), (b) and (c) of definition 6 on this region automaton.
- the transition $(q_0, ?m1(x,y), q_1)$ of $GA(S)$ (c.f., figure 2.11), is translated into a set of transitions $((q_0,r_\omega), ?m1(x,y), (q,r))$ with $r \in R_g(X,K)$ (case (b) of definition 6). This captures the fact that on a reception of a message $?m1(x,y)$, any new values may be associated to the variables x and y.
- the transition $(q_1, x>5 \mid Perm(y;x), q_2)$ of $GA(S)$, enables to a create new transition from the state (q_1, r_2) of R^S as illustrated below :
  - $((q_1,r_2), Perm(y;x), (q_2,r_3))$, this is because the region r_2 satisfies both the guard $x>5$ of the transition and the condition $u_1>5$ of the atomic process (case (c-1) of definition 6). Hence, in this case the atomic process Perm is executed. The atomic process Perm assigns variable y to the variable x, hence the region automata moves to a region where $\tau_x := \tau y$ and requires to have $x=y$ in the associated v-order. In our example, region r_3 satisfies both conditions.
  - $((q_1,r_5), Perm(y;x), (q_2,r_5))$, this is because the region r_5 satisfies the guard $x>5$ of the transition but does not satisfy the condition $u_1>5$ of the atomic process Perm (case (c-2) of definition 6). According to the Colombo semantics, the transition is fired but the atomic process Perm execute a *no-op* operation (no operation). As a consequence, the region automata moves to state q_2 while staying in the same region r_5.
- the transition $((q_2,r_5), !m2(x,), (q_3,r_5))$ (case (a) of definition 6). A send of a message does not modify values of the variables, hence upon sending the message $!m2(x)$,

the region automaton R^S moves into a new state (q_3, r_5) while staying in the same region r_5.

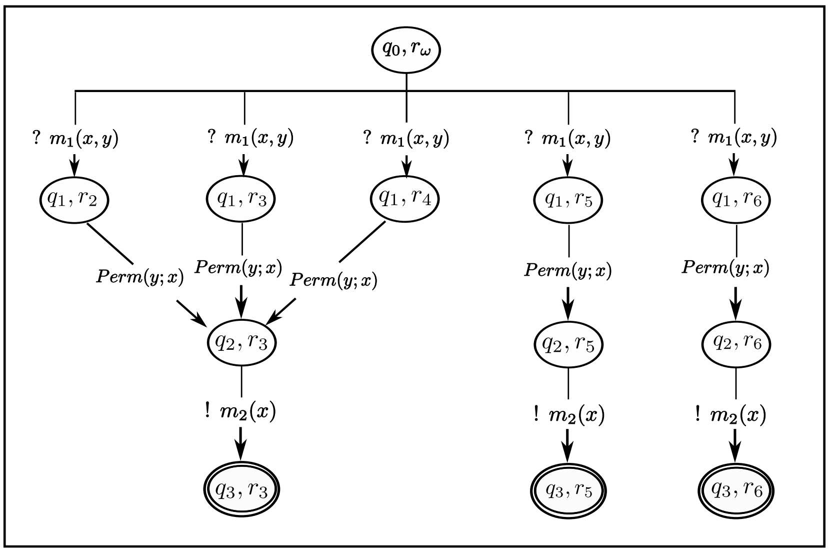


Figure 2.12 – A region automaton R^S.

In the following we show that the region automata R^S constitutes a compact representation of the extended state machine of $E(S)$ and hence it faithfully abstracts the original Colombo service S. To do so, we define the notion of unfolding of a region automaton $U_{nfold}(R^S)$ as given below.

Definition 7. *(unfolding of region automata) Let $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X,K))$ be a region automata of a service S. The associated extended state machine, noted $U_{nfold}(R^S)$, is a tuple $U_{nfold}(R^S) = (\mathbb{Q}^g, \mathbb{Q}_0^g, \mathbb{F}^g, \Delta^g)$ where :*

- $\mathbb{Q}^g = \bigcup_{r \in R_g(X,K)} \{(q,\alpha)\ s.t\ (q,r) \in Q^S, \alpha \in r\}$.
- $\mathbb{Q}_0^g = \{(q_0, \alpha_w)\}$, *with* $\alpha_w(x) = \omega$, $\forall x \in LStore(S)$.
- $\mathbb{F}^g = \bigcup_{r \in R_g(X,K)} \{(q,\alpha)\ s.t\ (q,r) \in F^S, \alpha \in r\}$..
- $\forall (q,r) \xrightarrow{\mu_i} (q',r') \in \delta^S$, *a new transition* $(q,\alpha) \xrightarrow{\mu_i} (q',\alpha')$ *is added to* Δ *such that* $\alpha \in r, \alpha' \in r'$ *and :*
  (a) *if* $\mu = !m(v_1,\ldots,v_m)$, *then* $\alpha' = \alpha$.
  (b) *if* $\mu = ?m(v_1,\ldots,v_m)$ *then* $\forall x \in LStore(S) \backslash \{v_1,\ldots,v_m\}$, *we have* $\alpha'(x) = \alpha(x)$.
  (c) *If* $\mu = p(u_1,\ldots,u_n; v_1,\ldots,v_m, \{c,E\})$, *we have two cases :*
  (c-1) *if* $r \wedge \theta \wedge c$ *is consistent then* $\forall x \in LStore(S) \setminus \{v_1,\ldots,v_m\}$, *we have* $\alpha'(x) = \alpha(x)$ *and for each* $i \in [1,m]$, *we have :*
  – *If* $v_i := k \in E$, *with* $k \in \mathcal{D} \cup \{\omega\}$, *then* $\alpha'(v_i) = k$
  – *If* $v_i := u_j \in E$ *then* $\alpha'(v_i) = \alpha(u_i)$
  (c-2) *if* $r \wedge \theta \wedge \neg\ c$ *is consistent, then* $\alpha' = \alpha$.

A run of $U_{nfold}(R^S)$ is any finite path from an initial configuration of $E(RS)$ to one of its final configurations.

Example 18. Figure 2.13(b) depicts part of the extended automata obtained by unfolding the region automata of figure 2.13(a) which corresponds to a fragment of the region automata of figure 2.12.

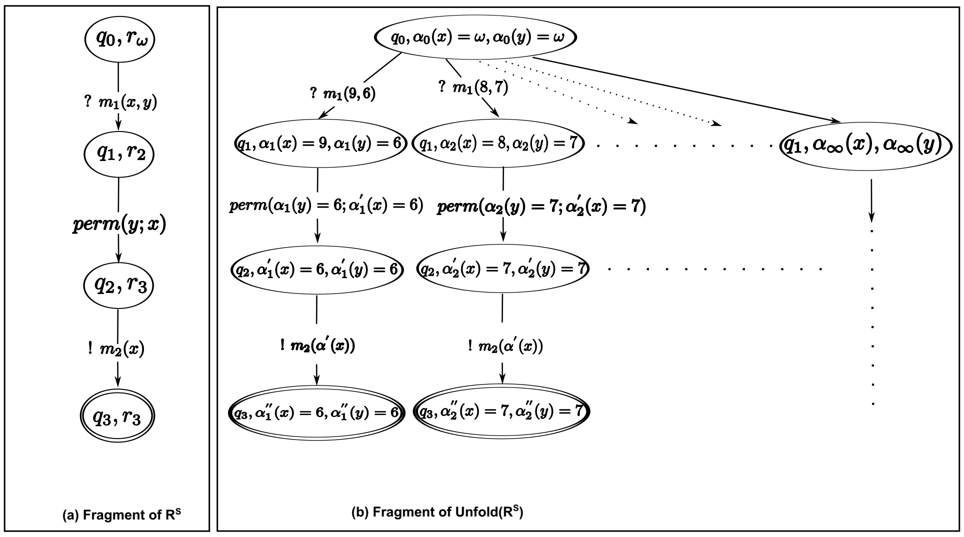

q_0, r_ω
$? m_1(x,y)$
q_1, r_2
$perm(y;x)$
q_2, r_3
$! m_2(x)$
q_3, r_3
(a) Fragment of R^S
$q_0, \alpha_0(x) = \omega, \alpha_0(y) = \omega$
$? m_1(9,6)$
$? m_1(8,7)$
$q_1, \alpha_1(x) = 9, \alpha_1(y) = 6$
$q_1, \alpha_2(x) = 8, \alpha_2(y) = 7$
$q_1, \alpha_\infty(x), \alpha_\infty(y)$
$perm(\alpha_1(y) = 6; \alpha_1'(x) = 6)$
$perm(\alpha_2(y) = 7; \alpha_2'(x) = 7)$
$q_2, \alpha_1'(x) = 6, \alpha_1'(y) = 6$
$q_2, \alpha_2'(x) = 7, \alpha_2'(y) = 7$
$! m_2(\alpha'(x))$
$! m_2(\alpha'(x))$
$q_3, \alpha_1''(x) = 6, \alpha_1''(y) = 6$
$q_3, \alpha_2''(x) = 7, \alpha_2''(y) = 7$
(b) Fragment of Unfold(R^S)

FIGURE 2.13 – Unfolding a region automaton.

The following lemma states that R^S preserves the semantics of the original Colombo service S in the sense that an unfolding of a region automaton coincides with the extended automaton of the original Colombo service.

Lemma 4. *Let $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ and $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X, K))$ with X and K defined as in definition 6. Then $E(S) = U_{nfold}(R^S)$.*

Démonstration. It is sufficient to show that a transition $(q, \emptyset, \alpha) \xrightarrow{\mu} (q', \emptyset, \alpha') \in \Delta$ iff $(q, \alpha) \xrightarrow{\mu} (q', \alpha') \in \Delta^g$.

By construction we have $(q_0, \emptyset, \alpha_0) \in \mathbb{Q}_0$ and $q_0^S = (q_0, r_w)$. Now, take $(q, \emptyset, \alpha) \xrightarrow{\mu} (q', \emptyset, \alpha') \in \Delta$. Hence, there exists $(q, \theta, \mu, q') \in GA(S)$ s.t. $\alpha \models \theta$. Let $r, r' \in R_g(X, K)$ such that $\alpha \in r$ and $\alpha' \in r'$. We show that $(q, r) \xrightarrow{\mu} (q', r') \in \delta^S$ which implies that $(q, \alpha) \xrightarrow{\mu} (q', \alpha') \in \Delta^g$. We distinguish the following three cases :

1. $\mu = !m(c_1, \ldots, c_n)$. We have $\alpha = \alpha'$ and therefore $(q, r) \xrightarrow{\mu} (q', r) \in \delta^S$ by definition 6.
2. $\mu = ?m(c_1, \ldots, c_n)$. By definition 3, we have :
$$\alpha'(x) = \begin{cases} \alpha(x) & \text{if } x \in LStore(S) \setminus \{x_1, \ldots, x_n\} \\ c_i & \text{if } x = x_i \text{ with } i \in [1, n] \end{cases}$$
$$\Rightarrow r' \text{ concides with } r \text{ on } LStore(S) \setminus \{x_1, \ldots, x_n\}$$
Moreover, we have $r \models \theta$ since $\alpha \models \theta$. So $(q, r) \xrightarrow{\mu} (q', r') \in \delta^S$.
3. $\mu = p(\alpha(u_1), \ldots, \alpha(u_n); \alpha'(v_1), \ldots, \alpha'(v_m), \{c, E\})$. We consider two cases.
   (a) $\alpha \models r \wedge \theta \wedge \neg c$: By lemma 3 all $\alpha_1 \in r, \alpha_1 \models r \wedge \theta \wedge \neg c$ and thus $r \models r \wedge \theta \wedge \neg c$. So $\alpha = \alpha'$ and therefore $(q, r) \xrightarrow{\mu} (q', r) \in \delta^S$ by Definition 6.
   (b) $\alpha \models r \wedge \theta \wedge c$: By definition 3, we have
   – $\alpha'(u_i) = \alpha(u_i)$
   – $\alpha'(v_i) = \alpha(u_i)$ or $\alpha'(v_i) = c$, where $c \in K$.
   So, r' concides with r on $LStore(S) \setminus \{v_1, \ldots, v_m\}$. Moreover $\tau_{vi} = \tau_{uj}$ or $\tau_{v_i} = [c, c]$. Thus By definition 6 we have $(q, r) \xrightarrow{\mu} (q', r') \in \delta^S$.

The other direction of the proof can be derived using a similar scheme.

□

2.4.1 Simulation between regions automata

In this section, we define a simulation relation between region automata and then we show how such a relation can be used to check simulation between two Colombo services.

Definition 8. ***(Simulation of*** **Colombo**$^{\mathbf{db}=\emptyset}$ ***region automata)*** *Let S and S' be two Colombo$^{db=\emptyset}$ services, $X = LStore(S)$, $X' = LStore(S')$ and let Θ_S (resp. $\Theta_{S'}$) be the set of atomic conditions used in $GA(S)$ (resp. $GA(S')$). Let K be the set of all constants appearing in $\Theta_S \cup \Theta_{S'}$ and let $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X,K))$ and $R^{S'} = (L^{S'}, l_0^{S'}, F^{S'}, \delta^{S'}, R_g(X',K))$ be, respectively, the region automata associated with S and S'.*

- *Let $((q,r_1),(l,r_2)) \in Q^S \times L^{S'}$ and β is a subset of the set of equalities of variables in S and S', i.e. $\{x = x'$ s.t. $x \in X, x \in X'\}$. The configuration $((q,r_1),\beta)$ is simulated by $((l,r_2),\beta)$ noted $((q,r_1),\beta) \preceq_g ((l,r_2),\beta)$ iff :*
  - *$\forall (q,r_1) \xrightarrow{\mu} (q',r_1') \in \delta^S$, there exists $(l,r_2) \xrightarrow{\mu'} (l',r_2') \in \delta^{S'}$ such that*
    1. *if $\mu = !m(x_1,\ldots,x_n)$ then $\mu' = !m(y_1,\ldots,y_n)$ and $r_1 \wedge r_2 \wedge \beta \Rightarrow x_i = y_i$ where $i \in [1,n]$ and $((q',r_1'),\beta) \preceq_g ((l',r_2'),\beta)$.*
    2. *if $\mu = ?m(x_1,\ldots,x_n)$ then $\mu' = ?m(y_1,\ldots,y_n)$ and $r_1' \wedge r_2' \wedge \beta'$ is consistent and $((q',r_1'),\beta') \preceq_g ((l',r_2'),\beta')$ where $\beta' = \{x_i = y_i, i \in [1,n]\} \cup \{z = t \in \beta$ s.t. $z \neq x_i, z \neq y_i, t \neq x_i, t \neq y_i\}$.*
    3. *if $\mu = p(x_1,\ldots,x_n;y_1,\ldots,y_m,\{c,E\})$ then $\mu' = p(u_1,\ldots,u_n;v_1,\ldots,v_m,\{c,E\})$ and $r_1 \wedge r_2 \wedge \beta \Rightarrow x_i = u_i$ and*
       - *if $r_1 \wedge c$ is consistent then $r_1' \wedge r_2' \wedge \beta' \Rightarrow y_i = v_i$ and $((q',r_1'),\beta') \preceq_g ((l',r_2'),\beta')$ where $\beta' = \{\beta\} \setminus \{z = t \in \beta$ s.t. $z = y_i, z = v_i, t = y_i, t = v_i\}$.*
       - *if $r_1 \wedge \neg c$ is consistent then $((q',r_1'),\beta) \preceq_g ((l',r_2'),\beta)$*
  - *$R^S \preceq_g R^{S'}$ iff $((q_0,r_\omega),\emptyset) \preceq_g ((l_0,r_\omega),\emptyset)$*

The following lemma ensures that the relation $\preceq_g$ captures correctly the simulation preorder on Colombo services.

Lemma 5. *Let S and S' be two Colombo$^{db=\emptyset}$ services and $X = LStore(S) \cup LStore(S')$ and Θ_S (resp. $\theta_{S'}$) be the set of atomic conditions appearing in the guards of $GA(S)$ (resp. $GA(S')$), and K be the sets of all constants appearing in $\Theta_S \cup \Theta_{S'}$. Let $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X,K))$ and $R^{S'} = (Q^{S'}, q_0^{S'}, F^{S'}, \delta^{S'}, R_g(X,K))$, then :*

$$U_{nfold}(R^S) \preceq U_{nfold}(R^{S'})\ iff\ R^S \preceq_g R^{S'}$$

Démonstration. Let $U_{nfold}(R^S) = (\mathbb{Q}^g, \mathbb{Q}_0^g, \mathbb{F}^g, \Delta^g)$ and $U_{nfold}(R^{S'}) = (\mathbb{Q}'^g, \mathbb{Q}'^g_0, \mathbb{F}'^g, \Delta'^g)$.

($\Leftarrow$) Assume that $R^S \preceq_g R^{S'}$. Take $\preceq = \{((q,\alpha_1),(l,\alpha_2))$ s.t. $((q,r_1),(l,r_2),\beta) \in \preceq_g$, with $\alpha_1 \in r_1, \alpha_2 \in r_2$ and $(\alpha_1,\alpha_2) \models \beta\}$. We show that $\preceq$ is a simulation relation (i.e., $U_{nfold}(R^S) \preceq U_{nfold}(R^{S'})$).

Clearly $((q,\alpha_w),(l,\alpha_w)) \in \preceq$ since $((q,r_w),(l,r_w),\emptyset) \in \preceq_g$. Now, suppose that $((q,\alpha_1),(l,\alpha_2)) \in \preceq$. We show that for any transition $((q,\alpha_1),\mu,(q',\alpha_1')) \in \Delta^g$, there exists a transition $((l,\alpha_2),\mu',(l',\alpha_2')) \in \Delta'^g$ such that $((q',\alpha_1'),(l',\alpha_2')) \in \preceq$.

Let $((q,\alpha_1),\mu,(q',\alpha_1')) \in \Delta^g$. Then, by lemma 4, there exists a transition $((q,r_1),\mu,(q',r_1')) \in \delta^S$ such that $\alpha_1 \in r_1$ and $\alpha_1' \in r_1'$.

By construction of $\preceq$, there exists β and $(l,r_2) \in Q^{S'}$ s.t. $((q,r_1),(l,r_2),\beta) \in \preceq_g$. Thus there exists a transition $((l,r_2),\mu',(l',r_2')) \in \delta'^S$ such that $((q',r_1'),(l',r_2'),\beta') \in \preceq_g$, since $R^S \preceq_g R^{S'}$. It suffices to take α_2' in r_2' s.t. $(\alpha_1',\alpha_2') \models \beta'$ (this is always possible since $r_1' \wedge r_2' \wedge \beta'$ is consistent). This implies that $((q',\alpha_1')(l',\alpha_2')) \in \preceq$.

($\Rightarrow$) Assume that $U_{nfold}(R^S) \preceq U_{nfold}(R^{S'})$. Following the same schema as previously, one can show that it is possible to derive from the relation $\preceq$ a relation $\preceq_g$ which can be used as a witness to deduce that R^S is simulated by $R^{S'}$. The relation $\preceq_g$ is constructed inductively, starting with $\preceq_g = \{((q, r_w), (l, r_w), \emptyset)\}$ and then recursively augmenting it with new elements by exploiting the relation $\preceq$ to identify target states and regions at each step and carefully defining the β conditions in order to cope with conditions of definition 8. The construction of $\preceq_g$ stops when a fix point is reached.

□

We provide below the main result of this section by showing that simulation between $Colombo^{db=\emptyset}$ services can be reduced to simulation between the corresponding region automata. This ensures the decidability of simulation in $Colombo^{db=\emptyset}$ setting since $Colombo^{db=\emptyset}$ region automata are finite state machines and hence exhaustive exploration of the state-space of such machines is possible.

Theorem 4. *Let S and S' be two $Colombo^{db=\emptyset}$ services then $S \preceq S'$ iff $R^S \preceq_g R^{S'}$.*

2.4.2 Complexity of simulation in DB-less services

This section is devoted to the complexity analysis of the simulation in DB-less Colombo model. We shall show that the simulation in DB-less Colombo services is EXPTIME-complete. We first show that the problem is in EXPTIME, then the EXPTIME-hardness is showed by a reduction inspired from the work of [MW07]. The reduction is obtained from the existence of an infinite execution of an alternating Turing machine M working on a space polynomially bounded by the input word size [CKS81]. That is, starting from M with an input word w of size n, we construct a test of simulation between two DB-less Colombo services $S_{spoiler}$ and $S_{duplicator}$. We prove that there exists an infinite execution of M on w iff $S_{spoiler} \preceq S_{duplicator}$.

Proposition 1. Testing the simulation for $Colombo^{db=\emptyset}$ services is EXPTIME.

Démonstration. Let S_1 and S_2 be two $Colombo^{db=\emptyset}$ services. Let K be the set of constants and X be the set of variables in S_1 and S_2. We suppose that $|K|{=}n$ is the number of constants and $|X|{=}m$ is the number of variables. The number of intervals (resp. v-orders) is bounded by $O(n)$ (resp. $O(!m)$). The number of regions is bounded by $O(n^m \times m!)$ and therefore the number of states in the region automata is bounded by $O(|Q| \times n^m \times m!) = O(2^{log\ (|Q_1|\times\ n^m \times\ m!)}) = O(2^{log\ |Q_1|+mlog\ n+log\ m!})$. Knowing that, $m! \leq m^m$, then we have $O(2^{log\ |Q_1|+mlog\ n+mlog\ m})$ which is equal to $O(2^{log\ |Q_1|+m(log\ n+log\ m)})$ We conclude that the size of the region automata associated to S_1 and S_2 are respectively $O(2^{2.(log\ |Q_1|+m(log\ n+log\ m))})$ and $O(2^{2.(log\ |Q_2|+m(log\ n+log\ m))})$.

□

The proof of exptime-hardness is achieved by a reduction from the problem of existence of an infinite execution of an alternating Turing machine M working on space polynomially bounded by an input word w of size n. This problem is know to be EXPTIME-hard [CKS81, MW07]. We provide below a reduction from this problem to a test of simulation between two DB-less Colombo services $S_{spoiler}$ and $S_{duplicator}$.
We first recall the definition of an alternating Turing machine M, then we give the intuition of the reduction. After that, we explain the construction of the services $S_{duplicator}$ and $S_{spoiler}$. Finally, we give the construction of the test of simulation $S_{spoiler} \preceq S_{duplicator}$ and prove that M has an infinite execution on w iff $S_{spoiler} \preceq S_{duplicator}$.

Alternating Turing machine M An alternating Turing machine M [CKS81] is a tuple $(\mathbb{Q}, q_0, \Gamma, \delta, mode)$ where :

– $\mathbb{Q}$ is the set of control states.
– q_0 is the initial state.
– Γ is the set of tape symbols.
– $mode : \mathbb{Q} \longrightarrow \{\forall, \exists,$ accept , reject $\}$ is the labelling function of control state.
– $\delta : \mathbb{Q}$ x $\Gamma \longrightarrow \mathcal{P}(\mathbb{Q}$ x Γ x $\{L, R\})$.

A configuration C of M is of the form $y_1, ..., qy_j, ..., y_n$, where q is a state of the machine, and the head points actually on the j'th letter on the tape (i.e., y_i are the letters of the word on the tape). A transition $qa \longrightarrow bRq^{'}$ is applicable from a configuration C if the letter pointed by the head is equal to a (y_j=a), then the successor C' of C is equal to $y^{'}_1, ...y^{'}_j, q^{'} y^{'}_{j+1}, ..., y^{'}_n$ s.t y_k= $y^{'}_k$ for k $\in$ [1,n] and k $\neq$ j and $y^{'}_j = b$. We note this step $C \overset{qa/bRq^{'}}{\longrightarrow} C'$ or $(y_1, ..., qy_j, ..., y_n) \overset{qa/bRq^{'}}{\longrightarrow} (y^{'}_1, ...y^{'}_j, q^{'} y^{'}_{j+1}, ..., y^{'}_n)$. The machine M starts on $C_0 = qy_1, ..., y_n$, where y_i=w_i, the i'th letter of the input word w.

The definition of acceptance of an alternating Turing machine is recursive :

– If the configuration C is in an accepting control state q, then C is accepting.
– If the configuration C is in an rejecting control state q, then C is rejecting.
– If the configuration C is in a universal control state q, then C is accepting if all the configurations reachable from C in one step are accepting and rejecting if some configurations reachable from C in one step are rejecting.
– If the configuration C is in an existential control state q, then C is accepting if some configurations reachable in one step are accepting and rejecting when all configurations reachable in one step are rejecting (the case of classical non-deterministic Turing machine correspond to an alternating machine where all states are existential).

M is said to accept an input word w if the initial configuration of M is accepting, and to reject w if the initial configuration is rejecting. A configuration reachable in one step from configuration C is called a *successor* of C and the set of *successors* of C is denoted $successors(C)$.

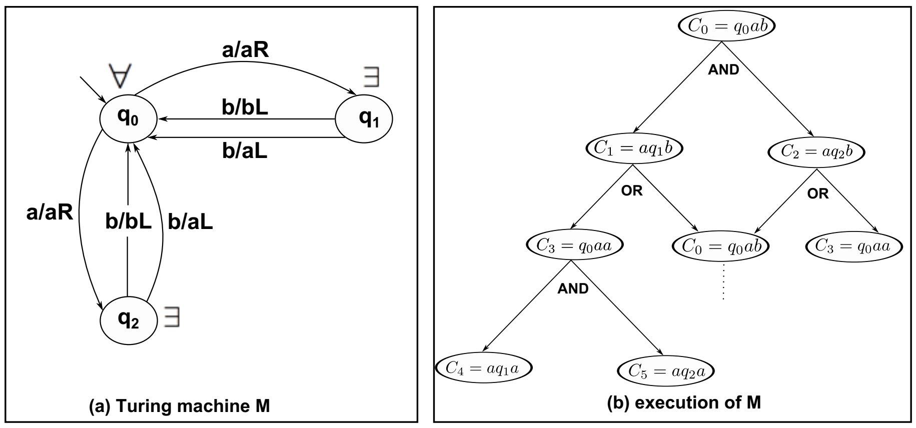


Figure 2.14 – Alternating Turing machine M.

We consider the problem of the existence of an infinite execution of an alternating Turing machine M on an input word $w = y_1, ..., y_n$, where y_i's are letters from Γ. That is given a word w as input, M can make choices of existential transitions such that whatever the transitions chosen by universal states the machine continues the execution. Assume that, the rejecting states are states without outgoing transitions. The machine works on a space bounded by the size n of the input word w. Hence, if the head points on y_1 the machine is not allowed to move to the left (i.e., execute a transition labelled with L), and if the head points on y_n the machine is not allowed to move to the right (i.e., execute a transition labelled with R).

Example 19. Figure 2.14(a) depicts an alternating Turing machine M, where the initial state q_0 is universal, and q_1, q_2 are existential states. Suppose w=ab, then starting from the initial configuration $C_0 = (q_0ab)$, the machine has two successors : $C_0 \overset{qa/aRq_1}{\longrightarrow} C_1 = (aq_1b)$, and $C_0 \overset{qa/aRq_2}{\longrightarrow} C_2 = (aq_2b)$. C_1 has two successors, one reads b and replaces it by itself leading to the configuration C_0, the other replaces b by a, hence the machine reaches C_3. The two successors of C_3 are blocking. Starting from the initial state, for all choices of the universal state, there is a successor of the existential state s.t the machine continue the execution. So the machine M has an infinite execution on the input word ab.

The idea of the proof is that, starting from the machine M, we construct the service $S_{spoiler}$ which is able to execute any action of the machine infinitely often. The service $S_{duplicator}$ encodes exactly the execution of M on the word w. During the execution of M, if M is in a configuration without successors then the corresponding configuration of the service $S_{duplicator}$ does not have successors. For each configuration of C with successors, the corresponding configuration of the service $S_{duplicator}$ has also successors. Now, because $S_{spoiler}$ can execute any action infinitely often, $S_{duplicator}$ must contain also an infinite execution to simulate $S_{spoiler}$ and this is possible if and only if the machine M has an infinite execution. Additional transitions will be added to $S_{duplicator}$. Those transitions will be used to prevent the $S_{spoiler}$ of cheating, because this service can execute any action at any time. Then, at a given step of execution of $S_{duplicator}$, if $S_{duplicator}$ can execute a transition representing a transition of M, then $S_{spoiler}$ must follow it. If $S_{spoiler}$ tries to execute a transition not allowed by the machine M, then $S_{spoiler}$ looses the simulation game. $S_{spoiler}$ wins the simulation game if and only if $S_{duplicator}$ blocks during an execution, which means the machine M blocks and does not contains an infinite execution on w.

Given a machine M bounded by the size n of the input word w. $S_{duplicator}$ will use n variables to simulate the n cells. The position of the head is encoded in a variable called *head*. A state q of M is encoded as a state l_q in $S_{duplicator}$. A transition $q \overset{a/bR}{\longrightarrow} q'$ of M, is encoded in a transition $(l_q, x_i = a \wedge head = i, qabq^{'}R_i(\emptyset; x_i, head), l_{q'})$(i.e., if the actual value of the i'th variable is equal to a and the variable *head* contains i, then executes the atomic process $qabq^{'}R_i(\emptyset; x_i, head)$ which increments the variable *head* and modifies the value of x_i to b). Because during the execution of the machine we do not control which cell is read, we create $n-1$ transitions in $S_{duplicator}$ from l_q to $l_{q'}$ (the head cannot move to the right of the last variable this why i is ranged in [1,n-1]). If the direction is L, the same construction is made, but now the atomic process decrements the head and i is ranged in [2,n].

Example 20. Figure 2.15(a) depicts the transition in $S_{duplicator}$ which encodes the transition $q_0 \overset{a/aR}{\longrightarrow} q_1$ of the machine M in example 19. The atomic process $qabq^{'}R_1(\emptyset; x_1, head)$ is depicted in Figure 2.15(b). There is only one transition, because in this example the size of w=2 hence it is not possible to move to the right from the second cell.

$S_{duplicator}$ starts by initializing the variables $x_1, ..., x_n$ to the input word w and assigning 1 to the variable *head*.

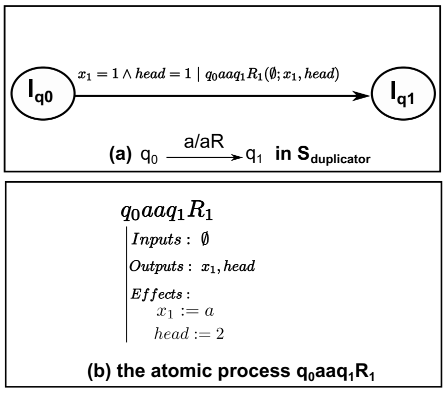

$x_1 = 1 \wedge head = 1 \mid q_0aaq_1R_1(\emptyset; x_1, head)$
l_{q0}
l_{q1}
(a) $q_0 \xrightarrow{a/aR} q_1$ in $S_{duplicator}$
$q_0aaq_1R_1$
Inputs : $\emptyset$
Outputs : $x_1, head$
Effects :
$x_1 := a$
$head := 2$
(b) the atomic process $q_0aaq_1R_1$

Figure 2.15 – A transition in $S_{duplicator}$ corresponding to a transition of M.

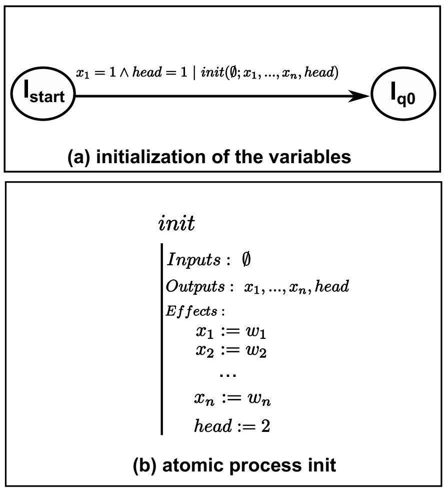

$x_1 = 1 \wedge head = 1 \mid init(\emptyset; x_1, ..., x_n, head)$
l_{start}
l_{q0}
(a) initialization of the variables
init
Inputs : $\emptyset$
Outputs : $x_1, ..., x_n, head$
Effects :
$x_1 := w_1$
$x_2 := w_2$
...
$x_n := w_n$
$head := 2$
(b) atomic process init

Figure 2.16 – initialization of variables in $S_{duplicator}$.

Example 21. Figure 2.16 depicts the initialization of the service $S_{duplicator}$ corresponding to the machine M of the example 19, where x_1 :=a, x_2 := b and *head* :=1.

Before giving the construction of $S_{duplicator}$, we need to introduce some notations :
- $\mathcal{P}$ is the set of all atomic processes used to encode actions of the machine M, it contains the following sets :
  - $\{qabq'R_i(\emptyset; x_i, head) \mid q \overset{a/bR}{\longrightarrow} q' \ \ in \ \ M \ \ and \ \ i \in [1, n-1]\}$
    For each transition of the machine labelled with a move to the right, we create n-1 atomic processes to encode it.
  - $\{qabq'L_i(\emptyset; x_i, head) \mid q \overset{a/bL}{\longrightarrow} q' \ \ in \ \ M \ \ and \ \ i \in [2, n]\}$
    For each transition of the machine labelled with a move to the left, we create n-1 atomic processes to encode it.

  The atomic process $qabq'R_i(\emptyset; x_i, head)$ has no condition, it assigns to x_i the value b and increments the head. The atomic process $qabq'L_i(\emptyset; x_i, head)$ has no conditions, it assigns to x_i the value b and decrements the head.
- g_i^a is a condition of the form x_i =a $\wedge$ *head*= i. It will be used as guard on transitions of $S_{duplicator}$.

The incrementation is not allowed in the definition of the Colombo model. When defining the effects of the atomic process, we write the result of the sum rather than the operation of incrementation. For example, in the atomic process $qabq'R_1(\emptyset; x_1, head)$, which represents the transition $q \xrightarrow{a/bR} q'$ depicted at figure 2.15, the variable head receives 2 instead of 1+1.

In the sequel, we make $\mathcal{P}^i_{qa}$ the subset of $\mathcal{P}$ restricted to the atomic processes modifying the variable x_i and representing only the transitions of the machine M from the state q and reading the letter a.

Construction of S$_{\textbf{duplicator}}$: Each configuration of M that is reachable during the execution of the machine on w corresponds to an *id* of $S_{duplicator}$. During an execution of M, the actual configuration of the machine M has a successor if the corresponding *id* of an execution of $S_{duplicator}$ has a successor. If the execution of M blocks on a configuration, then the service also blocks on the corresponding *id*. We will use a set of additional transitions to force the spoiler to follow the actions chosen by the duplicator during an execution.
Let $GA(S_{duplicator}$ be the *guarded automaton* of the service $S_{duplicator}$ where $GA(S_{duplicator}) = \langle Q_{duplicator}, \delta_{duplicator}, l^{'}_{start}, LStore(S_{duplicator})\rangle$ and :

- the set of states of $S_{duplicator}$ are :
  - $\{l_q \mid q \in \mathbb{Q}\}$.
  - the initial state of M is a final state in $S_{duplicator}$.
  - a state l_{copy}, which is also a final state.
  - $\{choice_{qbdq'} \mid q \xrightarrow{a/bd} q^{'}$ *and* q *an exitential state and* $d = R/L\}$
  - $l^{'}_{start}$ is the initial state.
- $Lstore(S_{duplicator}) = \{x_1, ..., x_n\} \cup \{head\}$, where $n = |w|$
- $\delta_{duplicator}$ is made of the following sets of transitions :
  - ($l^{'}_{start}$, true, $init(\emptyset; head, x_1, ..., x_n)$, l_q), where q the initial state of M.
  - (l_{copy}, true, a, l_{copy}), where a $\in \mathcal{P} \cup \{!m()\}$ and $\mathcal{P}$ is the set of all atomic processes in $S_{spoiler}$.
  - For each transition $q \xrightarrow{a/bd} q^{'}$ in M, where q is a universal state :
    - if $d = R$ then :
      - $\{(l_q, g^a_i, qabq^{'}R_i(\emptyset; x_i, head), l_{q^{'}}) \mid i \in [1, n-1]\}$.
      - $\{(l_q, \mathsf{true}, !m(), l_{copy})\}$.
      - $\{(l_q, g^a_i, \mathcal{P} \setminus \mathcal{P}^i_{qa}, l_{copy}) \mid i \in [1, n-1]\}$.
    - if $d = L$ then :
      - $\{(l_q, g^a_i, qabq^{'}L_i(\emptyset; x_i, head), l_{q^{'}}) \mid i \in [2, n]\}$.
      - $\{(l_q, \mathsf{true}, !m(), l_{copy})\}$.
      - $\{(l_q, g^a_i, \mathcal{P} \setminus \mathcal{P}^i_{qa}, l_{copy}) \mid i \in [2, n]\}$.
  - For each transition $q \xrightarrow{a/bd} q^{'}$ in M, where q is a existential state :
    - if $d = R$ then :
      - $\{(l_q, \mathsf{true}, !m(), choice_{qbRq'})\}$
      - $\{(l_q, \mathsf{true}, \mathcal{P} \setminus \{!m()\}, l_{copy})\}$.
      - $\{(choice_{qbRq'}, g^a_i, qabq^{'}R_i(\emptyset; x_i, head), l_{q^{'}}) \mid i \in [1, n-1]\}$.
      - $\{(choice_{qbRq'}, \mathsf{true}, \mathcal{P} \setminus \{qabq^{'}R_i(\emptyset; x_i, head)\}, l_{copy}) \mid i \in [1, n-1]\}$.
    - if $d = L$ then :
      - $\{(l_q, \mathsf{true}, !m(), choice_{qbLq'})\}$
      - $\{(l_q, \mathsf{true}, \mathcal{P} \setminus \{!m()\}, l_{copy})\}$.
      - $\{(choice_{qbLq'}, g^a_i, qabq^{'}L_i(\emptyset; x_i, head), l_{q^{'}}) \mid i \in [2, n]\}$.
      - $\{(choice_{qbLq'}, \mathsf{true}, \mathcal{P} \setminus \{qabq^{'}L(\emptyset; x_i, head)\}, l_{copy}) \mid i \in [2, n]\}$.

Note that, if the machine reads or writes the special blank character B during a transition, then we replace the constants a,b by the special symbol ω, in the construction of the corresponding transition.

$S_{duplicator}$ starts by initializing the variables representing the cells with the input word. If M has a transition $q \overset{a/bR}{\longrightarrow} q^{'}$ and q is a universal state, then the service contains $n-1$ transitions from l_q to $l_{q'}$ labelled with condition/action : if x_i=a and the head points on i then we can execute the atomic process which modifies x_i to b and increments the head. So, $S_{duplicator}$ can only execute the atomic process representing the transition $q \overset{a/bR}{\longrightarrow} q^{'}$ if the actual value of x_i=a and the head points on i. Note that, for any actual valuation of variables, there is only one transition from the "n-1" transitions which can be executed. This is due to the guards where several x_i can verify the condition but the head points only to one cell.
If q is an existential state, then $S_{duplicator}$ sends a message m before executing the atomic process. The state l_{copy} contains a set of self loop labelled with all atomic processes $\mathcal{P}$ and $!m()$ (if $S_{duplicator}$ reaches this state, it wins the simulation). All transitions which reach the state l_{copy} are used to prevent $S_{spoiler}$ from cheating during the test of simulation.

The next lemma asserts that each configuration of M on the input word w has a corresponding configuration in the extended state machine of $S_{duplicator}$. The proof is obtained by induction (details are given in appendix A).

Lemma 6. *Each configuration C of the execution of an alternating Turing machine M on an input w has a corresponding configuration in the extended state machine of $S_{duplicator}$.*

Example 22. The Figure 2.17 depicts the part of service $S_{duplicator}$ corresponding to the transition $q_0 \overset{a/aR}{\longrightarrow} q_1$ where q_0 is universal, and the two transitions $q_1 \overset{b/bL}{\longrightarrow} q_0$ and $q_1 \overset{b/aL}{\longrightarrow} q_0$ where q_1 is an existential state of the machine M of example 19.

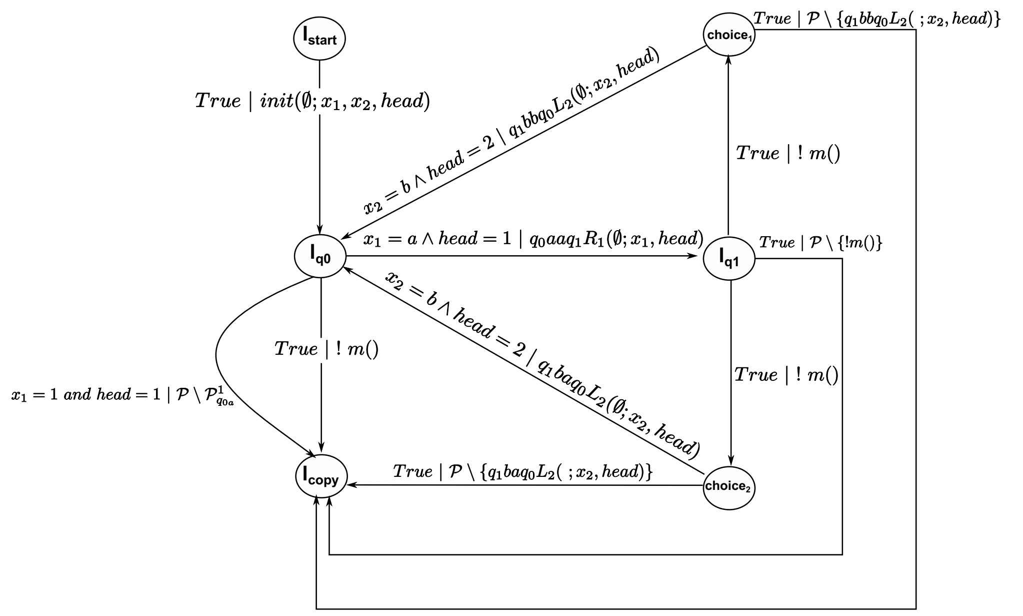

l_{start}
$True \mid init(\emptyset; x_1, x_2, head)$
$choice_1$
$True \mid \mathcal{P} \setminus \{q_1bbq_0L_2(\,; x_2, head)\}$
$x_2 = b \wedge head = 2 \mid q_1bbq_0L_2(\emptyset; x_2, head)$
$True \mid !\, m()$
l_{q0}
$x_1 = a \wedge head = 1 \mid q_0aaq_1R_1(\emptyset; x_1, head)$
l_{q1}
$True \mid \mathcal{P} \setminus \{!m()\}$
$x_2 = b \wedge head = 2 \mid q_1baq_0L_2(\emptyset; x_2, head)$
$True \mid !\, m()$
$True \mid !\, m()$
$x_1 = 1 \; and \; head = 1 \mid \mathcal{P} \setminus \mathcal{P}^1_{q_{0a}}$
l_{copy}
$True \mid \mathcal{P} \setminus \{q_1baq_0L_2(\,; x_2, head)\}$
$choice_2$

Figure 2.17 – A part of the service $S_{duplicator}$.

Construction of $\mathbf{S_{spoiler}}$: The spoiler uses n variables $z_1, ..., z_n$ and the variable *head*. It starts as $S_{duplicator}$ by initializing the variables to the letters of the word w and the head to 1. $S_{spoiler}$ encodes all transitions that the machine M can do. If M has a transition $q \overset{a/bR}{\longrightarrow} q^{'}$ and q is a universal state, then the service has (n-1) self loop on state q_{univ} labelled with $qabq^{'}R_i(\emptyset; z_i, head)$. If q is an existential state, first the service goes to an intermediate state q_{exist} by sending the message $m()$. Then, the service has $n-1$ transitions labelled with atomic processes $qabq^{'}R_i(\emptyset; z_i, head)$ from q_{exist} to q_{univ}. The transitions are not guarded. Hence, $S_{spoiler}$ can choose to execute any actions (infinitely often) without constraints on actual values of variables.

The guarded automata of $S_{spoiler}$ is given below. $GA(S_{spoiler}) = \langle Q_{spoiler}, \delta_{spoiler}, q_{start}, q_{univ}, LStore(S_{spoiler})\rangle$ where :
- $Q_{spoiler}$= $\{q_{start}, q_{univ}, q_{exist}\}$, where q_{start} is the initial state and q_{univ} the final state.
- $Lstore(S_{spoiler}) = \{z_1, ..., z_n\} \cup \{head\}$.
- $\delta_{spoiler}$ is made of the following sets of transitions :
  - (q_{start}, true, $init(\emptyset; head, z_1, ..., z_n)$, q_{univ}),
    where *init* initializes *head* to 1 and z_i to w_i (the i'th) letter of w.
  - (q_{univ}, true, $!m()$, q_{exist}).
  - For each transition $q \overset{a/bd}{\longrightarrow} q^{'}$ in M, where q is a universal state :
    - if $d = R$ then :
      - $\{(q_{univ}, \mathsf{true}, qabq^{'}R_i(\emptyset; z_i, head), q_{univ}) \mid i \in [1, n-1]\}$,
        where z_i receives b and *head* receives $i+1$ (moves to the right).
    - if $d = L$ then :
      - $\{(q_{univ}, \mathsf{true}, qabq^{'}L_i(\emptyset; z_i, head), q_{univ}) \mid i \in [2, n]\}$,
        where z_i receives b and *head* receives $i-1$ (moves to the left).
  - For each transition $q \overset{a/bd}{\longrightarrow} q^{'}$ where q is a existential state :
    - if $d = R$ then :
      - $\{(q_{exist}, \mathsf{true}, qabq^{'}R_i(\emptyset; z_i, head), q_{univ}) \mid i \in [1, n-1]\}$.
    - if $d = L$ then :
      - $\{(q_{exist}, \mathsf{true}, qab^{'}L_i(\emptyset; z_i, head), q_{univ}) \mid i \in [2, n]\}$.

Example 23. The Figure 2.18 depicts the part of the service $S_{spoiler}$ corresponding to :
- the transition $q_0 \overset{a/aR}{\longrightarrow} q_1$ where q_0 is universal and
- the two transitions $q_1 \overset{b/bL}{\longrightarrow} q_0$ and $q_1 \overset{b/aL}{\longrightarrow} q_0$ where q_1 is an existential state of the machine M in example 19.

Given an alternating Turing machine M an input word w, we call the services $S_{spoiler}$ and $S_{duplicator}$ constructed as explained previously, respectively the *Spoiler* and the *Duplicator* associated to M and w. The next lemma shows the connection between the existence of infinite execution of the machine M over the word w and the test of simulation between $S_{spoiler}$ and $S_{duplicator}$. The proof is given in appendix A.

Lemma 7. *Let M be an alternating Turing machine working in space bounded by the size of an input word w, and let $S_{spoiler}$ and $S_{duplicator}$ the services associated to M and w. Then, M has an infinite computation on w iff $S_{spoiler} \preceq S_{duplicator}$.*

From lemma 7 and knowing that the problem of existence of an infinite execution of an alternating Turing machine work on a space polynomially bounded by the size of the input is EXPTIME-hard [CKS81] we can derive the following lemma :

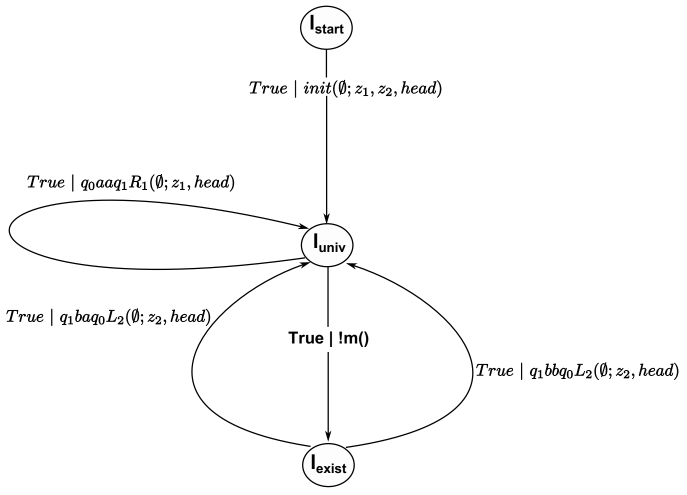

I_{start}
$True \mid init(\emptyset; z_1, z_2, head)$
$True \mid q_0aaq_1R_1(\emptyset; z_1, head)$
I_{univ}
$True \mid q_1baq_0L_2(\emptyset; z_2, head)$
True | !m()
$True \mid q_1bbq_0L_2(\emptyset; z_2, head)$
I_{exist}

FIGURE 2.18 – part of $S_{spoiler}$.

Lemma 8. *Given two DB-less Colombo services S, $S^{'}$, checking whether $S \preceq S^{'}$ is* EXPTIME-*hard.*

Hence, the following theorem can now be claimed from proposition 1 and lemma 8

Theorem 5. *Given two DB-less Colombo services S, $S^{'}$, checking whether $S \preceq S^{'}$ is* EXPTIME-*complete.*

2.5 Decidability of simulation in $Colombo^{bound}$

We study in this section the simulation problem in the setting of a Colombo model with a *bounded* global database (i.e., the size of the instance over $\mathcal{W}$ is at most equal to a constant k). Given two services S and $S^{'}$, S is *k-bounded simulated* by $S^{'}$ means that $S^{'}$ is able to reproduce the behavior of S on all executions where the size of the database is at most equal to k. We will prove that the simulation is decidable in this setting by providing a reduction to a test of simulation between two DB-less $Colombo^{DB=\emptyset}$ services. This is done by encoding the bounded database using a finite set of variables. First we start by giving the definition of *k-bounded extended state machines*, which is used to capture the notion of k-bounded simulation. Then we give the construction of the DB-less service and prove the equivalence of the two tests.

2.5.1 k-bounded extended state machine $E^k(S)$ and k-bounded simulation

Let k be an integer. We call a database instance I k-bounded if $|I| \leqslant k$. The k-bounded extended state machine $E^k(S)$ of a Colombo service S is the extended state machine $E(S)$ of S restricted to configurations having k-bounded instances.

Definition 9. *Let S be a Colombo service and $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ the associated extended state machine, then $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}_0^k, \mathbb{F}^k, \Delta^k)$ is the k-bounded extended state machine of S where :*
– $\mathbb{Q}^k = \{(l, \mathcal{I}, \alpha) \mid (l, \mathcal{I}, \alpha) \in \mathbb{Q}$ *and* $|\mathcal{I}| \leq k\}$.

The k-bounded extended state machine of S is the part of $E(S)$ where all configurations contain only k-bounded databases. Like $E(S)$, a run σ of $E^k(S)$ is a finite sequence

$\sigma = id_0 \xrightarrow{\mu_0} id_1 \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_{n-1}} id_n$ where id_0 is an initial configuration and id_n a final configuration but $|\mathcal{I}_i| \leq k$ for $i \in [0,n]$ where id_i=$((l_i, \mathcal{I}_i, \alpha_i))$. Due to infinite number of k-bounded initial databases, all runs of $E^k(S)$ form a forest.

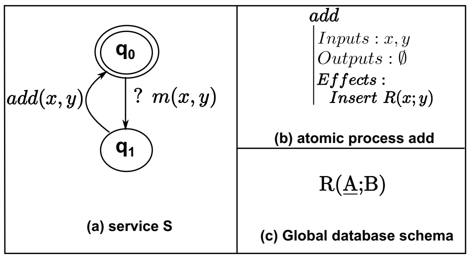


Figure 2.19 – A Colombo service S.

Example 24. Figure 2.19 depicts a simple Colombo service S which receives two variables x and y. The service S uses the atomic process *add* to insert the tuple (x,y) in the database R. The service can make an infinite loop during an execution and inserts an unbounded number of tuples in R.

Figure 2.20 depicts two execution paths of $E(S)$. The execution path depicted at figure 2.20(a) starts with an instance of R which contains one tuple $\langle 6, h\rangle$ then inserts the tuple $\langle 7, 2\rangle$. This execution path of $E(S)$ is also an execution path of the 2-bounded extended state machine $E^2(S)$. The second execution path (figure 2.20(b)) starts with an instance containing only the tuple $\langle 8, 1\rangle$, then inserts the tuple $\langle 9, 3\rangle$, and finally inserts the tuple $\langle 1, 2\rangle$. The second path does not belong to $E^2(S)$ because the database of the last configuration does not satisfies the condition $|R| \leq 2$.

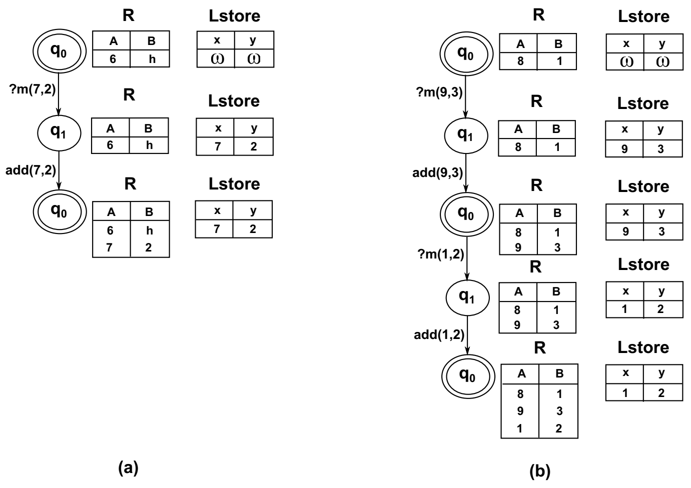


Figure 2.20 – example of an execution path in $E^2(S)$ and an execution path not in $E^2(S)$.

We define now the notion of k-bounded simulation, denoted $\preceq_k$.

Definition 10. *A Colombo service S is k-bounded simulated by a Colombo service S', noted $S \preceq_k S'$, iff $E^k(S) \preceq E^k(S')$.*

It is worth noting that if $S \preceq S^{'}$ then $S \preceq_k S^{'}$ but the converse is not true.

2.5.2 Mapping bounded Colombo services into Colombo DB-less services

We will prove that the decidability of k-bounded simulation by proving that, for any two colombo services S and $S^{'}$, testing k-bounded simulation $S \preceq_k S^{'}$ is equivalent to testing $\mathcal{M}(S) \preceq \mathcal{M}(S^{'})$, where $\mathcal{M}(S)$ and $\mathcal{M}(S^{'})$ are two DB-less services.

The main idea is to use a set of variables to encode k-bounded database instances. Assume that $\mathcal{W} = \{R\}$ and $arity(R)$= n, then the maximum number of values that can be stored in a k-bounded database is $n * k$. Hence, all k-bounded instances can be encoded with $n * k$ variables. We will use the following example to explain the transformation from a bounded Colombo service into a DB-less service.

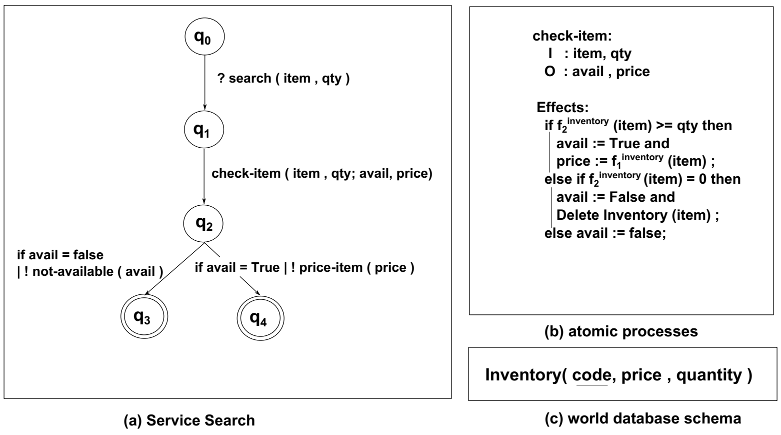

q0
? search (item , qty)
q1
check-item (item , qty; avail, price)
q2
if avail = false
| ! not-available (avail)
if avail = True | ! price-item (price)
q3
q4
(a) Service Search
check-item:
I : item, qty
O : avail , price
Effects:
if f2inventory (item) >= qty then
avail := True and
price := f1inventory (item) ;
else if f2inventory (item) = 0 then
avail := False and
Delete Inventory (item) ;
else avail := false;
(b) atomic processes
Inventory(code, price , quantity)
(c) world database schema

Figure 2.21 – Colombo services *Search*.

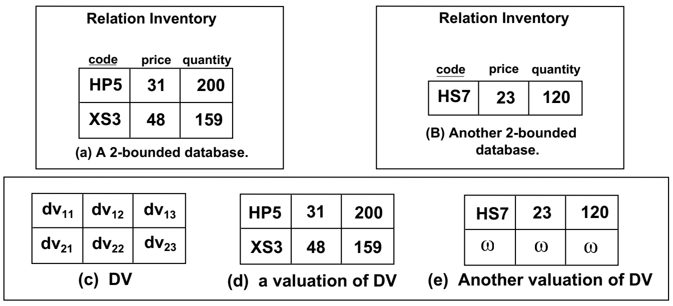
Example 25. Figure 2.21(a) depicts the Colombo service *Search*. *Search* makes use of the atomic process *check-item* depicted at figure 2.21(b) in order to retrieve a product for a client in the global relation *Inventory* (figure 2.21(c)) and sends the price of the product if the quantity requested is available. If the quantity of the product is equal to zero, then the product is deleted from the inventory.

Database variables DV

As said earlier, the number of variables used to encode the database depends on the arity of the database schema and the bound k. To simplify the presentation, and w.l.o.g, from now we suppose that $\mathcal{W}$ contains only one relation $R(A_1; B_1, \ldots, B_m)$. The set of variables used to encode the bounded database instances are called *database variables* (DV) and denoted dv_{ij}, where i and j are integers.

Definition 11. *Let $R(A_1; B_1, \ldots, B_m)$ be a world database schema and k be a constant. Then DV=$\{dv_{ij} \mid i \in [1, k]$ and $j \in [1, m+1]\}$.*

Note that the variables dv_{i1}, here often called *key variables*, represent the possible values of the keys (the attribute A_1). Figure 2.22 depicts two instances of the relation schema *Inventory* and the corresponding set DV. The elements of the tuple $\langle HP5, 31, 200\rangle$ of the first instance (figure 2.22(a)) are stored respectively in the variables dv_{11}, dv_{12}, dv_{13} and those of the tuple $\langle XS3, 48, 159\rangle$ are stored in dv_{21}, dv_{22}, dv_{23} (the valuation of DV is depicted at figure 2.22(d)). The tuple $\langle HS7, 23, 120\rangle$ of the second instance (figure 2.22(b)) is stored in dv_{11}, dv_{12}, dv_{13} (the second valuation of DV depicted at figure 2.22(e)).



Relation Inventory

<u>code</u>	price	quantity
HP5	31	200
XS3	48	159

(a) A 2-bounded database.

Relation Inventory

<u>code</u>	price	quantity
HS7	23	120

(B) Another 2-bounded database.

dv_{11}	dv_{12}	dv_{13}
dv_{21}	dv_{22}	dv_{23}

(c) DV

HP5	31	200
XS3	48	159

(d) a valuation of DV

HS7	23	120
ω	ω	ω

(e) Another valuation of DV

Figure 2.22 – Instances of the database *Inventory* and the corresponding sets of variables *DV*.

Initialization of DV

The execution of a Colombo service starts with a null valuation (ω) for all variables in the *Lstore*. An additional state and a transition are added in order to enable the initialization of the database. Let S be a Colombo service, the corresponding DB-less service will start with a transition labelled with $?database(dv_{1j}, ...dv_{1m+1}, ..., dv_{km+1})$.
It should be noted that, during the execution of a Colombo service, only database instances satisfying the key constraints ca be used. w.l.o.g, we assume that, the key constraints are always satisfied in our case. Indeed, it is possible to add to DB-less services a test to check that the values of the database variables correspond to a database instance that satisfies the key constraints.

Atomic process transformation

We recall that, a Colombo service accesses to the database only using *atomic processes*, either to retrieve information or modify it. An atomic process is a triplet $p = (I, O, CE)$ where : I=$u_1, ..., u_n$ are the input variables and O=$v_1, ..., v_m$ are the output variables and $CE = \{(\theta, es, ev)\}$ is a set of conditional effects with θ a condition, ev the set of modifications of output variables and es (state effects) are the modifications of the database instance. An atomic process accesses to the values of a database using the access function f_j^R through the condition θ or in ev by assigning a value of the database to an output variable. The atomic process can also modify the database using the state effects es(*Insert,Delete,Modify*).
In the following, We will explain how to transform an atomic process $p = (I, O, CE)$ acting on a database R to an atomic process $p_v = (I_v, O_v, CE_v)$ acting on the set of variables DV. We start by the access function f_j^R then the outputs effects ev and finally we explain how to transform the state effects.

1. Encoding f_j^R
   Let "$f_j^R(t)\ \ op\ \ t'$" be a condition in θ. The corresponding p_v will contain k conditions θ_{vi} with $i \in$ [1,k] of the form :
   – θ_{vi}= $\{\ \ (dv_{i1} = t)\ \ \wedge\ \ (dv_{ij+1}\ \ op\ \ t^{'})\ \ \}$
   $f_j^R(t)$ returns the value of the j+1'th element of the tuple having the key equal to t, and then compares this value with $t^{'}$. To encode this action, we need to check if the key variable dv_{i1} ($i \in$ [1,k]) is equal to t, then compare dv_{ij+1} to t' according to *op*. This test is repeated k times.

For example, the atomic process *check-item* depicted in figure 2.21(b), contains a condition $f_2^{Inventory}(item) \geq qty$. This test will be transformed into θ_{v1}, θ_{v2} where :
– $\theta_{v1} : (dv_{11} = item) \wedge \; (dv_{13} \geq qty)$.
– $\theta_{v2} : (dv_{21} = item) \wedge \; (dv_{23} \geq qty)$.

2. Encoding *ev*
Let "$v_l := \; f_j^R(t)$". Here the output variable v_l receives the value of $f_j^R(t)$. As for the condition θ, we need first to retrieve the key variable dv_{i1} equal to t and then assign dv_{ij+1} to v_l. The test and the assignment is made k times. The corresponding atomic process will contain a set of pairs (θ_i, ev_i) where :
– $\theta_i : dv_{i1} = t$ and
– $ev_i : v_l := dv_{ij+1}$
Continuing with the atomic process *check-item*. The assignment $price := f_1^{Inventory}(item)$, will be mapped into two pairs (θ_i, ev_i) :
– $(\theta_1, ev_1) : (dv_{11} = item, price := dv_{12})$.
– $(\theta_2, ev_2) : (dv_{21} = item, price := dv_{22})$.

3. Encoding *es*

– *insert* $R(t_1, s_1, \ldots, s_m)$. The insertion is encoded by retrieving a variable $dv_{i1} = \omega$, then assigning respectively $t_1, s_1, \ldots, s_m$ to dv_{i1}, $dv_{i2}, \ldots, dv_{m+1}$. p_v will contain k pairs of the form (θ_i, ev_i) where
  – $\theta_i = dv_{i1} = \omega$ and
  – $ev_i = \{ \; dv_{ij} := s_l \; \mid \; j \in [2, m+1] \; \; and \; \; l \in [1, m] \; \} \cup \{dv_{i1} := t_1 \; \}$.
– *delete* $R(t_1)$. The deletion is made by retrieving the key variable dv_{i1} equal to t_1 then assigning to the variable dv_{i1} the value ω. The new atomic process p_v will contain k pairs of the form (θ_i, ev_i) :
  – $\theta_i = dv_{i1} = t_1$ and
  – $ev_i = dv_{i1} := \omega$.
– *modify* $R(t_1, r_1, \ldots, r_m)$. To simulate the modification we need to find the key variable dv_{i1} equal to t_1, then assign to dv_{ij} the corresponding r_l if r_l is different from "_" . As for the previous cases, we add k pairs of the form (θ_i, ev_i) :
  – $\theta_i = dv_{i1} = t_1$ and
  – $ev_i = \{ \; dv_{ij} := r_l \; \mid \; r_l \neq \text{"\_"} \; \; and \; \; l = j + 1 \; \}$.

For example, the atomic process *check-item* deletes a product if its quantity is equal to zero (i.e., with the state effect $Delete \;\; Inventory(item)$). This action on database will be transformed into two pairs (θ_i, ev_i) :
– $(\theta_1, ev_1) : (dv_{11} = 0, \; v_{11} := \omega)$.
– $(\theta_2, ev_2) : (dv_{21} = 0, \; dv_{21} := \omega)$.

Let $p = (I, O, CE)$ be an atomic process updating the database $R(A_1; B_1, \ldots, B_m)$. then $p_v = (I_v, O_v, CE_v)$ is constructed as follows :
– $I_v = I \cup DV$.
– $O_v = O \cup DV$.
– The set CE_v is obtained by applying the rules defined before.
The Figure 2.23(b) depicts the atomic process *check-item*$_v$.

Now we will give the definition of the mapping from a bounded Colombo service S to a corresponding DB-less service $\mathcal{M}(S)$.

Definition 12. *Let* $GA(S) = \langle Q, \delta, l_0, F, LStore(S) \rangle$ *be a guarded automata of a service* S *and* k *a constant. Then* $GA(\mathcal{M}(S)) = \langle Q_{\mathcal{M}(S)}, \delta_{\mathcal{M}(S)}, l_{init}, F_{\mathcal{M}(S)}, LStore(\mathcal{M}(S)) \rangle$ *where :*
– $Q_{\mathcal{M}(S)} = Q \cup \{l_{init}\}$. $\mathcal{M}(S)$ *contains all states of* S *and an additional state* l_{init}

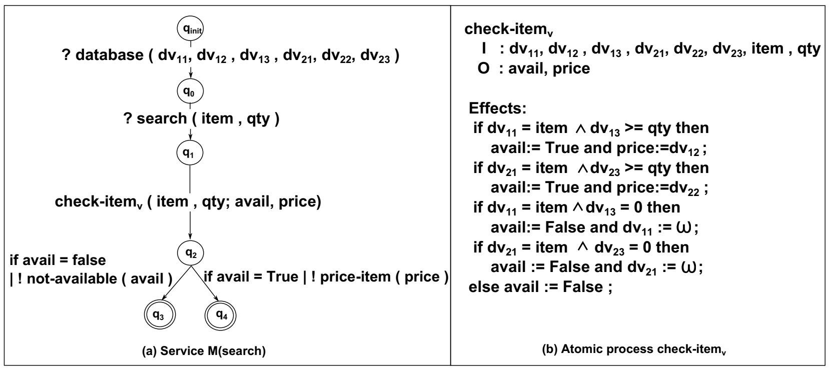

q_init
? database (dv11, dv12 , dv13 , dv21, dv22, dv23)
q0
? search (item , qty)
q1
check-itemv (item , qty; avail, price)
q2
if avail = false
| ! not-available (avail)
if avail = True | ! price-item (price)
q3
q4
(a) Service M(search)
check-itemv
I : dv11, dv12 , dv13 , dv21, dv22, dv23, item , qty
O : avail, price
Effects:
if dv11 = item ∧ dv13 >= qty then
avail:= True and price:=dv12 ;
if dv21 = item ∧dv23 >= qty then
avail:= True and price:=dv22 ;
if dv11 = item ∧dv13 = 0 then
avail:= False and dv11 := ω;
if dv21 = item ∧ dv23 = 0 then
avail := False and dv21 := ω;
else avail := False ;
(b) Atomic process check-itemv

Figure 2.23 – $\mathcal{M}(Search)$.

- *l_{init} is the initial state of $\mathcal{M}(S)$.*
- *$F_{\mathcal{M}(S)} = F$ is the set of final states.*
- *$LStore(\mathcal{M}(S))$=$LStore(S) \cup DV$.*
- *$\delta_{\mathcal{M}(S)}$ is constructed as follows :*
  - *A transition from l_{init} to l_0 labelled with the reception of the message $?database(v_{11}, ..., v_{ij})$, where $i \in [1, k]$ and $j \in [1, m+1]$.*
  - *If $(l, \theta, \mu, l') \in \delta$ and μ is a send or a reception of a message, then $(l, \theta, \mu, l') \in \delta_{\mathcal{M}(S)}$.*
  - *If $(l, \theta, \mu, l') \in \delta$ and μ is the atomic process $p(u_1, ..., u_i; v_1, ..., v_j, (\psi, E))$, then $(l, \theta, p_v, l') \in \delta_{\mathcal{M}(S)}$.*

Figure 2.23 depicts the DB-less service $\mathcal{M}(Search)$. The service starts by initializing the set of variables DV. The atomic process *check-item$_v$* modifies the values of the variables of the set DV instead of the database R.

The next two lemmas show the equivalence between testing k-bounded simulation and the test of simulation between the corresponding DB-Less services. The proof is given in appendix A. Note that a valuation of variables α restricted to a subset of variables m is denoted $\alpha_{|m}$.

Lemma 9. *Let S be a Colombo service, $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}^k_0, \mathbb{F}^k, \Delta^k)$ its k-bounded extended state machine and $E(\mathcal{M}(S))$ the extended state machine of DB-less $\mathcal{M}(S)$, then*
- *If $(q_i, \mathcal{I}_i, \alpha_i) \in \mathbb{Q}^k$ then $\exists$ $(q_i, \alpha_i') \in \mathbb{Q}_{\mathcal{M}(S)}$ s.t $\alpha'_{i|Lstore} = \alpha_i$ and $\alpha'_{i|DV} = \mathcal{I}_i$ and*
- *$\forall$ $(q_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (q_j, \mathcal{I}_j, \alpha_j)$, $\exists$ $(q_i, \alpha_i') \xrightarrow{\mu_i'} (q_j', \alpha_j')$ s.t $\alpha'_{j|Lstore} = \alpha_j$ and $\alpha'_{j|DV} = \mathcal{I}_j$.*

Lemma 9 asserts that for each state in the k-bounded state machine of S there exists a corresponding state in the extended state machine of $\mathcal{M}(S)$ s.t the valuation of DV is equal to database $\mathcal{I}$ and the valuation of variables of *Lstore* in the two states are equal.

Lemma 10. *Let S be a Colombo service, $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}^k_0, \mathbb{F}^k, \Delta^k)$ its k-bounded extended state machine and $E(\mathcal{M}(S))$ the extended state machine of DB-less $\mathcal{M}(S)$, then*
- *If $(q_i, \alpha_i') \in \mathbb{Q}_{\mathcal{M}(S)}$ then $\exists$ $(q_i, \mathcal{I}_i, \alpha_i) \in \mathbb{Q}^k$ s.t $\alpha'_{i|Lstore}$=α_i and $\alpha'_{i|DV}$=$\mathcal{I}_i$ and*
- *$\forall$ $(q_i, \alpha_i') \xrightarrow{\mu_i'} (q_j', \alpha_j')$, $\exists$ $(q_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (q_j, \mathcal{I}_j, \alpha_j)$ s.t $\alpha'_{j|Lstore} = \alpha_j$ and $\alpha'_{j|DV} = \mathcal{I}_j$.*

Hence, from lemma 10 and lemma 9 we can derive the following theorem :

Theorem 6. *Let S, S' two Colombo services, then $S \preceq_k S'$ iff $\mathcal{M}(S) \preceq \mathcal{M}(S')$.*

2.5.3 Complexity of k-bounded simulation

In this section, we will prove the 2-EXPTIME completeness of checking k-bounded simulation. First, we show the membership in 2-EXPTIME. Then, the 2-EXPTIME hardness is proved by reduction from the problem of the existence of an infinite execution of an exponentially space bounded alternating Turing machine M (for an input word w of size n, M can explore 2^n cells).

Proposition 2. Let S_1 and S_2 be two Colombo services, testing $S_1 \preceq_k S_2$ is in 2-EXPTIME.

Démonstration. Let S_1 and S_2 be two *Colombo* services with C is the set of constants and X the set of variables in S_1 and S_2. Testing $S_1 \preceq_k S_2$ is achieved by testing $\mathcal{M}(S_1) \preceq \mathcal{M}(S_2)$, where $\mathcal{M}(S_1)$ and $\mathcal{M}(S_2)$ are two DB-less services. We suppose that n is the number of constants and $m + (k \times l)$ the number of variables, with l the arity of $\mathcal{W}$ and m number of variables in S_1, S_2. A region is a set of intervals and a v-order on $X \cup \{\omega\}$. The number of intervals (resp. v-orders) is bounded by $O(n)$ (resp. $O((m + (k \times l))!)$). The number of regions is bounded by $O(n^{m+(k\times l)} \times (m + (k \times l))!)$ and therefore the number of states in the region automata is bounded by $O(|Q_1| \times n^{m+(k\times l)} \times (m + (k \times l))!) = O(2^{log\ |Q_1|+(m+(k\times l))(log\ n+log\ (m+(k\times l)))})$. We conclude that the size of the region automata associated to $\mathcal{M}(S_1)$ and $\mathcal{M}(S_2)$ are respectively $O(2^{2.(log\ |Q_1|+(m+(2^{log\ k}\times l))(log\ n+log\ (m+(2^{log\ k}\times l)))})$ and $O(2^{2.(log\ |Q_2|+(m+(2^{log\ k}\times l))(log\ n+log\ (m+(2^{log\ k}\times l)})$. □

We will prove the 2-exptime-hardness of the problem by proving that, given a Turing machine M working on an exponential space bounded by the size n of an input word w, we can construct two Colombo services $S_{spoiler}$ and $S_{duplicator}$ such that the machine M has an infinite execution on w iff $S_{spoiler} \preceq_k S_{duplicator}$, where $k = 2^n$.

The proof is in the same spirit of the proof of lemma 8. But knowing that the machine M can reach 2^n cells, where n is the size of the input word w. If we use directly two DB-less services in the reduction, we need 2^n variables to store the 2^n cells. Hence, the construction is exponential. To avoid this problem, the services $S_{spoiler}$ and $S_{duplicator}$ will use a database schema $R(A_1, ..., A_n; W)$ to encode the 2^n cells. The key is on n attributes. Taking the domain of $A'i$ to be $\{0, 1\}$, the key is a binary number on n position. Hence, the services can reach 2^n tuples where their keys are ranged from $(0, ..., 0)$ to $(1, ..., 1)$. To simulate the head, the services will use n variables where the value of each variable is either 0 or 1. Then, the actual values of $x_1, ..., x_n$ correspond to the binary number $x_1...x_n$. Hence, a valuation of the variables $x_1, ..., x_n$ is a key for an instance of R. The move to the right of the machine is made by incrementing the binary number $x_1...x_n$, then the new binary number points on the next tuple. Similarly, the move to the left is made by decrementing the binary number $x_1...x_n$ and the new valuation of the variables $x_1, ..., x_n$ points on the previous tuple. The attribute W is used to store the letter of the cell. Assume that, the machine M is at a configuration C and the head points on a cell containing a and $q \xrightarrow{a/bR} q'$ is a transition of M. This transition will be executed by 3 transitions in $S_{duplicator}$:

– first the service tests if the actual tuple contains a,

– then writes b in the attribute W of this tuple,

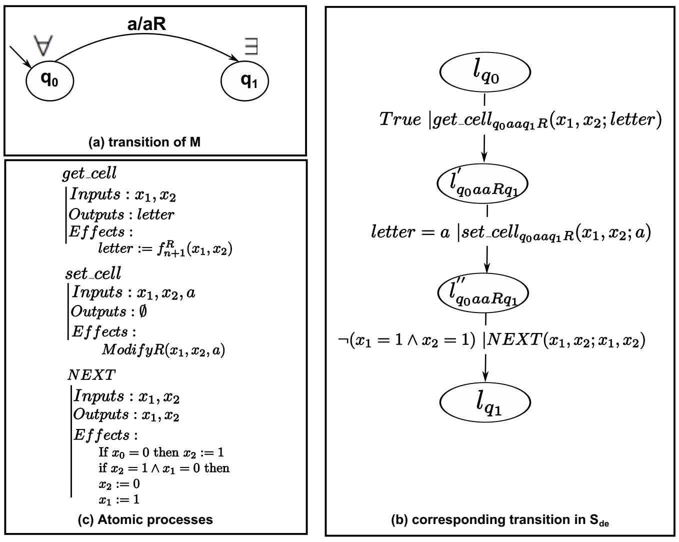


Figure 2.24 – transitions corresponding to $q_0 \xrightarrow{a/aR} q_1$ in M.

– finally moves to the next tuple by executing a binary addition on $x_1 ... x_n$.

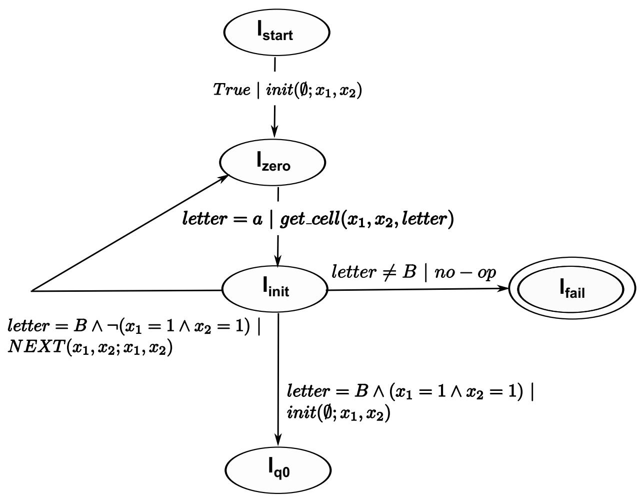
Example 26. Figure 2.24(a) depicts a transition of the machine M of example 19. If the actual value of the cell pointed by the head is equal to a and the machine is in the state q_0, the machine writes a, and moves to the next cell and reaches the state q_1. Suppose the input word is ab, so n=2. The part of $S_{duplicator}$ representing this transition starts by storing the value of the attribute W corresponding to the tuple identified with the key x_1x_2 in the variable letter (i.e., $letter := f^R_{n+1}(x_1, x_2)$) using the atomic process *get_cell*. Then, the service tests if letter = a and writes in the current tuple the new value of W with *set_cell*. After that, the service increments the binary number x_1x_2 using the atomic process NEXT. As a consequence, x_1x_2 points on the next tuple. The guard $\neg(x_1 = 1 \wedge x_2 = 1)$ prevents a move to the right if the service points on the last cell. Note that, when encoding a transition of M, the service $S_{spoiler}$ will not contain the guard letter = a, because $S_{spoiler}$ will encode all transitions that the machine can do infinitely often.

The services $S_{duplicator}$ and $S_{spoiler}$ will start with an initialization part where they :

1. Check if all tuples identified with key from $(0, ..., 0)$ to $(1, ..., 1)$ contain the symbol B, which means the 2^n cells are empty. In following, we will call the database instances which satisfy this condition *standard* instances and those that do not satisfy it *non-standard* instances.
2. initialize the n first tuples with the n letters of the input word w.

Example 27. Continuing with our example, figure 2.25 depicts the initialization part of the two services. The services start by assigning zero to x_1 and x_2, then check if the value of the attribute W of the actual tuple identified with the key x_1x_2 is equal to B. If x_1x_2 points on an empty tuple and it is not the last tuple (key equal 11), the services increment the key and test the next tuple. If one of them does not contain B, then the database is *non-standard* and there is simulation. If all tuples ranged from 00 to 11 contain B, the services reinitialize the variables to zero.

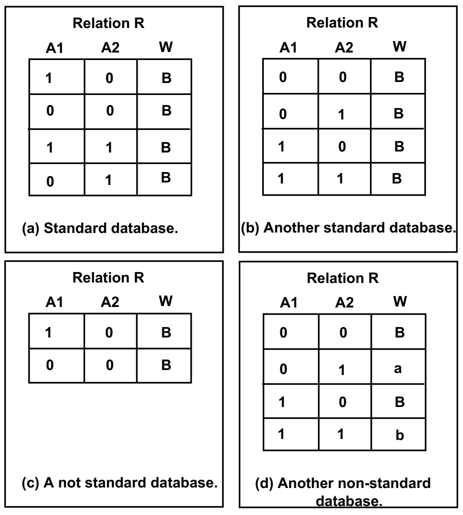
For all executions starting with a *non-standard* database, $S_{spoiler} \preceq S_{duplicator}$ is true, because the two services have the same initialization part. Figure 2.26 depicts examples of



l_{start}
$True \mid init(\emptyset; x_1, x_2)$
l_{zero}
$letter = a \mid get\_cell(x_1, x_2, letter)$
l_{init}
$letter \neq B \mid no - op$
l_{fail}
$letter = B \wedge \neg(x_1 = 1 \wedge x_2 = 1) \mid$
$NEXT(x_1, x_2; x_1, x_2)$
$letter = B \wedge (x_1 = 1 \wedge x_2 = 1) \mid$
$init(\emptyset; x_1, x_2)$
l_{q0}

FIGURE 2.25 – initialization part of $S_{duplicator}$ and $S_{spoiler}$.

standard and *non-standard* databases. As we can see, the order of tuples is not important for standard databases (Figure 2.26(a) and figure 2.26(b)). The database depicted at figure 2.26(c) fails in the initialization part because $f_3^R(1,1)$ and $f_3^R(0,1)$ are equal to ω, and the database depicted at figure 2.26(d) is *non-standard* because there are tuples with values different from B for the attribute W.



Relation R

A1	A2	W
1	0	B
0	0	B
1	1	B
0	1	B

(a) Standard database.

Relation R

A1	A2	W
0	0	B
0	1	B
1	0	B
1	1	B

(b) Another standard database.

Relation R

A1	A2	W
1	0	B
0	0	B

(c) A not standard database.

Relation R

A1	A2	W
0	0	B
0	1	a
1	0	B
1	1	b

(d) Another non-standard database.

FIGURE 2.26 – Standard database.

Now we will give the formal definition of atomic processes.

Atomic processes $\mathcal{P}$ is the set of all atomic processes used to encode the actions of the machine M :

- for each transition $q \xrightarrow{a/bR} q'$ in M :
  - $get\_cell_{qabq'R}(x_1, ..., x_n; letter, CE)$ is an atomic process with one conditional effect :
    - θ : true.
    - ev : letter $:= f_{n+1}^R(x_1, ..., x_n)$.
  - $set\_cell_{qabq'R}(x_1, ..., x_n, b)$ is an atomic process with one conditional effect :

- θ : true.
- ev : MODIFY R$(x_1, ..., x_n; b)$.
- NEXT$(x_1, ..., x_n; x_1, ..., x_n, \{CE\})$ is an atomic process with n conditional effect where $CE = \{(\theta_n, v_n)\} \cup \{(\theta_k, v_k) \mid k \in [n-1, 1]\}$ and
  - $\theta_n : x_n{=}0$
    $v_n : x_n := 1$
  - $\theta_k : x_n = 1 \wedge x_{n-1} = 1 \wedge ... \wedge x_{k-1} = 1 \wedge x_k = 0$
    $v_k : x_n := 0 \wedge x_{n-1} := 0 \wedge ... \wedge x_{k-1} := 0 \wedge x_k := 1$
- for each transition $q \xrightarrow{a/bL} q'$ in M
  - $get\_cell_{qabq'L}(x_1, ..., x_n; letter, CE)$ is an atomic process with one conditional effect :
  - θ : true.
  - ev : letter $:= f^R_{n+1}(x_1, ..., x_n)$.
  - $set\_cell_{qabq'L}(x_1, ..., x_n, b)$ is an atomic process with one conditional effect :
  - θ : true.
  - ev : MODIFY R$(x_1, ..., x_n; b)$.
  - PREVIOUS$(x_1, ..., x_n; x_1, ..., x_n, \{CE\})$ is an atomic process with n conditional effect
    where $CE = \{(\theta_n, v_n)\} \cup \{(\theta_k, v_k) \mid k \in [n-1, 1]\}$ and
  - $\theta_n : x_n{=}1$
    $v_n : x_n := 0$
  - $\theta_k : x_n = 0 \wedge x_{n-1} = 0 \wedge ... \wedge x_{k-1} = 0 \wedge x_k = 1$
    $v_k : x_n := 1 \wedge x_{n-1} := 1 \wedge ... \wedge x_{k-1} := 1 \wedge x_k := 0$

Now we give the formal definition of the service $S_{duplicator}$

Service $\mathbf{S_{duplicator}}$. Let $GA(S_{duplicator}) = \langle Q_{duplicator}, \delta_{duplicator}, l'_{start}, LStore(S_{duplicator})\rangle$ where :

- the set of states of $S_{duplicator}$ are the following
  - For each state q in M, a state l_q.
  - a set of states $l_{copy}, l_{zero}, l_{init}, l_{fail}$, where l_{fail} is final.
  - $\{choice_{qbdq'} \mid q \xrightarrow{a/bd} q'$ *in* M *and* q *an exitential state and* $d = R/L\}$
  - $\{l'_{qbdq'},\ l''_{qbdq'} \mid q \xrightarrow{a/bd} q'$ *in* M *and* $d = R/L\}$
  - l'_{start} is the initial state.
  - for each w_i (i'th letter of the input word) a state l_{w_i}.
- $Lstore(S_{duplicator}) = \{x_1, ..., x_n\} \cup \{letter\}$.
- $\delta_{duplicator}$ is composed of the following sets of transitions :
  - $(l'_{start}$, true, $init(\emptyset; letter, x_1, ..., x_n)$, $l_{zero})$.
  - $(l_{zero}$, true, $get\_cell(x_1, ..., x_n; letter)$, $l_{init})$.
  - $(l_{init}$, $letter = B \wedge \neg(x_1 = 1 \wedge ... \wedge x_n = 1)$, $NEXT(x_1, ..., x_n; x_1, ..., x_n)$, $l_{zero})$.
  - $(l_{init}$, $letter \neq B$, no-op, $l_{fail})$.
  - $(l_{init}$, $letter = B \wedge x_1 = 1 \wedge ... \wedge x_n = 1$, $init(\emptyset; letter, x_1, ..., x_n)$, $l_{w_1})$.
  - $(l_{copy}$, true, $\mathcal{P} \cup \{!m()\}$, $l_{copy})$, where $\mathcal{P}$ is the set of all atomic process in $S_{duplicator}$.
  - for each w_i (i'th letter of the input word) two transition :
    - $(l_{w_{i-1}}$, true, Insert$(x_1, ..., x_n, w_i)$, $l_{w_i})$
    - $(l_{w_i}$, true, NEXT$(x_1, ..., x_n)$, $l_{w_{i+1}})$
  - For each transition $q \xrightarrow{a/bd} q'$ in M, where q is a universal state :
    - if $d = R$ then :

- $\{(l_q, \textsf{true}, get\_cell_{qabq'R}(x_1,...,x_n; letter), l'_{qbRq'})\}$.
- $\{(l'_{qbRq'}, letter = a, set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset), l''_{qbRq'})\}$.
- $\{(l''_{qbRq'}, \neg(x_1 = 1 \wedge ... \wedge x_n = 1), NEXT(x_1,...,x_n;x_1,...,x_n), l_{q'})\}$.
- $\{(l_q, \textsf{true}, !m(), l_{copy})\}$.
- $\{(l_q, \textsf{true}, \mathcal{P}\backslash\mathcal{P}_{test\_cellq}, l_{copy})\}$. $\mathcal{P}_{test\_cellq}$ is the set of atomic process *test_cell* used to encode transition from state q of M.

– if $d = L$ then :
- $\{(l_q, \textsf{true}, get\_cell_{qabq'L}(x_1,...,x_n; letter), l'_{qbRq'})\}$.
- $\{(l'_{qbRq'}, letter = a, set\_cell_{qabq'L}(x_1,...,x_n,b;\emptyset), l''_{qbRq'})\}$.
- $\{(l''_{qbRq'}, \neg(x_1 = 0 \wedge ... \wedge x_n = 0), Previous(x_1,...,x_n;x_1,...,x_n), l_{q'})\}$.
- $\{(l_q, \textsf{true}, !m(), l_{copy})\}$.
- $\{(l_q, \textsf{true}, \mathcal{P}\backslash\mathcal{P}_{test\_cellq}, l_{copy})\}$. $\mathcal{P}_{test\_cellq}$ is the set of atomic process *test_cell* used to encode transition from state q of M.

– For each transition $q \xrightarrow{a/bd} q'$ in M, where q is a existential state :
– if $d = R$ then :
- $\{(l_q, \textsf{true}, !m(), choice_{qbRq'})\}$
- $\{(l_q, \textsf{true}, \mathcal{P} \setminus \{!m()\}, l_{copy})\}$.
- $\{(choice_{qbRq'}, \textsf{true}, get\_cell_{qabq'R}(x_1,...,x_n; letter), l'_{qbRq'})\}$.
- $\{(l'_{qbRq'}, letter = a, set\_cell_{qabq'R}(x_1,...,x_n,b), l''_{qbRq'})\}$.
- $\{(l''_{qbRq'}, \neg(x_1 = 1 \wedge ... \wedge x_n = 1), NEXT(x_1,...,x_n;x_1,...,x_n), l_{q'})\}$.
- $\{(choice_{qbRq'}, \textsf{true}, \mathcal{P} \setminus \{get\_cell_{qabq'R}(x_1,...,x_n; letter)\}, l_{copy})\}$.

– if $d = L$ then :
- $\{(l_q, \textsf{true}, !m(), choice_{qbLq'})\}$
- $\{(l_q, \textsf{true}, \mathcal{P} \setminus \{!m()\}, l_{copy})\}$.
- $\{(choice_{qbLq'}, \textsf{true}, get\_cell_{qabq'L}(x_1,...,x_n; letter), l'_{qbLq'})\}$.
- $\{(l'_{qbLq'}, letter = a, set\_cell_{qabq'L}(x_1,...,x_n,b), l''_{qbLq'})\}$.
- $\{(l''_{qbLq'}, \neg(x_1 = 0 \wedge ... \wedge x_n = 0), PREVIOUS(x_1,...,x_n;x_1,...,x_n), l_{q'})\}$.
- $\{(choice_{qbLq'}, \textsf{true}, \mathcal{P} \setminus \{get\_cell_{qabq'L}(x_1,...,x_n; letter)\}, l_{copy})\}$.

The next lemma asserts that, each configuration C of the execution of an alternating Turing machine M on the input w has a corresponding configuration in the extended state machine of $S_{duplicator}$. The proof is by induction (the details are given in appendix A).

Lemma 11. *Each configuration C of the execution of an alternating Turing machine M on the input w has a corresponding configuration in the extended state machine of $S_{duplicator}$.*

Service $\mathbf{S_{spoiler}}$. Let $GA(S_{spoiler}) = \langle Q_{spoiler}, \delta_{spoiler}, q_{start}, LStore(S_{spoiler})\rangle$ where :

- the set of states of $S_{spoiler}$ are following
  - a set of states $q_{start}, q_{zero}, q_{init}, q_{fail}, q_\forall, q_\exists$, where q_{fail} is final.
  - $\{q_{qbdq'}, q'_{qbdq'} \mid q \xrightarrow{a/bd} q' \;\; in \;\; M\}$
  - q'_{start} is the initial state.
  - for each w_i (i'th letter of the input word) a state q_{w_i}.
- $Lstore(S_{duplicator}) = \{z_1,...,z_n\} \cup \{letter\}$.
- $\delta_{duplicator}$ is as follows :
  - The part of initialization of the input word and checking the database is the same as in $S_{duplicator}$, the difference is after storing the last letter of the input word w,

$S_{spoiler}$ goes to the state $q_\forall$.
- $(q_\forall, \mathsf{true}, !m(), q_\exists)$.
- For each transition $q \overset{a/bd}{\longrightarrow} q'$ in M, where q is a universal state :
  - if $d = R$ then :
    - $\{(q_\forall, \mathsf{true}, get\_cell_{qabq'R}(z_1, ..., z_n; letter), q_{qbdq'})\}$.
    - $\{(q_{qbdq'}, \mathsf{true}, set\_cell_{qabq'R}(z_1, ..., z_n, b; \emptyset), q'_{qbdq'})\}$.
    - $\{(q'_{qbdq'}, \neg(z_1 = 1 \wedge ... \wedge z_n = 1), NEXT(z_1, ..., z_n; z_1, ..., z_n), q_\forall)\}$.
  - if $d = L$ then :
    - $\{(q_\forall, \mathsf{true}, get\_cell_{qabq'L}(z_1, ..., _z n; letter), q_{qbdq'})\}$.
    - $\{(q_{qbdq'}, \mathsf{true}, set\_cell_{qabq'L}(z_1, ..., z_n, b; \emptyset), q'_{qbdq'})\}$.
    - $\{(q'_{qbdq'}, \neg(z_1 = 0 \wedge ... \wedge z_n = 0), Previous(z1, ..., z_n; z_1, ..., z_n), q_\forall)\}$.
- For each transition $q \overset{a/bd}{\longrightarrow} q'$ in M, where q is a existential state :
  - if $d = R$ then :
    - $\{(q_\exists, \mathsf{true}, get\_cell_{qabq'R}(z_1, ..., z_n; letter), q_{qbdq'})\}$.
    - $\{(q_{qbdq'}, \mathsf{true}, set\_cell_{qabq'R}(z_1, ..., z_n, b; \emptyset), q'_{qbdq'})\}$.
    - $\{(q'_{qbdq'}, \neg(z_1 = 1 \wedge ... \wedge z_n = 1), NEXT(z_1, ..., z_n; z_1, ..., z_n), q_\forall)\}$.
  - if $d = L$ then :
    - $\{(q_\exists, \mathsf{true}, get\_cell_{qabq'L}(z_1, ..., z_n; letter), _q qbdq')\}$.
    - $\{(q_{qbdq'}, \mathsf{true}, set\_cell_{qabq'L}(z_1, ..., z_n, b; \emptyset), q'_{qbdq'})\}$.
    - $\{(q'_{qbdq'}, \neg(z_1 = 1 \wedge ... \wedge z_n = 1), PREVVIOUS(z_1, ..., z_n; z_1, ..., _z n), q_\forall)\}$.

Lemma 12 shows the connection between the existence of infinite execution of the machine M on the word w and the test of simulation between $S_{spoiler}$ and $S_{duplicator}$. The proof is given in appendix A.

Lemma 12. *Given an alternating Turing machine M working in space bounded by the size of the input w, M has an infinite computation on w iff $S_{spoiler} \preceq S_{duplicator}$.*

From lemma 12 and knowing that the problem of existence of an infinite execution of an alternating Turing machine working on a space exponentially bounded by the size of the input word is 2-EXPTIME-hard, we can derive the following lemma :

Lemma 13. *Given two Colombo services S, S', checking whether $S \preceq_k S'$ is* 2-EXPTIME-*hard.*

Hence, the following theorem can now be derived from lemma 13 and proposition 2 :

Theorem 7. *Given two Colombo services S, S', checking whether $S \preceq_k S'$ is* 2-EXPTIME-*complete.*

2.6 Conclusion

In this chapter, we studied the decidability and the complexity issues related to the simulation problem in the framework of the Colombo model. Our results, ranging from EXPTIME to undecidability show that the marriage between data and web service business protocols gives rise to some challenging issues. The decidability and complexity results, EXPTIME-complete for $Colombo^{DB=\emptyset}$ and 2-EXPTIME-complete for $Colombo^{bound}$ are far from being straightforward, due to the fact we are dealing with infinite state systems.

This chapter proposed also a *symbolic* procedure based on the notion of *region automata* to handle the infiniteness of the framework.

The next chapter will be devoted to the definition of a generic framework that generalizes the Colombo model, where the messages exchanged as well as the updates over the databases are expressed using queries. The main goal is to identify the parameters that impact the decidability and the complexity of the simulation for data-centric web services.

Chapitre 3

Data-Centric Generic Model

This chapter is organized as follows : we start by some preliminaries in section 3.1. In section 3.2 we introduce a generic data centric model and define the associated simulation problem. Section 3.3 describes our results regarding decidability and complexity of simulation for *guarded* services (i.e., generic services with guards and empty send messages). Section 3.4 considers the case of *send* services (i.e., unguarded generic services with send messages) and show decidability and complexity results of simulation in this context. Section 3.5 is devoted to *insert* services (i.e., unguarded generic services with insert actions).

3.1 Notations

Let $\mathcal{L}$ be a query language, $R \in \mathcal{R}$ be a relation schema and let I be a database over $\mathcal{R}$ with $r \in I$ a relation over R. Let q, q' be $\mathcal{L}$ queries with $schema(q) = schema(q') = schema(R)$. An update language, noted $\mathcal{L}_U$, defines the update queries that can be used to modify a database. In this thesis, we focus our attention on insertions. An insert query is an expression $U = \mathsf{insert\ R\ (q)}$. The semantics is that the answers of q are inserted in R (i.e., $U(r) = r \cup q(I)$). If q is a query and I a database, we write $q(I)$ to denote the set of answers of q when it is executed on I. In the similar way, if U is an update query, we write $U(I)$ to denote the database obtained after the application of the insert U on I. We use $\mathcal{L}^{insert(\mathcal{I})}$ to denote an update language restricted to insertions expressed in the language $\mathcal{I}$. Let $\mathcal{L}$ be a boolean query language, we denote by $\wedge^b$ and $\neg^b$ the conjunction and negation operators applied on $\mathcal{L}$ formulas. A formula θ in $\mathcal{L} \cup \{\wedge^b, \neg^b\}$ is constructed with the following recursive definition : $\theta : := \theta \wedge \theta \mid (\neg)\beta$ where $\beta \in \mathcal{L}$.

3.2 Generic web service

We start this section with the formal definition of a *generic* web service and give its semantics, then we define the relation of simulation in the context of this model.

3.2.1 Generic web service model

A generic data-centric web service is a state machine :

- acting on a *global* database (shared with all services of the system) and a *local* (private) database.
- labelled with guards on the transitions. The guards are boolean queries defined over the databases and expressed in a language $\mathcal{L}_T$.

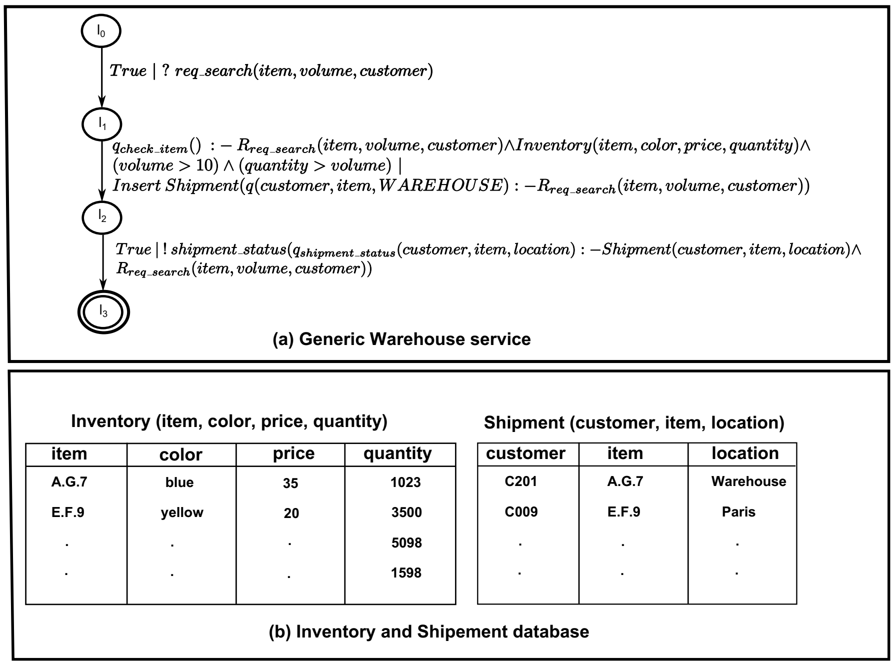
– the service communicate through messages. The messages exchanged are relations where the outgoing messages are queries defined over the local and the global databases expressed in a language $\mathcal{L}_S$.
– the service can modify the databases (*local* and *global*), through update queries expressed in the language $\mathcal{L}_U$.

Definition 13. ***generic web service*** *Let $\mathcal{L}_T$ be a boolean query language, $\mathcal{L}_S$ a query language and $\mathcal{L}_U$ an update query language. An $(\mathcal{L}_T, \mathcal{L}_S, \mathcal{L}_U)^{?,p}$ service S is a tuple $S = \langle \Sigma, \mathcal{W}, L, l_0, F, \delta \rangle$ where :*
– *Σ is a set of messages. We associated to a message $m(p_1, \ldots, p_n)$ the relation schema R_m where $schema(R_m) = (A_1, \ldots, A_n)$.*
– *$\mathcal{W} = \mathcal{W}_l \cup \mathcal{W}_g \cup \mathcal{W}_m$ is a relational schema made of three disjoint schemas :*
  – *$\mathcal{W}_l$ the local schema of the service,*
  – *$\mathcal{W}_g$ the global schema (i.e., visible to all other services) and*
  – *$\mathcal{W}_m$ the set of messages schema.*
– *L is a finite set of locations (or control states) with $l_0 \in L$ the initial state, and $F \subseteq L$ a set of final states,*
– *The transition relation δ contains tuples (l, q, μ, l') where $l, l' \in L$, q is a boolean query in $\mathcal{L}_T$ defined over $\mathcal{W}$, and μ has one of the following forms :*
  – *$\mu = ?m(p_1, ..., p_n)$ (incoming message) or*
  – *$\mu = !m(q')$ (send message). q' is a query expressed in the language $\mathcal{L}_S$ and defined over $\mathcal{W}$ such that schema(m)= schema(q') or*
  – *$\mu = u$ where u is an update query expressed in the language $\mathcal{L}_U$ and defined over $\mathcal{W}_l \cup \mathcal{W}_g$.*

Example 28. Figure 3.1 depicts a service *Warehouse*. This service starts when receiving a message *req_search* containing the product, the quantity requested and the ID of the customer. Then the service checks whether the product is listed in the database Inventory and the quantity requested is grater than 10 and grater the available stock. If it is the case, the service processes the shipment by adding a row in the database Shipment and records the customer ID, the item code and the actual location (initially the warehouse). Finally, it sends the message *Shipment_status* containing the actual status of the order. In this example, the global database schema $\mathcal{W}_g$=$\{Inventory\}$, the local database schema $\mathcal{W}_l$=$\{Shipment\}$ and the message database schema is $\mathcal{W}_m$= $\{R_{req\_search}\}$. The languages $\mathcal{L}_U$, $\mathcal{L}_S$ are conjunctive queries (CQ) and the language of guards $\mathcal{L}_T$ is boolean conjunctive queries with arithmetic comparisons.

3.2.2 Extended state machines

We use the notion of an extended automata to define the semantic of a generic web service. At every point in time, the behavior of an instance of a generic service (or simply a service) is determined by its *instantaneous description (ID)*. An ID of a service is given by a pair $id = (l, I)$ where l is its current location (or control state) and I the current database instance over the schema $\mathcal{W}$. We note $I = I_l \cup I_g \cup I_m$, with I_l the local database (i.e., the database over the schema $\mathcal{W}_l$), I_g the global database (i.e., the database over the schema $\mathcal{W}_g$), and I_m the messages database over the schema $\mathcal{W}_m$. In the sequel, we use the notation id^{db} to refer to the database associated with the instantaneous description id (i.e., $id^{db} = I$) and id^{db_l}, id^{db_g}, id^{db_m} to refer respectively to the local database I_l, global database I_g and messages database I_m.


l0
True | ? req_search(item, volume, customer)
l1
q_check_item() : − R_req_search(item, volume, customer)∧Inventory(item, color, price, quantity)∧ (volume > 10) ∧ (quantity > volume) |
Insert Shipment(q(customer, item, WAREHOUSE) : −R_req_search(item, volume, customer))
l2
True | ! shipment_status(q_shipment_status(customer, item, location) : −Shipment(customer, item, location)∧ R_req_search(item, volume, customer))
l3
(a) Generic Warehouse service
(b) Inventory and Shipement database

Inventory (item, color, price, quantity)

item	color	price	quantity
A.G.7	blue	35	1023
E.F.9	yellow	20	3500
.	.	.	5098
.	.	.	1598

Shipment (customer, item, location)

customer	item	location
C201	A.G.7	Warehouse
C009	E.F.9	Paris
.	.	.
.	.	.

Figure 3.1 – Data-centric generic web service *Warehouse.*

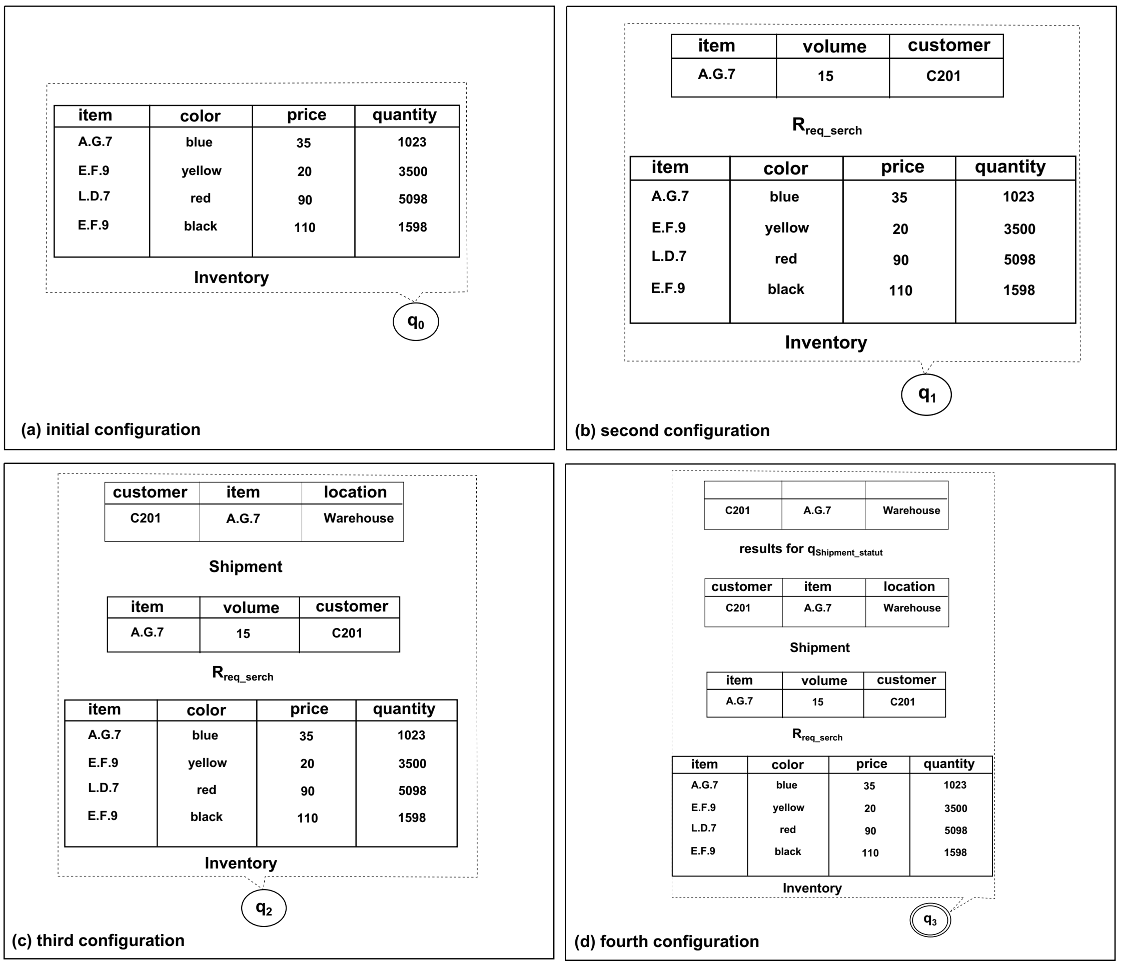
A run of a generic service starts with an arbitrary instance over $\mathcal{W}_g$ and $\mathcal{W}_l$ and an empty instance of $\mathcal{W}_m$. The database representing the incoming message is not cumulative, in the sense that each time the service receives a message m, the associated instance R_m is overwritten.

Definition 14. ***service runs (executions)*** *Let $S = \langle \mathcal{W}, L, l_0, F, \delta \rangle$ be a service. A run σ of S is a finite sequence $\sigma = id_0 \xrightarrow{\mu_0} id_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{n-1}} id_n$ which satisfies the following conditions :*

- *(Initiation) $id_0 = (l_0, I_0)$ is the initial state of the run and I_0 is an arbitrary instance over $\mathcal{W}_g$ and $\mathcal{W}_l$. $\mathcal{W}_m$ starts empty.*
- *(Consecution) $\forall i \in [0, n-1]$, there is a transition $(l_i, q, \mu, l_{i+1}) \in \delta$ such that $id_i^{db} \models q$ and one of the following conditions holds :*
  - *μ =?$m()$ then μ_i =?$m(r_{m_{i+1}})$, with $r_{m_{i+1}}$ an instance over R_m and $id_{i+1} = (l_{i+1}, I_{l_i}, I_{g_i}, I_{m_{i+1}})$ with $I_{m_{i+1}} = (I_{m_i} \setminus \{r_{m_i}\}) \cup \{r_{m_{i+1}}\}$ and r_{m_i} the instance of R_m in $id_i^{db_m}$.*
  - *μ =!$m(q_m)$ then μ_i =!$m(q_m(id_i^{db}))$ and $id_{i+1} = (l_{i+1}, id_i^{db})$.*
  - *$\mu = u$ then μ_i = "u" and $id_{i+1} = (l_{i+1}, u(id_i^{db}))$.*

A service moves from an id_i to id_j according to the mechanics defined by its set of transitions. If $id_i \xrightarrow{\mu_i} id_j$ satisfies the consecution condition above, we say that μ_i is allowed from id_i.

Example 29. Figure 3.2 depicts a run of the service *Warehouse*. The service starts with an arbitrary instance of *inventory*, all the others relations are empty (figure 3.2.a). Then, the service receives the message ?*req_search* and moves to the control state q_1 with the same global database but with a new instance of R_{req_search}, representing the instance received (figure 3.2.b). Then, if the conditions are verified, the service adds a tuple in the database Shipment (figure 3.2.c). Finally, it sends the shipment status in the message *Shipment_status*, the result of the query $q_{Shipment\_status}$ is depicted in figure 3.2.d. At



(a) Inventory

item	color	price	quantity
A.G.7	blue	35	1023
E.F.9	yellow	20	3500
L.D.7	red	90	5098
E.F.9	black	110	1598

(b) R_{req_serch}

item	volume	customer
A.G.7	15	C201

(b) Inventory

item	color	price	quantity
A.G.7	blue	35	1023
E.F.9	yellow	20	3500
L.D.7	red	90	5098
E.F.9	black	110	1598

(c) Shipment

customer	item	location
C201	A.G.7	Warehouse

(c) R_{req_serch}

item	volume	customer
A.G.7	15	C201

(c) Inventory

item	color	price	quantity
A.G.7	blue	35	1023
E.F.9	yellow	20	3500
L.D.7	red	90	5098
E.F.9	black	110	1598

(d) results for $q_{Shipment\_statut}$

C201	A.G.7	Warehouse

(d) Shipment

customer	item	location
C201	A.G.7	Warehouse

(d) R_{req_serch}

item	volume	customer
A.G.7	15	C201

(d) Inventory

item	color	price	quantity
A.G.7	blue	35	1023
E.F.9	yellow	20	3500
L.D.7	red	90	5098
E.F.9	black	110	1598

Figure 3.2 – A run of the data-centric generic web service *Warehouse*.

this point of execution, if the service receives another message *req_search*, the previous instance of R_{req_search} is deleted and replaced by the new one.

The semantics of a generic service can be captured by the following notion of an extended infinite state machine.

Definition 15. ***extended state machine*** *Let* $S = \langle \mathcal{W}, L, l_0, F, \delta \rangle$ *be a generic service. The associated infinite state machine, noted* $E(S)$*, is a tuple* $E(S) = (\mathbb{L}, \mathbb{L}_0, \mathbb{F}, \Delta)$ *where :*
- $\mathbb{L} = \{(l, I)\}$ *is the set of all instantaneous description.*
- $\mathbb{L}_0 = \{(l_0, I)\}$ *is the set of initial configurations, with* I *an arbitrary database over* $\mathcal{W}$ *where the messages database are empty.*
- $\mathbb{F} = \{(l, I) \mid l \in F\}$ *is the set of final instantaneous description.*
- Δ *is an (infinite) set of transitions of the form* $\tau = (l_i, I_i) \xrightarrow{\mu_i} (l_j, I_j)$ *such that* μ_i *is allowed from* (l_i, I_i) *(i.e.,* τ *satisfies the* consecution *condition).*

A finite run of $E(S)$ is any finite path from an initial state of $E(S)$ to a final state. Given an initial state id_0 of $E(S)$, all the possible runs of $E(S)$ starting from id_0 can be captured by an (infinite) execution tree noted $tree(id_0)$ having as its root id_0.

Example 30. Figure 3.3(a) depicts the infinite tree of executions of the service *Warehouse* for a initial instance over the *global* (*Inventory*) and the *local* (*Shipment*) schema. The

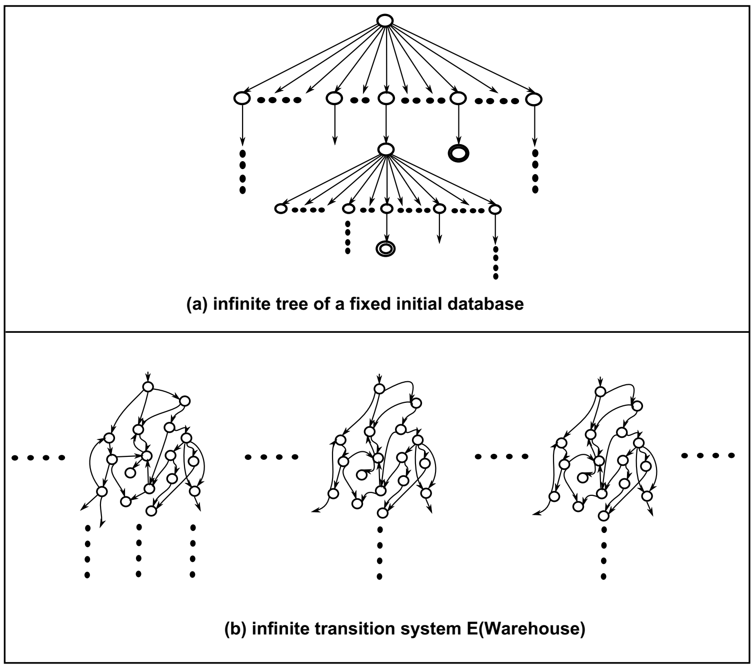

(a) infinite tree of a fixed initial database
(b) infinite transition system E(Warehouse)

Figure 3.3 – E(*Warehouse*)

source of infiniteness comes from the infinite number of instances that the service can receive with the message *req_search*. Another source of infiniteness, is the number of initial databases as depicted in figure 3.3(b).

3.2.3 Simulation and weak simulation

We introduce now the notion of *weak* transition, *weak* simulation and then the relation of simulation for generic services.

Weak Simulation

When the modification of the *local* database ($\mathcal{W}_l$) is allowed, we talk about *weak* simulation. This is because this kind of modification, called hereafter *silent* transitions, is not observable from an external point of view (i.e., another service or the client). Before giving the definition of the *weak* simulation, we introduce the notion of *weak* transition :

Definition 16. ***Weak transition*** *Let $S = \langle \mathcal{W}, L, l_0, F, \delta \rangle$ be a data-centric generic service, and $E(S) = (\mathbb{L}, \mathbb{L}_0, \mathbb{F}, \Delta)$ its extended state machine.*

A weak transition denoted by $(id_1) \overset{\mu_n}{\Longrightarrow} (id_{n+1})$ is the path $(id_1) \xrightarrow{\mu_1} (id_2) \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} (id_{n+1})$ in $E(S)$ where $\mu_i = u(id_i^{db_l}))$ with $i \in [1, n-1]$ and μ_n is not a silent transition.

A *weak* transition collapses a path where only observable actions are kept (i.e., the modification of the global database as well as the exchanged messages).

Definition 17. *Let S and S' be two $(\mathcal{L}_T, \mathcal{L}_S, \mathcal{L}_U)^{?,p}$ services defined over the same global database schema and let $E(S) = (\mathbb{L}, \mathbb{L}_0, \mathbb{F}, \Delta)$ and $E(S') = (\mathbb{L}', \mathbb{L}'_0, \mathbb{F}', \Delta')$ be respectively their associated extended state machines.*

- *Let $(id_i, id'_i) \in \mathbb{L} \times \mathbb{L}'$. The state id_i weakly simulates id'_i, noted $id_i \preceq_w id'_i$, iff :*
  - *if $id_i \in \mathbb{F}$ then $id_i^{'} \in \mathbb{F}^{'}$ and*

– $id_i^{db_g} = id_i'^{db_g}$, *and*
– $\forall id_i \xRightarrow{\mu_i} id_j \in \Delta$, *there exists* $id_i' \xRightarrow{\mu_i'} id_j' \in \Delta'$ *such that*
  – *if* $\mu_i = ?m(r_m)$ *then* $\mu_i^{'} = ?m(r_m)$ *and* $id_j \preceq_w id_j'$.
  – *if* $\mu_i = !m(q)$ *then* $\mu_i^{'} = !m(q^{'})$ *and* $q(id_i^{db}) = q^{'}(id_i^{'db})$ *and* $id_j \preceq_w id_j'$.
  – *if* $\mu_i = u$ *then* $\mu_i^{'} = u'$ *and* $u(id_i^{db_g}) = u(id_i^{'db_g})$ *and* $id_j \preceq_w id_j'$
– $E(S) \preceq_w E(S')$ *iff* $\forall id_0 \in \mathbb{L}_0, \exists id_0' \in \mathbb{L}'_0$ *such that* $id_0 \preceq_w id_0'$
– $S \preceq_w S'$ *iff* $E(S) \preceq_w E(S')$

When we study the simulation, the *local* database is always empty. The relation of simulation is defined as the weak simulation, but we replace $\preceq_w$ by $\preceq$ and $\Longrightarrow$ by $\longrightarrow$ (i.e., when the database is empty, this means there is no silent transition, hence no weak transitions). The external visible behavior of a service is defined here with respect to the content of the global database as well as the exchanged *concrete* messages (i.e., message name together with the instance exchanged).

The existence of a simulation relation ensures that each execution tree of S is also an execution tree of S' (in fact, a subtree of S'), modulo a relabeling of control states.

3.2.4 Analyzing various classes of the generic model

We investigate the decidability and the complexity issues of simulation for various classes of our generic model, each class is characterized by :
– the type of actions supported in the model ,e.g., the service can only send messages or can only insert in the database, ...etc,
– the languages used to instantiate respectively $\mathcal{L}_T$, $\mathcal{L}_U$ and $\mathcal{L}_S$ (e.g., CQ, etc...),
– the presence or not of the local database (i.e., in the presence of local database, we study the weak simulation).

Table 3.1 – sub-models.

Class of services	$\mathcal{A}$	$\mathcal{W}_g$	$\mathcal{W}_l$
$(\mathcal{L}_T, \emptyset, \emptyset)$	!	+	-
$(\emptyset, \mathcal{L}_S, \emptyset)$	!	+	-
$(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$	*Insertion*	+	-
$(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})^p$	*Insertion*	+	+
$(\mathcal{L}_T, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$	*Insertion*	+	-

Table 3.1 depicts the different classes we study. We consider the following parameters :
– $\mathcal{A}$ specifies the type of actions allowed. For example, $(\mathcal{L}_T, \emptyset, \emptyset)$ denotes the class of services that are able to send messages (symbolized by ! in the table 3.1), but the messages are empty because $\mathcal{L}_S$= $\emptyset$. $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ denotes the services that are only able to make insertion in the database using the language $\mathcal{L}_I$.
– Regarding the other columns of table 3.1, the symbol + denotes the presence of the component in the considered class while the symbol − indicates that the corresponding component is not provided by the class.
We consider more precisely the following classes :
– *Update-free* services. This class represents services which are not able to modify the databases. This class enables to focus on the role played by the language of guards

$(\mathcal{L}_T)$ and the query language used to send messages $(\mathcal{L}_S)$ on the decidability of the simulation. The main sub-classes investigated in this class are described below :

 – Guarded services $(\mathcal{L}_T, \emptyset, \emptyset)$. This class deals with guards expressed in a language $\mathcal{L}_T$. An $(\mathcal{L}_T, \emptyset, \emptyset)$ service can only send empty messages. Our purpose is to study the impact of the guards language on checking the simulation. The ability of sending empty messages is added for convenience to simplify the proofs.
 – Send services $(\emptyset, \mathcal{L}_S, \emptyset)$. This class represent services which can only send messages. The content of an outgoing message is the result of a query expressed in the language $\mathcal{L}_S$. This class enables to analyze the impact of $\mathcal{L}_S$ on the simulation.

– *Insert* services. This class describes services without guards. The considered services are able to insert data in the global database. In this context, we study the simulation as well as the *weak* simulation relation. The main sub-classes investigated in this context are described below :
 – $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services. This class focuses on services able to insert data in the global database (there is no other action than the insertion in the global database). In this case a service can encode, for example, a recursive program. As we shall see, this is an important property that it will be exploited to prove the undecidability of simulation for some instances of this class.
 – $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})^p$ services. This class is used to study the *weak simulation.* A service can insert in the *global* and the *local* database. An insertion in the *local* database is considered as a *silent* transition. We show that, the *weak* simulation is undecidable when $\mathcal{L}_I = CQ$.

– $(\mathcal{L}_T, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services. This class is used to study the interaction between the guards language and the update language.

The analysis of the aforementioned classes is presented in the subsequent sections.

3.3 Guarded services $(\mathcal{L}_T, \emptyset, \emptyset)$

We study in this section the impact of the language $\mathcal{L}_T$ on the decidability of the simulation for guarded services. For this purpose, we consider the sub-class $(\mathcal{L}_T, \emptyset, \emptyset)$ of *Update-free* services having transitions guarded with boolean queries expressed in the language $\mathcal{L}_T$.

Example 31. Figure 3.4(a) depicts an example of two $(\mathcal{L}_T, \emptyset, \emptyset)$ services over the same *global* database R , where the guards are boolean conjunctive queries (figure 3.4(b)). The service S_1 sends $m_1()$ either if the instance contains a tuple (guard q_1) or if it contains a tuple with the same value for the two attributes (guard q_2).

3.3.1 Characterization of simulation for guarded services

We will prove that, the simulation of $(\mathcal{L}_T, \emptyset, \emptyset)$ services is decidable iff checking the satisfiability of formula in $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is decidable, where $\mathcal{L}_T$ is a boolean query language.

From definition of simulation (c.f., definition 17), one can expect two sources of difficulties to test simulation between two guarded services S and S' :

1. the problem of testing whether $id \preceq id'$ with id (respectively, id') an instantaneous description of S (respectively, S'). We refer to this test as simID.
2. the possibly infinite number of simID tests required to check whether $S \preceq S'$.

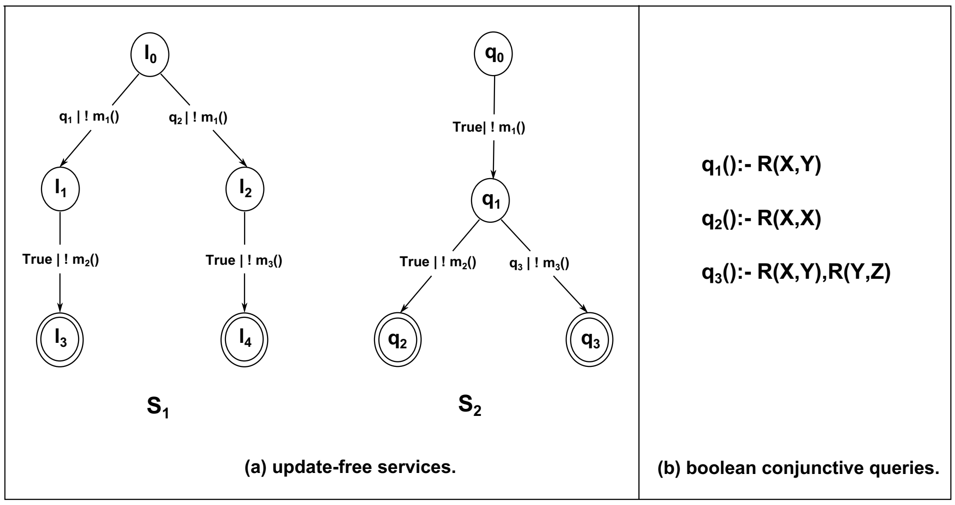


FIGURE 3.4 – $(\mathcal{L}_T, \emptyset, \emptyset)$ services S_1, S_2 and their corresponding guards.

The test simID is clearly decidable since every (eventually infinite) execution tree $tree(id_0)$ of an $(\mathcal{L}_T, \emptyset, \emptyset)$ service S can be represented by a finite state machine $FSM_{id_0}(S)$. Hence, checking simulation between $FSM_{id_0}(S)$ and $FSM_{id_0}(S')$ can be reduced to a simulation test between two finite state machines.

Definition 18. *Let $S = \langle \mathcal{W}, L, l_0, F, \delta, \Sigma \rangle$ be a $(\mathcal{L}_T, \emptyset, \emptyset)$ service. Let I be an instance over $\mathcal{W}_g$. We denote by $FSM_I(S) = \langle \mathcal{W}, L, l_0, F, \delta_I, \Sigma_I \rangle$ a finite state machine such that :*
(i) $\delta_I = \{(l, a_m, l') \mid \exists (l, q, !m(), l') \in \delta$ and $q(I) = \mathsf{true}\}$.
(ii) $\forall l \in \delta_I$, there is a path from l_0 to a final state through l.

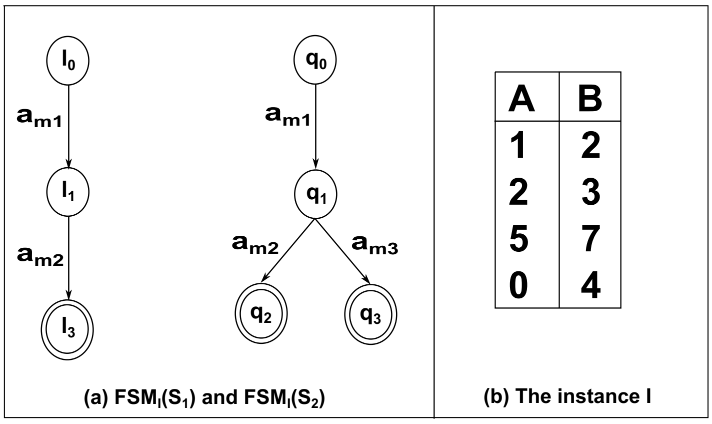
Note that, each message $!m()$ of S is renamed in the state machine $FSM_I(S)$ to a string a_m. Σ_I represents the alphabet obtained by renaming the messages. The point (ii) ensures that each state of $FSM_I(S)$ is reachable from the initial state and reaches a final state. We denote by $tree(FSM_I(S))$ the possibly infinite execution tree of $FSM_I(S)$.

Hence, to define a simulation algorithm for the class $(\mathcal{L}_T, \emptyset, \emptyset)$ it remains to cope with the problem 2, i.e., to handle an infinite number of simID tests. Let S and S' be two $(\mathcal{L}_T, \emptyset, \emptyset)$ services defined over the same global database schema $\mathcal{W}_g$. To cope with problem 2, the main idea is to partition the infinite set of instances over $\mathcal{W}_g$ into a finite set of partitions such that :

(i) the number of partitions is finite (but a given partition may represent an infinite number of instances over $\mathcal{W}_g$),
(ii) the simulation between S and S' can be reduced to a set of simulation tests between partitions, and
(iii) the simulation test between two partitions is decidable, since it can be recast to test of simulation between finite state machines.

Example 32. Figure 3.5(a) depicts the $FSM_I(S_1)$ and the $FSM_I(S_2)$, (S_1 and S_2 are depicted at figure 3.4). According to definition 18 : since $I \not\models q_2$, there is no transition from l_0 to l_2, hence there is no *valid* path from l_0 to l_4. On another hand, since $I \models q_1$, we have a path from l_0 to l_3. The messages $!m_1()$ and $!m_2()$ are renamed as a_{m_1} and a_{m_2} respectively. Note that, $FSM_I(S_1) \preceq FSM_I(S_2)$.

Lemma 14. *Let $S = \langle \mathcal{W}, L, l_0, F, \delta, \Sigma \rangle$ be a $(\mathcal{L}_T, \emptyset, \emptyset)$ service and $id_0 = (l_0, I, \emptyset)$ an initial configuration of S, with I an arbitrary instance over $\mathcal{W}_g$, then $tree(id_0) \cong tree(FSM_I(S))$.*



A	B
1	2
2	3
5	7
0	4

Figure 3.5 – $FSM_I(S_1)$, $FSM_I(S_2)$.

Démonstration. The lemma is a direct consequence of the construction of $FSM_I(S)$ (c.f., definition 18). □

As a direct consequence of lemma 14, simulation between two $(\mathcal{L}_T, \emptyset, \emptyset)$ services S and S' can be rephrased as follows :

Lemma 15. *Let S and S' be two $(\mathcal{L}_T, \emptyset, \emptyset)$ services over the same global schema $\mathcal{W}_g$, then $S \preceq S'$ iff for every instance I over $\mathcal{W}_g$, we have $FSM_I(S) \preceq FSM_I(S')$*

Definition 19. *Let G be a set of $\mathcal{L}_T$ boolean queries over a database schema $\mathcal{W}_g$ and let $g \subseteq G$. Let $\mathcal{I}_\mathcal{W}$ be the infinite set of all the possible instances over $\mathcal{W}_g$. We denote by q_G^g the formula obtained by the conjunction of the queries in g and the negation of the queries of G not in g, i.e., : $q_G^g := (\bigwedge_{q \in g} q) \wedge (\bigwedge_{q \in G \setminus g} \neg q)$*

Example 33. Let G be the set of boolean queries of the service S_1 depicted at figure 3.4 :
- If g=$\{q_1\}$ then $q_G^g()$= $R(X_1, Y_1) \wedge \neg(R(X_2, X_2))$.
- If g=$\{q_2\}$ then $q_G^g()$= $R(X_1, X_1) \wedge \neg(R(X_2, Y_2))$, which is unsatisfiable, because $q_2 \subseteq q_1$.

Given such a formula q_G^g, we define the following associated sets :
– $P_g(\mathcal{I}_\mathcal{W}) = \{I \in \mathcal{I}_\mathcal{W} \mid q_G^g(I) = \texttt{true}\}$, this set denotes the set of instances of $\mathcal{W}$ which returns a positive answer to the boolean query q_G^g, and
– $\mathcal{P}_G = \{P_g(\mathcal{I}_\mathcal{W}) \mid g \in 2^G\}$, where 2^G denotes the powerset of G. The set $\mathcal{P}_G$ forms a partition of $\mathcal{I}_\mathcal{W}$ since :
  – $\forall g, g' \in 2^G$, with $g \neq g'$, we have $q_G^g \wedge q_G^{g'}$ is unsatisfiable (and hence $P_g(\mathcal{I}_\mathcal{W}) \cap P_{g'}(\mathcal{I}_\mathcal{W}) = \emptyset$), and
  – the query $\bigvee_{g \in 2^G} q_G^g$ is valid (and hence $\bigcup_{g \in 2^G} P_g(\mathcal{I}_\mathcal{W}) = \mathcal{I}_\mathcal{W}$).

Lemma 16. *Let S be a $(\mathcal{L}_T, \emptyset, \emptyset)$ service over the schema $\mathcal{W}_g$ and let G^S be the set of boolean queries used as guards in S. Then, $\forall g \subseteq G^S$, we have $\forall I, I' \in P_g(\mathcal{I}_\mathcal{W}), FSM_I(S) \cong FSM_{I'}(S)$*

Démonstration. The lemma is a direct consequence of the definition of $FSM_I(S)$ and the definition of a partition (c.f., definition 19 and 18). □

In the sequel, given a partition $P_g(\mathcal{I}_\mathcal{W})$ that satisfies the conditions of lemma 16, we denote by $FSM_{P_g(\mathcal{I}_\mathcal{W})}(S)$ the FSM representing, modulo simulation equivalence, the finite state machines of the instances of $\mathcal{I}_\mathcal{W}$ contained in the partition $P_g(\mathcal{I}_\mathcal{W})$. $FSM_{P_g(\mathcal{I}_\mathcal{W})}(S)$

is obtained by constructing the finite state machine of an arbitrary database that belong to the partition $P_g(\mathcal{I}_{\mathcal{W}})$.

Lemma 17. *Let S and S' be two $(\mathcal{L}_T, \emptyset, \emptyset)$ services over the same schema $\mathcal{W}_g$ and let G be the set of boolean queries used as guards in S or S'. Then : $S \preceq S'$ iff $\forall P_g(\mathcal{I}_{\mathcal{W}}) \in \mathcal{P}_G$, such that $P_g(\mathcal{I}_{\mathcal{W}})$ is not empty, we have $FSM_{P_g(\mathcal{I}_{\mathcal{W}})}(S) \preceq FSM_{P_g(\mathcal{I}_{\mathcal{W}})}(S')$*

Démonstration. The lemma is a direct consequence of lemma 15 and lemma 16. □

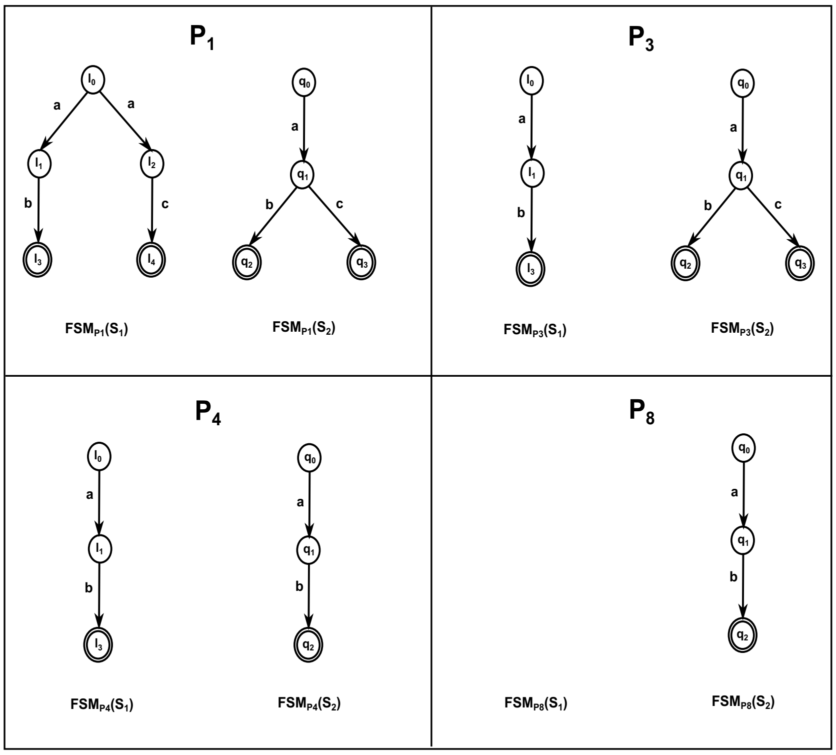


Figure 3.6 – The set of satisfiable partitions and their associated $FSM_P(S_1)$, $FSM_P(S_2)$.

Example 34. Let S_1, S_2 the two services depicted at figure 3.4, then G=$\{q_1, q_2, q_3\}$. The set $\mathcal{P}_G$ contains the following elements :

- P_1=$q_1 \wedge q_2 \wedge q_3$.
- P_2=$q_1 \wedge q_2 \wedge \neg q_3$. P_2 is unsatisfiable because $q_2 \subseteq q_3$
- P_3=$q_1 \wedge q_3 \wedge \neg q_2$.
- P_4=$q_1 \wedge \neg q_2 \wedge \neg q_3$.
- P_5=$q_2 \wedge q_3 \wedge \neg q_1$. P_5 is unsatisfiable because $q_2 \subseteq q_1$ and $q_3 \subseteq q_1$.
- P_6=$q_2 \wedge \neg q_3 \wedge \neg q_1$. P_6 is unsatisfiable because $q_2 \subseteq q_3$
- P_7=$q_3 \wedge \neg q_1 \wedge \neg q_2$. P_7 is unsatisfiable because $q_3 \subseteq q_1$
- P_8=$\neg q_3 \wedge \neg q_1 \wedge \neg q_2$. P_8 is satisfiable by the empty instance.

Figure 3.6 depicts the set of satisfiable partitions $\{P_1, P_3, P_4, P_8\}$ and their corresponding FSM for S_1 and S_2. From lemma 17 we can conclude that $S_1 \preceq S_2$.

Lemma 17 asserts that, the simulation between two guarded services is decidable if checking the satisfiability of the query associated to a partition is decidable (i.e., the partition is not empty). The query associated to a partition is a formula expressed in the language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$. To provide a full characterization of simulation in this context, we shall prove now that simulation in $(\mathcal{L}_T, \emptyset, \emptyset)$ is undecidable if checking the satisfiability of a formula expressed in the language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is undecidable.

Lemma 18. *Let P be a formula expressed in the language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$. Then, there exists two $(\mathcal{L}_T, \emptyset, \emptyset)$ services S_1 and S_2 such that the formula P is satisfiable iff $S_1 \not\preceq S_2$*

Démonstration. The proof is based on the observation that the test of satisfiability of the formula P can be reduced to a test of simulation between two $(\mathcal{L}_T, \emptyset, \emptyset)$ services. P is of the form $q_1() \wedge q_2() ... \wedge q_i() \wedge \neg q_{i+1}() \wedge .... \neg q_n()$ where each q_j where $j \in [1, n]$ is a boolean query expressed in the language $\mathcal{L}_T$. For $k \in [1, i]$, q_k is a positive boolean query of the form $q_k()$:-$body_k$. We construct the boolean query $q_{pos} = \bigwedge_{k \in [1,i]} body_k$. Note that, the obtained query q_{pos} is a boolean query expressed in the language $\mathcal{L}_T$. The figure 3.7 depicts the obtained services S_1 and S_2. The service S_1 contains one transition guarded by the query q_{pos} and sends the message $m()$. The service S_2 contains n-i transitions (i.e. the number of negated queries), where each transition is guarded by a query q_k with $k \in [i+1, n]$ and sends the message $m()$. Hence $S_1 \not\preceq S_2$ iff there exists an instance I such that $I \models q_{pos}$ and for each $k \in [i+1, n]$ $I \not\models q_k$. This is the case if the formula P is satisfiable. Hence, simulation in $(\mathcal{L}_T, \emptyset, \emptyset)$ services is undecidable if satisfiability in $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is undecidable. □

Hence, from lemma 17 and lemma 18, we can derive the following theorem :

Theorem 8. *Simulation of $(\mathcal{L}_T, \emptyset, \emptyset)$ services is decidable iff checking the satisfiability of formula in $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is decidable.*

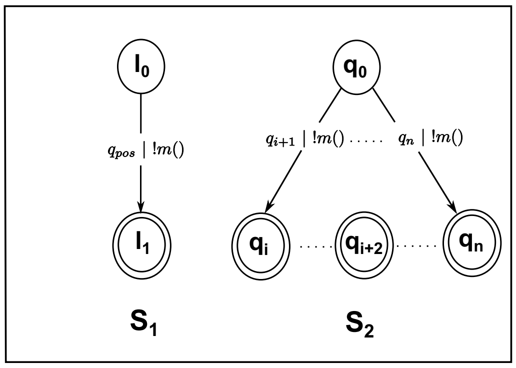


Figure 3.7 – Connection between simulation and the language $\mathcal{L}_T$.

In the following, we study the complexity of simulation for guarded services when $\mathcal{L}_T$ is propositional logic, noted PL. we will prove that, for this case the complexity of simulation is ranged between co-np-hard and Π_2^p where Π_2^p represents the set of decision problems solvable by a non deterministic Turing machine augmented with an oracle for some co-np-complete problems.

Problem 3. co-sim $(\mathcal{L}_{PL}, \emptyset, \emptyset)$
Inputs : two $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ services S and S'.
Question : $S \not\preceq S'$?

Proposition 3. co-sim $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ is in Σ_2^p.

Démonstration. Let S and S' be two $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ services and let G be the set of propositional logic queries (i.e, boolean queries containing only constants) used as guards in S and S'. Given a partition $P \in P_G$ and an oracle for checking the satisfiability of partitions of P_G, it is possible to check in polynomial time whether S is not simulated by S'. Indeed, it is sufficient to check the consistency of the partition P and then check the simulation between the two finite state machines $FSM_P(S)$ and $FSM_P(S')$. Since satisfiability of a propositional logic formula is NP-complete, CO-SIM $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ is in Σ_2^p. □

We shall prove now the NP-hardness of the problem CO-SIM in the case of $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ services using a reduction from 3-SAT problem [Coo71]. The 3-SAT problem is stated as following : given n boolean variables $\{x_1, ..., x_n\}$, 3-SAT problem is the problem of testing the satisfiabilty of a formula composed of a conjunction of clauses where each clause contains a disjunction of exactly three boolean (or it negation) variables (called also a literal). Given a 3-SAT problem, we construct two services $S_{3SAT-spoiler}$ and $S_{3SAT-duplicator}$ and reduce 3-SAT to a simulation test between these services.

Lemma 19. *Given a* 3-SAT *problem instance with* n *boolean variables, the problem has a solution iff* $S_{3SAT-spoiler} \not\preceq S_{3SAT-duplicator}$*, where* $S_{3SAT-spoiler}$ *and* $S_{3SAT-duplicator}$ *are* $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ *services.*

Démonstration. Let the formula φ be a 3-SAT problem instance with n variables. The idea of the proof is that, $S_{3SAT-spoiler}$ will have a transition guarded with φ and the action will be the send of the message $m()$. If there exists a database instance I over $\mathcal{W}_g$ such that φ is true then $S_{3SAT-spoiler}$ can make an action which cannot be simulated by $S_{3SAT-duplicator}$, hence there is no simulation. Now we will detail the construction.
$\mathcal{W}_g$ contains n boolean relational schema $\{R_1, ..., R_n\}$. An instance I over $\mathcal{W}_g$ corresponds to a set of instance $\{I_{R_1}, ..., I_{R_n}\}$. The relations R does not have any attribute, and there are only two possible instances : one containing the empty tuple, then we say that R is evaluated to true, the other instance is empty then we say that R is evaluated to false.
$S_{3SAT-spoiler}$ contains only one transition $(l_0, q_\varphi, !m(), l_1)$ and $S_{3SAT-duplicator}$ contains one state u_0 without any transition. A literal x_i (or its negation $\neg x_i$) in φ is transformed to $R_i()$ $(\neg R_i())$ in q_φ. If there exists an instance $I \models \varphi$ (hence, 3-SAT has a solution), then $S_{3SAT-spoiler}$ sends the message $m()$ while $S_{3SAT-duplicator}$ cannot reproduce this action. Hence, $S_{3SAT-spoiler} \not\preceq S_{3SAT-duplicator}$. If 3-SAT does not accept any solution, there is no database instance I which satisfies the guard q_φ, hence there is simulation. The figure 3.8 depicts the test of simulation between $S_{3SAT-spoiler}$ and $S_{3SAT-duplicator}$.

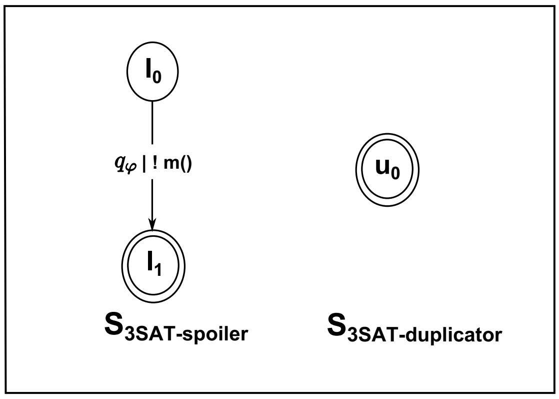


FIGURE 3.8 – Reduction from 3-SAT problem to CO-SIM $(\mathcal{L}_{PL}, \emptyset, \emptyset)$.

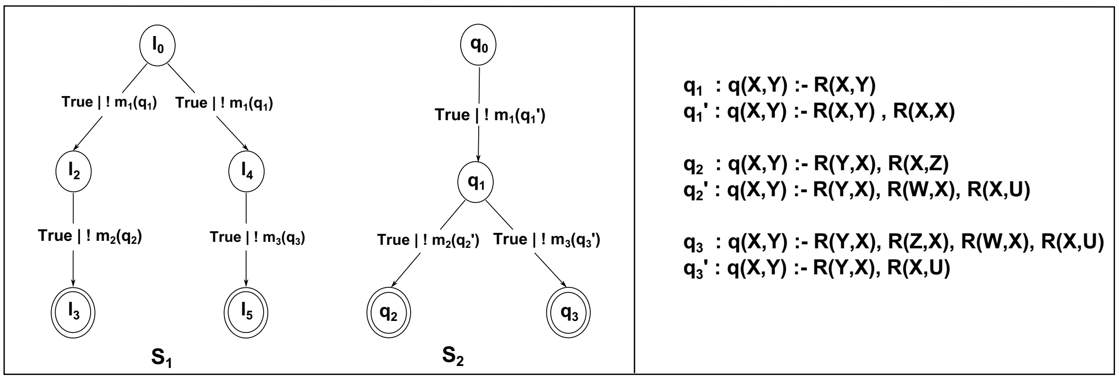
□

Theorem 9. CO-SIM *$(\mathcal{L}_{PL}, \emptyset, \emptyset)$ is* NP-HARD.

Démonstration. From lemma 19 and knowing that 3-SAT problem is NP-HARD [Coo71]. □

Hence, from proposition 3 and theorem 9 we can state that simulation in $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ is ranged between CO-NP-HARD and Π_2^p.

3.4 Send services $(\emptyset, \mathcal{L}_S, \emptyset)$



l0
True | ! m1(q1)
True | ! m1(q1)
l2
l4
True | ! m2(q2)
True | ! m3(q3)
l3
l5
S1
q0
True | ! m1(q1')
q1
True | ! m2(q2')
True | ! m3(q3')
q2
q3
S2
q1 : q(X,Y) :- R(X,Y)
q1' : q(X,Y) :- R(X,Y) , R(X,X)
q2 : q(X,Y) :- R(Y,X), R(X,Z)
q2' : q(X,Y) :- R(Y,X), R(W,X), R(X,U)
q3 : q(X,Y) :- R(Y,X), R(Z,X), R(W,X), R(X,U)
q3' : q(X,Y) :- R(Y,X), R(X,U)

FIGURE 3.9 – $(\emptyset, \mathcal{L}_S, \emptyset)$ services S_1 and S_2.

An $(\emptyset, \mathcal{L}_S, \emptyset)$ service denotes an *unguarded* service that is only able to send messages. The content of a message sent correspond to the result of the associated query when executed over the current global database. Note that, different queries expressed in $\mathcal{L}_S$ can be associated to the same message. As an example, figure 3.9 depicts two service S_1 and S_2 defined over the same schema $\mathcal{W}_g$ where q_1 and $q_1^{'}$ are associated to the same message m_1 respectively in S_1, S_2.

Undecidability of simulation between send services We will show the connection between the decidability of checking the simulation for $(\emptyset, \mathcal{L}_S, \emptyset)$ services and the query language $\mathcal{L}_S$ used to send the messages. The next theorem states that, simulation in $(\emptyset, \mathcal{L}_S, \emptyset)$ services is undecidable if satisfiability of formula expressed in the language $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable, where $\mathcal{L}_S$ is a boolean query language. The proof is similar to the proof of lemma 18. It is based on a reduction from the problem of testing satisfiablity of a formula to the problem of checking simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services (the proof is given in appendix A).

Theorem 10. *Simulation in $(\emptyset, \mathcal{L}_S, \emptyset)$ services is undecidable if satisfiability of formula in $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable, where $\mathcal{L}_S$ is a boolean query language.*

3.4.1 Decidability of simulation between send services.

In this section, we will present a framework that enables to check the simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services. As for the previous class, we will use a partitioning approach to reduce the test of simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services to a set of tests between finite state machines.

Example 35. Figure 3.10 depicts two $(\emptyset, \mathcal{L}_S, \emptyset)$ services. S_1 sends the result of the query q_1. S_2 sends either the result of q_2 or q_3. The language $\mathcal{L}_S$= CQ. Hence, $S_1 \preceq S_2$ iff $\forall$ I $q_1(I)$=$q_2(I)$ or $q_1(I)$=$q_3(I)$. Observe that, the test of simulation cannot be reduced to a test of equivalence between union of conjunctive queries. In fact in this example there is simulation but $q_1 \not\equiv q_2 \cup q_3$

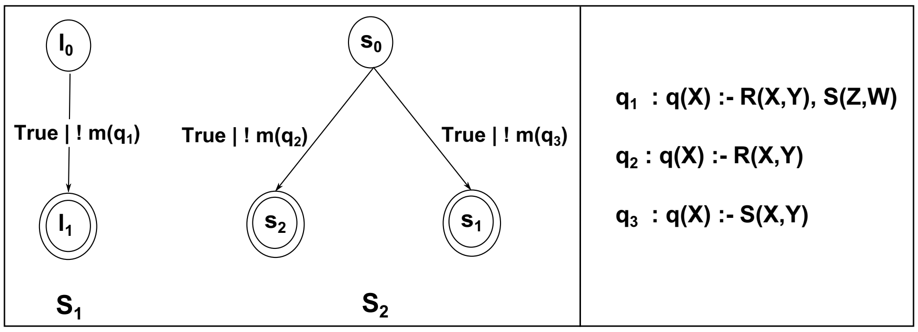


FIGURE 3.10 – two $(\emptyset, \mathcal{L}_S, \emptyset)$ services S_1 and S_2.

Let S_1 and S_2 be two $(\emptyset, \mathcal{L}_S, \emptyset)$ services. Every execution tree $tree(id_0)(S_1)$ (resp. $tree(id_0^{'})(S_2)$) of service S_1 (resp. S_2) can be represented by a finite state machine $FSM_I(S_1)$ ($FSM_I(S_2)$) where the transitions are labelled with the answers of queries on I. Then, checking the simulation between $FSM_I(S_1)$ and $FSM_I(S_2)$ is equivalent to check the equality between answers of queries for the same database. Note that, this transformation is applicable because there is no modification of the database during the execution of the service.

Definition 20. *Let $S = \langle \mathcal{W}, L, l_0, F, \delta, \Sigma \rangle$ be a $(\emptyset, \mathcal{L}_S, \emptyset)$ service. Let I be an instance over $\mathcal{W}_g$. We denote by $FSM_I(S) = \langle \mathcal{W}, L, l_0, F, \delta_I, \Sigma_I \rangle$ a finite state machine such that :*

(i) $\delta_I = \{(l, q(I), l') \mid \exists (l, True, !m(q), l')\}$. δ_I is a transition relation where we replace the message m(q) by the answers of q on I.

(ii) $\Sigma_I = \{q(I) \mid$ q is a query appearing in S $\}$

Lemma 20. *Let $S = \langle \mathcal{W}, L, l_0, F, \delta, \Sigma \rangle$ be a $(\emptyset, \mathcal{L}_S, \emptyset)$ service and $id_0 = (l_0, I, \emptyset)$ an initial configuration of S, with I an arbitrary instance over $\mathcal{W}_g$, then $tree(id_0) \cong tree(FSM_I(S))$.*

Démonstration. The lemma follows the construction of $FSM_I(S)$ (c.f., definition 20). □

As a direct consequence of lemma 20, simulation between two send services S and S' can be rephrased as follows :

Lemma 21. *Let S and S' be two $(\emptyset, \mathcal{L}_S, \emptyset)$ services over the same schema $\mathcal{W}_g$, then : $S \preceq S'$ iff for every instance I over $\mathcal{W}_g$, we have $FSM_I(S) \preceq FSM_I(S')$.*

The number of instances over $\mathcal{W}_g$ is infinite. Hence, the number of finite state machines $FSM_I(S)'s$ is infinite. To handle this problem, we provide below an abstraction technique. This abstraction framework allows to regroup the infinite number of finite state machines into a finite set of FSM.

Let $\mathcal{Q}$ be the a set of queries and $p = \{b_1, b_2, ..., b_n\}$ be a partition of $\mathcal{Q}$, then each b_i is a subset of $\mathcal{Q}$ and all b_i are pairwise disjoint.

Definition 21. *Let S_1 and S_2 be two $(\emptyset, \mathcal{L}_S, \emptyset)$ services over the same global database schema $\mathcal{W}_g$. Let $\mathcal{Q}$ be the set of queries appearing in the two services and p be a partition of $\mathcal{Q}$ then :*
$\mathcal{I}_{\mathcal{W}}(p) = \{I \in \mathcal{I}_{\mathcal{W}} \mid \forall q_j, q_k \in b_i$ then $q_j(I) = q_k(I)$ and $\forall q_j \in b_i$ and $q_k \in b_l$ then $q_j(I) \neq q_k(I)$ where $i \neq l$ and $i, l \in [1, n]\}$

We call the subset b of a partition a *bucket*. An instance I of $\mathcal{W}_g$ belongs to a partition p of $\mathcal{Q}$ if and only if the answers of queries on I in the same bucket are equal and the answers of queries on I from different buckets are different.
We denote by $\mathcal{P}_\mathcal{Q}$ the set of partitions of the set $\mathcal{Q}$. Note that, an instance cannot be in two different partitions and a partition may be empty.

We associate to each partition a formula such that the formula is unsatisfiable iff the partition is empty.

Definition 22. *Let $p = \{b_1, b_2, ..., b_n\}$ be a partition. The formula f_p associated to p is constructed as follows :*
$f_p = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{|b_i|} (q_{i1} = q_{ij}) \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} (q_{i1} = q_{j1})$

Example 36. Continuing with the example depicted at figure 3.10, the set of partitions $\mathcal{P}_\mathcal{Q}$ is :
– $p_1 = \{(q_1), (q_2), (q_3)\}$
– $p_2 = \{(q_1, q_2), (q_3)\}$
– $p_3 = \{(q_1, q_3), (q_2)\}$
– $p_4 = \{(q_2, q_3), (q_1)\}$
– $p_5 = \{(q_1, q_2, q_3)\}$
Each instance I belonging to p_2 satisfies the following formula f_{p_2} : $(q_1(I) = q_2(I)) \wedge (q_1(I) \neq q_3(I))$.

Definition 23. *Let $S_1 = \langle \mathcal{W}_g, L^1, l_0^1, F^1, \delta^1, \Sigma^1 \rangle$ and $S_2 = \langle \mathcal{W}_g, L^2, l_0^2, F^2, \delta^2, \Sigma^2 \rangle$ be two $(\emptyset, \mathcal{L}_S, \emptyset)$ services. Let $\mathcal{Q}$ be the set of queries appearing in the two services and $p = \{b_1, b_2, ..., b_n\}$ be a partition of $\mathcal{Q}$. Then $FSM_p(S_1) = \langle L^1, l_0^1, F^1, \delta_p^1, \Sigma_p \rangle$ is a finite state machine such that :*
– $\delta_p^1 = \{(l, b_i, l') \mid \exists (l, True, !m(q_j), l') \in \delta^1$ *and* $q_j \in b_i\}$ *and*
– $\Sigma_p = \{b_1, b_2, ..., b_n\}$.

Because answers of queries in the same bucket are equal for a fixed instance, we can reduce the test of simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services to a set of simulation tests between finite state machines representing the partitions.

Lemma 22. *Let S be a $(\emptyset, \mathcal{L}_S, \emptyset)$ service and p a partition of the set of query $\mathcal{Q}$. Then for each database instance I over $\mathcal{W}_g$ such that $I \in p$, we have $FSM_I(S) \cong FSM_p(S)$.*

The previous lemma is a direct consequence of the definition of FSM_p (c.f., definition 23). It asserts that the finite state machines of the instances belonging to a same partition are all simulation equivalent to the finite state machine of the partition.
Because the number of queries in the service is finite, the number of partitions is also finite. Hence, we can reduce the test of simulation between two send services to a set of simulation tests between finite state machines.

Example 37. The figure 3.11 depicts the set of tests of simulation for S_1 and S_2 (from example 35). The partitions p_1 and p_4 are not satisfiable.

Lemma 23. *Let S and S' be two $(\emptyset, \mathcal{L}_S, \emptyset)$ services over the schema $\mathcal{W}_g$ and let $\mathcal{P}_\mathcal{Q}$ be the set of queries used in S or S'. Then, $S \preceq S'$ iff $\forall p \in \mathcal{P}_\mathcal{Q}$ such that p is not empty, we have $FSM_p(S) \preceq FSM_p(S')$*

Démonstration. This lemma is a direct consequence of lemma 21 and lemma 22. □

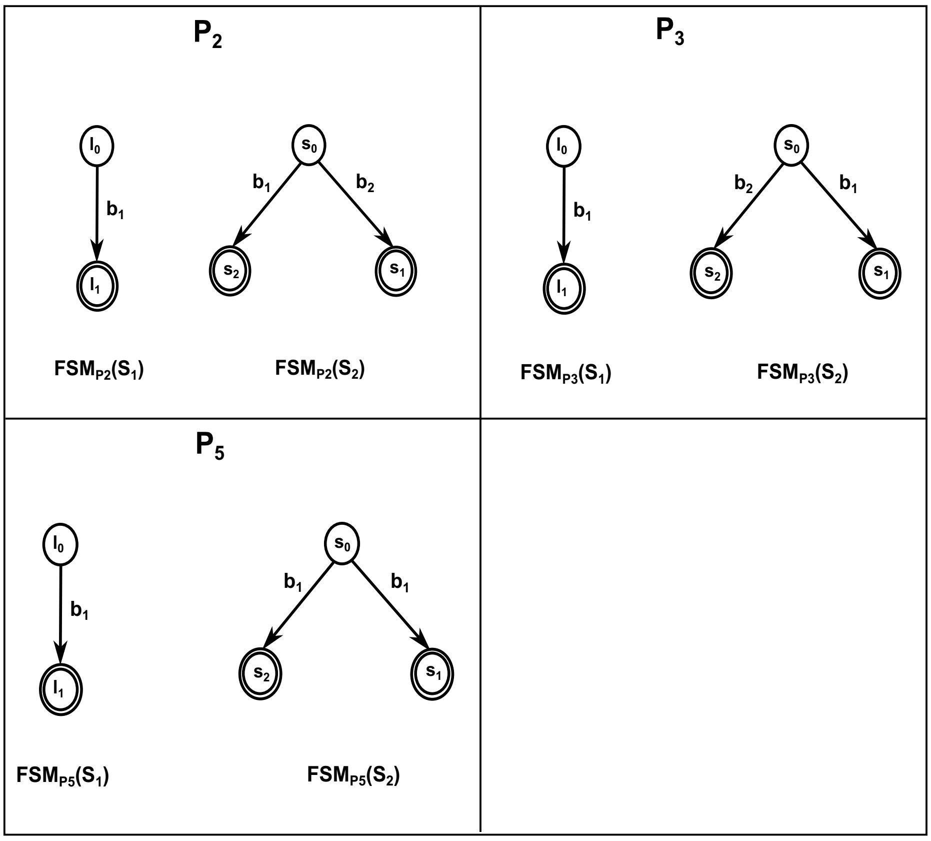


Figure 3.11 – set of tests of simulation.

Now, we can derive the following theorem :

Theorem 11. *Checking the simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services is decidable if checking the satisfiablity of formula f_p associated to a partition as constructed above is decidable.*

Démonstration. from lemma 23. □

Unfortunately, theorem 10 and theorem 11 do not provide a full characterization of simulation between send services. This is due to the fact that, in current state of affairs, we are not able to reduce the test of satisfiability of a formula associated to a partition to a test of simulation between two send services.

The next subsection is devoted to the complexity of simulation between send services when $\mathcal{L}_S$ is the propositional logic. We prove that, for this case the problem of simulation is ranged between co-NP-HARD and Π_2^p. Because the proofs of the next proposition and theorem are respectively similar to the proofs of proposition 3 and theorem 9, we give them in appendix A.

Problem 4. **CO-SEND** $(\emptyset, \mathcal{L}_{LP}, \emptyset)$
Inputs : two $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ services S and S'. *Question* : $S \not\preceq S'$?

Proposition 4. the problem CO-SEND $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ is in Σ_2^p

Theorem 12. CO-SEND *$(\emptyset, \mathcal{L}_{LP}, \emptyset)$ is* NP-HARD

Hence, from proposition 4 and theorem 12 we can state that simulation in $(\emptyset, \mathcal{L}_{PL}, \emptyset)$ is ranged between co-NP-HARD and Π_2^p.

3.5 Insert services $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$

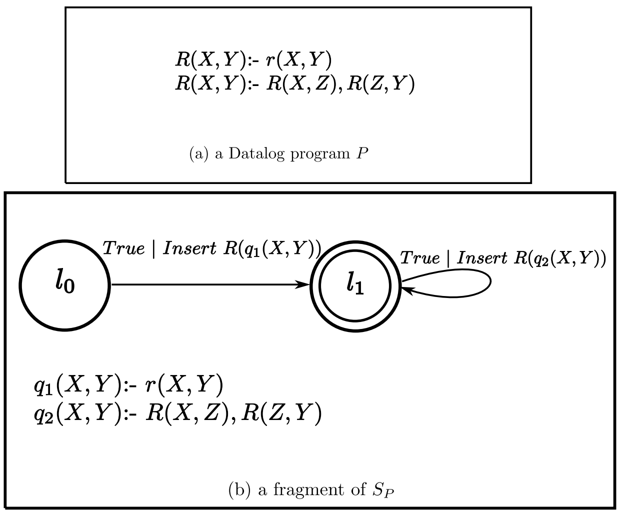
In this section we study the simulation for generic services without guards. The considered services are able to insert data in the global database. In this context, we study the simulation as well as the *weak* simulation relation.

The next theorem shows the connection between satisfiability of the insert language $\mathcal{L}_I$ and simulation.

Theorem 13. *Simulation in* $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ *services is undecidable if checking satisfiability of formulas in* $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ *is undecidable.*

The proof is given in appendix A.
From theorem 13, one can expect to characterize decidability of simulation by establishing a correspondence with decidability of satisfiability of formulas in $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$. Next, we will show that this is not true. In fact, there exists a language $\mathcal{L}_I = GNCQ$ such that satisfiability in $GNCQ \cup \{\wedge^f, \neg^f\}$ is decidable while simulation in $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ)})$ is undecidable. The *Guarded Negation Conjunctive Query* ($GNCQ$) language is included in *Guarded Negation First Order* language $GNFO$ [BtCS11]. $GNCQ$ queries are conjunctive queries with guarded negations (i.e., all free variables appearing in a negative atom must appear in a positive atom). Next, we will give the proof of undecidability of simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ)})$ services. The proof is by reduction from the problem of containment between two Datalog programs [Shm93].

3.5.1 Undecidability of simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ)})$ services


$R(X,Y)$:- $r(X,Y)$
$R(X,Y)$:- $R(X,Z), R(Z,Y)$

(a) a Datalog program P

$q_1(X,Y)$:- $r(X,Y)$
$q_2(X,Y)$:- $R(X,Z), R(Z,Y)$

(b) a fragment of S_P
l_0
$True \mid Insert\ R(q_1(X,Y))$
l_1
$True \mid Insert\ R(q_2(X,Y))$

Figure 3.12 – A Datalog program and its corresponding $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(CQ)})$ service.

In this section we prove the undecidability of simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$ services by a reduction from the containment problem of *Datalog* programs [Shm93]. A *Datalog* program can be simulated by an insert service using CQ as insertion language.
The figure 3.12(a) shows a simple Datalog program P with a query predicate R that computes the transitive closure of a relation r. A fragment of the corresponding service S_P is shown at figure 3.12 (b). The service starts by copying the relation r in R, then moves to the state l_1 and calculates the transitive closure. Hence, it is easy to check that any

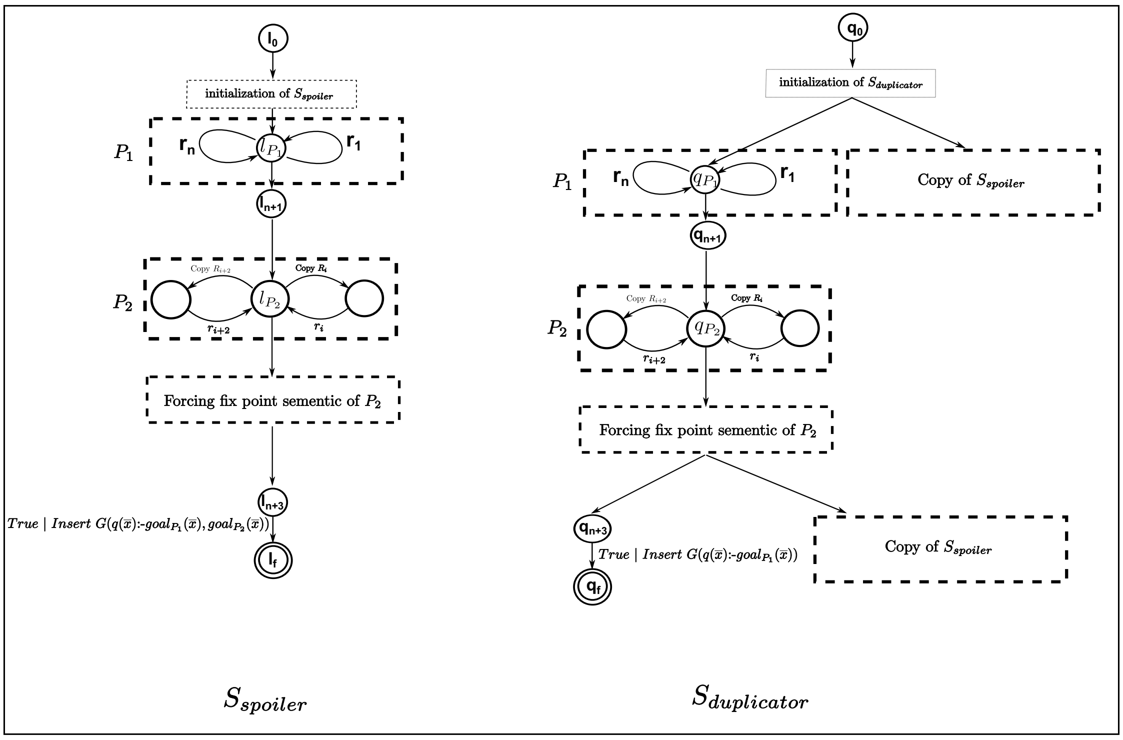
answer computed by P can also be computed by S_P. There are two difficulties to ensure that any answer computed by P can also be computed by S_P :

– all the IDB (i.e., intentional predicates, which appear in the head of a rule of the program Datalog) must be initially empty.
– enforce fixpoint semantics of Datalog program in a given service. S_P can compute only partial answers of P and stops. This is because during an execution of the service S_P, each time the service reaches the final state l_1, S_P can decide either to terminate or to compute additional answers. Therefore, to reduce Datalog query containment to simulation, one have to deal with this problem.

Starting from a test of containment of two Datalog programs $P_1 \sqsubseteq P_2$, we construct a test of simulation between two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$ services $S_{spoiler}$ and $S_{duplicator}$ such that $P_1 \sqsubseteq P_2$ iff $S_{spoiler} \preceq S_{duplicator}$.

The figure 3.13 depicts the two services $S_{spoiler}$ and $S_{duplicator}$. $S_{spoiler}$ starts by executing the part of the service encoding the program P_1, then executes the part encoding the program P_2 and finally calculate the intersection between the goal relations of P_1 and P_2, respectively noted $goal_{P_1}$ and $goal_{P_2}$, and inserts the result in a relation G. The service $S_{duplicator}$ will also start by executing the part of service encoding P_1, then executes the part encoding the program P_2 but instead of calculating the intersection of the goals of P_1 and P_2, it copies the goal of P_1 in G. Clearly, $P_1 \sqsubseteq P_2$ iff all the instances of the relation G calculated by $S_{spoiler}$ is equal to the instance G calculated by $S_{duplicator}$. A rule of a Datalog program of the form $R(\overline{x})$:-$T_1(\overline{x_1}), ..., T_k(\overline{x_k})$, where T_j is an IDB or an EDB and the set of variables $\overline{x} \subseteq \bigcup_{i=1}^{k} \overline{x_i}$, is encoded with a transition labelled with INSERT R $(q_R(\overline{x})$:-$T_1(\overline{x_1}), ..., T_k(\overline{x_k}))$.

It should be noted that, at this step of construction, $S_{spoiler}$ can cheat to win the simu-



l_0
initialization of $S_{spoiler}$
P_1
r_n
l_{P_1}
r_1
l_{n+1}
Copy R_{i+2}
Copy R_i
P_2
l_{P_2}
r_{i+2}
r_i
Forcing fix point sementic of P_2
l_{n+3}
True | Insert $G(q(\overline{x})$:-$goal_{P_1}(\overline{x}), goal_{P_2}(\overline{x}))$
l_f
$S_{spoiler}$
q_0
initialization of $S_{duplicator}$
P_1
r_n
q_{P_1}
r_1
Copy of $S_{spoiler}$
q_{n+1}
Copy R_{i+2}
Copy R_i
P_2
q_{P_2}
r_{i+2}
r_i
Forcing fix point sementic of P_2
q_{n+3}
True | Insert $G(q(\overline{x})$:-$goal_{P_1}(\overline{x}))$
Copy of $S_{spoiler}$
q_f
$S_{duplicator}$

FIGURE 3.13 – reduction of containment of Datalog program to test of simulation between two $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ)})$ services.

lation game. This is because after executing some actions of P_1, $S_{spoiler}$ can decide to not execute the transitions of P_2. Hence when $S_{spoiler}$ calculates the intersection between the two goals, the result is always empty, while $S_{duplicator}$ will insert some answers of P_1 in G, hence there is no simulation. To handle this problem we have to force $S_{spoiler}$ to calculate the fix point (i.e., all the answers are calculated) of P_2. This is done by using a set of additional relations.

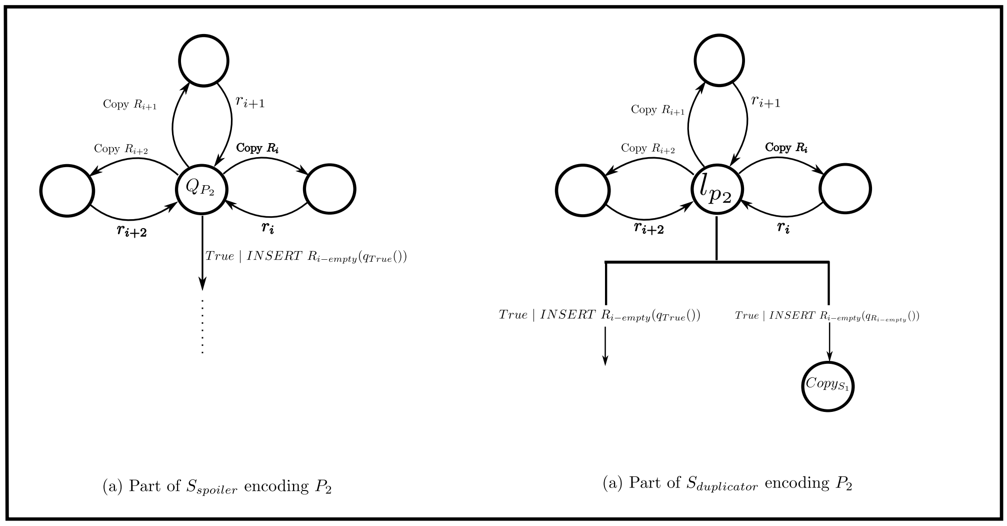

Copy R_{i+1}
r_{i+1}
Copy R_{i+2}
Copy R_i
Q_{P_2}
r_{i+2}
r_i
$True \mid INSERT\ R_{i-empty}(q_{True}())$
(a) Part of $S_{spoiler}$ encoding P_2
Copy R_{i+1}
r_{i+1}
Copy R_{i+2}
Copy R_i
l_{p_2}
r_{i+2}
r_i
$True \mid INSERT\ R_{i-empty}(q_{True}())$
$True \mid INSERT\ R_{i-empty}(q_{R_{i-empty}}())$
$Copy_{S_1}$
(a) Part of $S_{duplicator}$ encoding P_2

Figure 3.14 – forcing the spoiler to calculate the fix point of P_2.

The figure 3.14 depicts the part of services $S_{spoiler}$ and $S_{duplicator}$ encoding the execution of P_2. Assume that, P_2 contains n rules $r_1, ..., r_n$ which modify m IDB. We denote an IDB by R and we add m relation R_i^{copy} where $i \in m$. For each rule r of P_2, the services contain a cycle made of two transitions. The first transition copies the content of the head R in R^{copy}. Then, the second transition executes the rule r. We will also add m relation $R_{i\text{-empty}}$ where $i \in [1, m]$. After the part encoding P_2, for each IDB R of P_2, we add a transition in $S_{spoiler}$. This transition inserts $true$ into the relation $R_{i\text{-empty}}$ using the query q_{true} ($q_{true}()$:-). In the same time, $S_{duplicator}$ will contain two transitions, one execute the same transition as $S_{spoiler}$, the other one inserts $true$ in $R_{i\text{-empty}}$ if there exists a tuple in R_i which does not exist in R_i^{copy} (i.e. the transition inserts the result of the query $q_{R_{i\text{-empty}}}()$:-$R_i(\overline{x}) \wedge \neg R_i^{copy}(\overline{x})$) and reaches a state which simulated $S_{spoiler}$.

Assume that, during its execution, $S_{spoiler}$ does not calculate the fix point of P_2. Then there exists in some R_i a tuple which does not exist in R_i^{copy}. When $S_{spoiler}$ executes the transition leblled with q_{true}, the service inserts $true$ in $R_{i\text{-empty}}$. $S_{duplicator}$ can simulate this transition by executing the two transitions and reaches a state which simulates $S_{spoiler}$ (the transition on the right depicted at figure 3.14). Hence, $S_{spoiler}$ looses the simulation game. Assume now, $S_{spoiler}$ calculates the fix point of P_2, then for each transition labelled with q_{true} (for $i \in [1, m]$) the service inserts $true$ in the corresponding relation $R_{i\text{-empty}}$. In the same time, to maintain simulation $S_{duplicator}$ must execute the same transition labelled with q_{true}. if $S_{duplicator}$ chooses to execute the transition labelled with $q_{R_{i\text{-empty}}}$, it looses the simulation game (because there is no tuple in R_i^{copy} which is not in R_i, hence it inserts noting in $R_{i\text{-empty}}$).
With this construction, we ensure that, before calculating the intersection of the goals of

P_1 and P_2, $S_{spoiler}$ is simulated by $S_{duplicator}$ iff $S_{spoiler}$ calculate the fix point of P_2.

Now, we come back to the first problem : how to ensure that the IDBs and the additional relations added are initially empty. Assume that R is an IDB of P_1 or P_2. The figure 3.15 depicts the part of $S_{spoiler}$ and $S_{duplicator}$ which ensures that R starts empty. The service $S_{duplicator}$ insert $true$ into the relation $empty_R$ using the query q_{true}. $S_{duplicator}$ will have two transitions, one inserts $true$ in $empty_R$ iff R is not empty and reaches a state which simulates $S_{spoiler}$. The other one inserts $true$ in $empty_R$ using q_{true}(i.e., the same transition of $S_{spoiler}$). Assume that, R is not empty, when $S_{spoiler}$ executes the transition labbeled with q_{true}, $S_{duplicator}$ can simulates $S_{spoiler}$ by choosing the transition which inserts $true$ if R is not empty. Hence, $S_{spoiler}$ looses the simulation game. Now, assume that R is empty, $S_{spoiler}$ insert $true$ in $empty_R$, to maintain the simulation, $S_{duplicator}$ must execute the transition labelled with q_{true}, otherwise $S_{duplicator}$ looses the simulation.

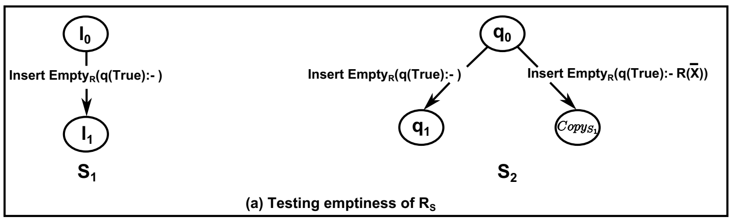

I0
Insert EmptyR(q(True):-)
I1
S1
q0
Insert EmptyR(q(True):-)
Insert EmptyR(q(True):- R(X̄))
q1
CopyS1
S2
(a) Testing emptiness of RS

FIGURE 3.15 – initialization part.

Now we can state the following lemma :

Lemma 24. *Let P_1 and P_2 be two Datalog programs. $P_1 \sqsubseteq P_2$ iff $S_{spoiler} \preceq S_{duplicator}$, where $S_{spoiler}$ and $S_{duplicator}$ are $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$ services constructed as described above.*

Hence, from lemma 24 and knowing that the problem of containment between two Datalog programs is undecidable [Shm93], we can derive the following theorem :

Theorem 14. *Checking simulation between two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$ services is undecidable.*

3.5.2 Undecidability of weak simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(CQ)})^p$ services

In this section we prove the undecidability of *weak* simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{insert(CQ)})^p$ by a reduction from the containment problem of *Datalog* programs [Shm93]. As we have seen before, a *Datalog* program can be simulated by an insert service using *CQ* as insertion language.
Like the proof of undecidability of simulation for insert services where $\mathcal{L}_I = GNCQ$, starting from a test of containment between two Datalog program $P_1 \sqsubseteq P_2$, we construct a test of weak simulation between two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(CQ)})^p$ services $S_{spoiler} \preceq_w S_{duplicator}$. The difference with the previous proof is that P_2 will only be executed by $S_{duplicator}$, where the service stores the answers in a local database.

Lemma 25. *Let P_1, P_2 be two Datalog programs then there exists two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(CQ)})^p$ services $S_{spoiler}$ and $S_{duplicator}$ such that $P_1 \sqsubseteq P_2$ iff $S_{spoiler} \preceq_w S_{duplicator}$.*

Démonstration. As depicted in figure 3.16, Service $S_{spoiler}$ has 3 steps : (i) the initialization part is used to verify the emptiness of all IDB of P_1 and an additional relation G (ii)

$S_{spoiler}$ executes P_1 (iii) finally, $S_{spoiler}$ copies the answers of P_1 into a new relation G. Service $S_{duplicator}$ executes the two first steps as $S_{spoiler}$, but then executes the program P_2, where the corresponding IDB of P_2 are *local* databases, and finally copies the intersection of answers of P_1 and P_2 into the relation G.
Assume that, all IDB and G are empty. $S_{spoiler}$ and $S_{duplicator}$ execute P_1, then $S_{spoiler}$ copies the answers of P_1 into the *global* database G while $S_{duplicator}$ can execute P_2 then copies in G the intersection of P_1 and P_2. There is a weak simulation if and only if the two services calculates the same *global* database G (which implies that $P_1 \sqsubseteq P_2$). $S_{duplicator}$ must calculate the fix point of P_2 to have a chance to win the simulation game. On the other hand, $S_{spoiler}$ must calculate the fix point of P_1 trying to find answers not calculated by P_2. Note that, $S_{spoiler}$ does not have a *local* database, and all transitions which involve P_2 are considered as *silent* transitions in $S_{duplicator}$.

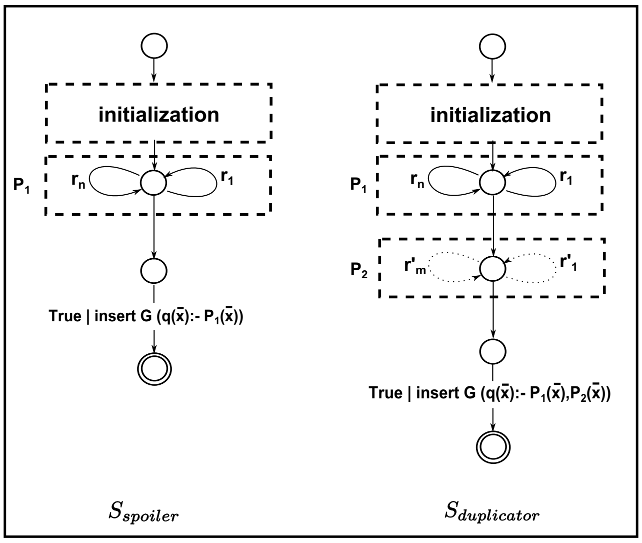

initialization
initialization
P₁
rₙ
r₁
P₂
r'ₘ
r'₁
True | insert G (q(x̄):- P₁(x̄))
True | insert G (q(x̄):- P₁(x̄),P₂(x̄))
$S_{spoiler}$
$S_{duplicator}$

FIGURE 3.16 – S_1 and S_2.

Now we come back to the initialization part. To ensure the emptiness of the *global* databases (G, and IDB of P_1) we use the same construction as for $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$. We face, however, on additional problems since we have to ensure that the IDB of P_2 start empty, knowing that they belong to the *local* databases. Figure 3.17 depicts the part of initialization which fix this problem. To achieve this goal, we introduce the following construction : the service $S_{spoiler}$ inserts noting in $empty_{P_2}$, while $S_{duplicator}$ inserts true in $empty_{P_2}$ if R_{P_2} is not empty (where $empty_{P_2}$ is a shared database, and R_{P_2} an IDB of P_2). Hence, $S_{duplicator}$ looses the simulation game if it starts with an IDB instance R_{P_2} which is not empty. This construction is repeated for all the IDBs of P_2.

□

From lemma 25 and knowing that the problem of containment between two Datalog programs is undecidable [Shm93], we can derive the following theorem :

Theorem 15. *Checking weak simulation for* $(\emptyset, \emptyset, \mathcal{L}_U^{insert(CQ)})^p$ *services is undecidable.*

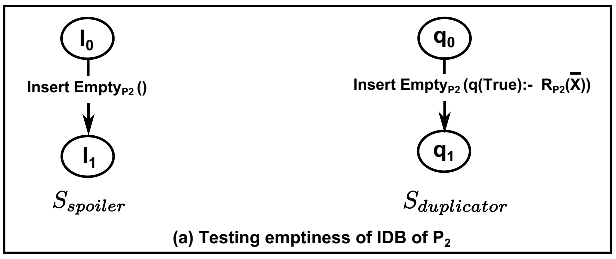

l0
Insert Empty_P2 ()
l1
S_spoiler
q0
Insert Empty_P2 (q(True):- R_P2(X))
q1
S_duplicator
(a) Testing emptiness of IDB of P2

FIGURE 3.17 – Initialization part.

3.5.3 $(\mathcal{L}_T, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ Services

This class enables to study the connection between the *insertion* language and the *guards* language. We will prove that, the simulation for $(GNCQ, \emptyset, \mathcal{L}_U^{insert(CQ)})$ is undecidable. The reduction is obtained from the problem of containment of datalog programs. Given two Datalog programs P_1 and P_2 we construct a test of simulation between two $(GNCQ, \emptyset, \mathcal{L}_U^{insert(CQ)})$ namely $S_{spoiler}$ and $S_{duplicator}$ and we prove that, $P_1 \sqsubseteq P_2$ iff $S_{spoiler} \preceq S_{duplicator}$. The construction is similar to the one used for the case of $(\emptyset, \emptyset, \mathcal{L}_U^{insert(GNCQ)})$ services (Theorem 16), with the only difference being in the part that force $S_{spoiler}$ to calculate the fix point of P_2.

Lemma 26. *Let P_1 and P_2 be two Datalog programs. Then, there exists two $(CQ^{\neq}, \emptyset, \mathcal{L}_U^{insert(CQ})$ services $S_{spoiler}$ and $S_{duplicator}$ such that : $P_1 \sqsubseteq P_2$ iff $S_{spoiler} \preceq S_{duplicator}$.*

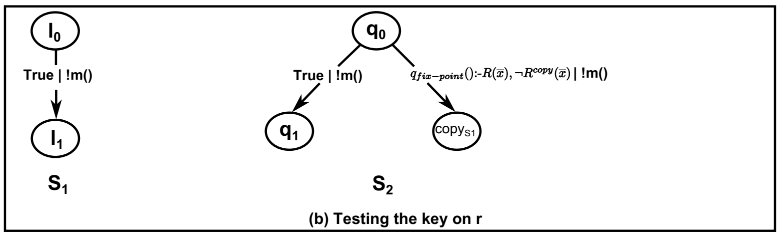

l0
True | !m()
l1
S1
q0
True | !m()
$q_{fix-point}()$:-$R(\overline{x}), \neg R^{copy}(\overline{x})$ | !m()
q1
copy_S1
S2
(b) Testing the key on r

FIGURE 3.18 – testing if $S_{spoiler}$ calculates the fix point of P_2.

Démonstration. We present below the part of services which tests if the service $S_{spoiler}$ calculates the fix point of P_2. Figure 3.18 depicts the part of the two services $S_{spoiler}$ and $S_{duplicator}$ used to ensure that $S_{spoiler}$ calculates the fix point of P_2. Assume that R is an IDB of P_2. The service $S_{spoiler}$ sends the empty message m without any restriction (guard), while $S_{duplicator}$ has two cases : (i) $S_{spoiler}$ has reached a fix point, hence R and R^{copy} contain the same set of tuples. In this case, the transition on the right is not allowed (i.e., this transition is allowed if and only if there exist a tuple in R which is not in R^{copy}). So, $S_{duplicator}$ sends the empty message m and the game of simulation continues, or (ii) $S_{spoiler}$ has not yet reached the fix point of P_2, then $S_{duplicator}$ can execute the two transitions, but the transition on the right ensures that $S_{duplicator}$ will win the simulation game. Hence, to have a chance to win the simulation game, $S_{spoiler}$ must calculate the fix point of P_2. □

Hence, we ca state the following theorem :

Theorem 16. *Checking simulation between two $(GNCQ, \emptyset, \mathcal{L}_U^{insert(CQ)})$ services is undecidable.*

Note that the result of theorem 16 is interesting in the sense that it exhibits a class of services where the satisfiability of guards is decidable while simulation is not.

Chapitre 4

Related Work and Conclusion

In this chapter we review related works then we conclude by summarizing the main results of this thesis and drawing few future research directions.

4.1 Related works

Up to our knowledge, there are only very few works that address the simulation problem in the context of data-centric services. We review below closed works related to data-centric service composition and, independently from the web service area, we mention also similar works in the formal verification area.

[BCG$^+$05] investigates the service composition problem using a very constrained class of Colombo, called $Colombo^{k,b}$, which poses several semantic restrictions : (i) the number of accesses to the database, and (ii) the number of new incoming values. As a consequence, $Colombo^{k,b}$ is included in k-bounded Colombo service (i.e., k-bounded Colombo can access infinitely often to a bounded database, which is not the case for $Colombo^{k,b}$). The main result of [BCG$^+$05] is to show that service composition is 2-EXPTIME in $Colombo^{k,b}$. This is done using a symbolization framework which abstracts the infinite number of configurations into finite numbers of symbolic configurations. The values of variables as well as the domain of instances are taken from a finite symbolic domain. This domain is construct with respect to the constants appearing in the services and the bounds, respectively, k and b.

In [PG09], the authors study the composition problem for data-centric services using an approach based on the simulation relation. More precisely, [PG09] models a service as a finite transition system modifying a shared binary relation. During an execution of a service, the service can only receive one parameter from the outside. The expressiveness of the model is also restricted, for example $y := f_1^R(x)$ cannot be encoded in this model. The obtained framework is still an infinite transition system, where a configuration of a service is made of a control state and an instance of the shared database. The authors shown that, when the database instance is bounded, the service composition problem can be reduced into a simulation test between finite state transition systems. They use a symbolization framework to prove the decidability of simulation. They construct a finite symbolic domain with respect to the bound, then construct the transition systems using this finite symbolic domain and test the simulation between them. The used model is less expressive than k-bounded Colombo model.

[LPT14] addresses the problem of checking simulation between probabilistic data-centric services. He considered the case of data-centric services which take as input a fixed probabilistic database and shows that in this context the simulation problem is in 2-EXPTIME and is EXPTIME-hard.

The simulation problem between infinite transition systems has also been addressed independently from the web service area. This problem is shown undecidable in the general case but there are few classes, e.g., one-counter nets [AC98], automate with finite memory [KF94], where the problem is known to be decidable.

[GKS10] introduces a new formalism called *Variable Automata* (*VA*). A *VA* is a finite state machine where a transition is labelled either with a constant or a variable. The set of variable in a *VA* is made of only one *free* variable and a set of *bounded* variables. During an execution, the values of the *bounded* variables is fixed (the value does not change during an execution) while the value of the *free* variable changes each time the automata execute a transition labelled with this free variable. [GKS10] proved that the language containment in this context is undecidable.

In [BCR13], the authors define *Fresh Variable Automata* (*FVA*). In a *FVA*, the transitions are labelled with constants or variables. Here, during an execution the value of a variable changes in specific states, called *refresh* states. In [BCR14], the authors extend *fresh-variable automata* with guards on transitions (conjunction of equality and inequality over variables and constants). The model is called *guarded variable automata* (*GVA*). *FVA* are not comparable with *VA* and the two models are included in *GVA*. The authors study the simulation for the two models *FVA* and *GVA*. They prove that the problem of simulation is in EXPTIME for *GVA*. The authors prove the decidability of simulation for the guarded variables automata by proving the equivalence between the test of simulation for infinite machines and the test of simulation between two finite state machines. They use a finite symbolic domain representing the constants appearing in the two services and a set of new constant representing the set of variables. As mentioned earlier, as a side effect of our work on DB-less Colombo service, we can derive the EXPTIME-hardness of simulation for *GVA*. More precisely, we can use the same reduction from existing infinite execution for an Alternating Turing machine work on a space polynomially bounded by the size of its input to test of simulation between two *GVA*. A *GVA* can be encoded in a DB-less Colombo service.

Data-centric services attracted a lot of attention from the formal verification community these recent years (see [CDM13] for a detailed survey). The most important models include :

– *relation transducer* [AVFY98], generalized by M.Spielmann in [Spi00] with the Abstract State Machine (ASM) model. The verification problems are undecidable in the general case. M.Spielmann in [Spi00] proves positive results (regarding decidability) of verification problems using a syntactic restriction on the model. This restriction, namely *input boundedness*, ensures that, during an execution, the machine can only access to a bounded number of tuples, hence an input-bounded ASM works only on a bounded database. The same restrictions is also used in another data-centric model proposed by [DSV04]. The authors propose a framework to model a web service interacting with a user. A service is described as a guarded transition system, where control states represent web pages. Each control state provides a set of input choices to the user, based on queries over a fixed database (i.e., does not change during the execution) and a dynamic database (i.e., can be updated during the execution). Transition from one state to another depends on user's choices and the actual instances of the dynamic and static databases.
– The artifact-centric approach was introduced by IBM research in [AN03] and studied in many works [BGL+05, BCK+07, BGH+07, GS07, KRG07, KLW08, DHPV09,

DDHV11, DDV11]. A *Business artifact* record key business-relevant entities. They are augmented with a data model and they are modified with a set of actions. An artifact evolves through its life-cycle (i.e.,transition system) and can interact with other artifacts or external users (see [CH09, Hul08] for a survey on research directions and challenges for this approach). In [BLP11], the authors consider the problem of verifying artifact system against specifications expressed in quantified temporal logic. In this framework, an artifact is made of a database schema, an initial database instance and a set of actions which modify the database. An action is modeled as a precondition and a set of postconditions over the database schema. During an execution, an artifact can introduce new values into the database instance through the actions. The verification problem is undecidable in the general setting. So, the paper considers a semantic restriction by bounding the number of values stored in a state of the system during a given execution. The authors use a specific abstraction technique to construct a finite symbolic system which is bisimilar to the original infinite system. By this way, model checking can be carried out over the (finite) symbolic model instead of the original infinite artefact system. The upper bound time complexity of the proposed procedure is doubly exponential. More precisely, the size of the symbolic system is doubly exponential in the arity of the database schema and the bound, which coincide with our upper bound for simulation. [HCD+13] proposes a syntactic restriction based on the notion of weak acyclicity studied in data exchange [FKMP05] ensuring the boundedness of the database instances during an execution of the artifact.

Note that, the common point between those works on data-centric models is that : the positive results (of the different problems of verification studied) are ensured by bounding the database instance used by the model.

4.2 Conclusion

In this work, we focus on the decidability and complexity of the (weak) simulation preorder for data-centric web services, i.e., checking if the behaviour of a service can be reproduced by another one. We considered services that export their behaviour using state machines augmented with data.

Table 4.1 – Summarization of actual results.

Class of services	Simulation
$Colombo^{Unb}$	Undecidable
$Colombo^{DB=\emptyset}$	EXPTIME-complete
GVA	EXPTIME[BCR14]**-complete**
$Colombo^{bound}$	2-EXPTIME-complete
$(\mathcal{L}_T, \emptyset, \emptyset)$	decidable iff satisfiability of $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ is decidable
$(\emptyset, \mathcal{L}_S, \emptyset)$	Undecidable if satisfiability of $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable
$(\emptyset, \mathcal{L}_S, \emptyset)$	decidable if satisfiability of a partition in $\mathcal{L}_S$ is decidable
$(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$	Undecidable if satisfiability of $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ is undecidable
$(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ)})$	Undecidable
$(\emptyset, \emptyset, \mathcal{L}_U^{Insert(CQ)})^p$	Undecidable
$(GNCQ, \emptyset, \mathcal{L}_U^{Insert(CQ)})$	Undecidable

Table 4.1 summarizing the results of this work. In Chapter 2 we studied the Colombo framework [BCG+05]. A Colombo service is specified as a guarded transition system dealing with a shared database and a set of variables used to send and receive messages. The modification of the database and the variables is achieved through *atomic processes*. An atomic process describes actions in terms of its inputs, outputs, preconditions and postconditions. In this context, we showed that the simulation is undecidable for $Colombo^{unb}$ i.e., services able to read an unbounded number of tuples in the shared database. Then, we focused on the simulation of Colombo services with a bounded database (i.e. the class of Colombo services having global databases with a number of tuples that cannot exceed a given constant k). Such a class is called $Colombo^{bound}$. Hence, by definition, $Colombo^{bound}$ cannot read an unbounded number of information from the database. We showed that the simulation is 2-EXPTIME-complete for $Colombo^{bound}$. The proof is achieved in 2 steps. First, we showed the EXPTIME completeness of the simulation for Colombo services without any access to the database (namely DB-less services $Colombo^{DB=\emptyset}$). As a side effect of this work, we establish a correspondence between $Colombo^{DB=\emptyset}$, restricted to equality, and Guarded Automata (GVA) [BCR14]. As a consequence, we derived EXPTIME completeness of simulation for GVA. Then, we showed that checking the simulation for $Colombo^{bound}$ services can be rewritten into equivalent $Colombo^{DB=\emptyset}$ while preserving the simulation preorder. Up to our knowledge, it is the first results of lower bound for the simulation problem in the context of data-centric web services.

The second part of the thesis tackled the problem of simulation for our generalization of the Colombo model, namely the generic model, where a generic service $(\mathcal{L}_T, \mathcal{L}_S, \mathcal{L}_U)^{?,p}$ uses a local database instead of a set of variables. The messages exchanged are databases, where the outgoing messages are results of queries expressed in a language $\mathcal{L}_S$. The guards are also expressed as boolean queries in a language $\mathcal{L}_T$. Finally, the updates are represented with queries over the language $\mathcal{L}_U$. In this context, we provide first results regarding the decidability of simulation w.r.t the presence or not of each parameter i.e., guards, updates and send query languages. We detailed the results in the following :

- Guarded services $(\mathcal{L}_T, \emptyset, \emptyset)$. For this class, we obtained a full characterization of the decidability of simulation w.r.t to the satisfiability of the language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$.
- Send services $(\emptyset, \mathcal{L}_S, \emptyset)$. Unlike guarded services, we do not obtain a full characterization of decidability of simulation for send services. We provided sufficient conditions of undecidability of simulation w.r.t to the language $\mathcal{L}_S$. More specifically, the simulation is undecidable if the test of satisfiability in the language $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable. In the other hand, the simulation is decidable if the satisfiability of a formula representing a partition is decidable. The language of the formula representing a partition is more expressive than $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$. For guarded ans send services we define a symbolization technique enabling the reduction of test of simulation between two infinite state machine to a finite set of tests of simulation between finite state machines w.r.t the satisfiability of the language considered. This reduction is applicable because the database does not change during the execution of a guarded or send service. Note also that, for the send services, when we consider only the boolean queries over the language $\mathcal{L}_S$ then we obtain a full characterization of the simulation w.r.t the decidability of the satisfiability of the language $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$.
- $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$. For this class, we proved that testing the simulation is undecidable if satisfiability of formula expressed in the insert language $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ is undecidable. Unfortunately, we are not able to provide a full characterization of the decidability of the simulation regarding the insert language $\mathcal{L}_I$. In fact the simulation is undecidable for $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(GNCQ))})$ services, while satisfiability of boolean query over $GNCQ \cup$

  $\{\wedge^b, \neg^b\}$ is decidable. We left the problem open when $\mathcal{L}_I = CQ$.
- $(\emptyset, \emptyset, \mathcal{L}_U^{Insert(CQ)})^p$. We proved that, the weak simulation is undecidable for this class.
- $(\mathcal{L}_T, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$. This class enables to study the interaction between the guards language and updates language. We proved that the simulation is undecidable when $\mathcal{L}_T = GNCQ$ and the language of insertion is $\mathcal{L}_I = CQ$.

Up to our knowledge, the results on generic model are the first results that characterize the problem of simulation for data-centric model w.r.t the satisfiablity of the languages of guards, send messages and updates. We note also that, our abstraction techniques (for Colombo cases and generic model) are not just a way to prove the decidability of the problem of simulation for the different cases, but also a crucial step leading to the implementation of simulation algorithms.

Below, we give some future directions :
- We give tight results on decidability and complexity of the relation of simulation for the Colombo model, but a major assumption we made is that all services share the **same** database. This scenario is realistic when the web services come from the same company, but when we move to an inter-organization framework, the assumption made is not adequate. We plane to handle this case by considering the inclusion rather than the equality between the shared databases of the two services. The same remark can be done for the generic model.
- Another direction we plane to explore is instead of imposing semantic restriction to ensure decidability of simulation of Colombo model (i.e $Colombo^{bound}$), is to find syntactic restriction ensuring the boundedness of the database instance. In this context, we are interested to use the weak acyclicity studied in data exchange [FKMP05] framework (e.g., used to prove the decidability of verification problems in artifact models [HCD+13]). The use of weak acyclicity is not direct and straightforward and need more investigations.
- Finally, we believe that our techniques to prove lower bounds of complexity of simulation for $Colombo^{bound}$ and $Colombo^{DB=\emptyset}$ can be extended to prove the complexity of verification problem in other models where the lower bound is left open (i.e. [HCD+13, BLP11]).
- For the generic model, we plane to address the following problems :
  - The decidability of simulation when the generic service can update the shared database using CQ queries.
  - Find semantics or structural restrictions (i.e., restrictions on the structure of the service : with only self loops) leading to the decidability of the simulation problem. For example, the result on the decidability of verification problems when data-centric services satisfy the weak acyclicity, suggest that this property can be used to prove that generic services that verify weak acyclicity ensure the decidability of the simulation preorder.

Bibliographie

[ABGM09] Serge Abiteboul, Pierre Bourhis, Alban Galland, and Bogdan Marinoiu. The axml artifact model. In *TIME*, pages 11–17, 2009.

[AC98] P.A. Abdulla and K. Cerans. Simulation Is Decidable for One-Counter Nets (Extended Abstract). *Proceedings of CONCUR'98, Lecture Notes in Computer Science 1466 : 253–268*, pages 26–8, 1998.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications.* Springer, 2004.

[AD07] V. Vianu A. Deutsch, L. Sui. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, pages 442–474, 2007.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AN03] NS. Caswell A. Nigam. Business artifacts : An approach to operational specification. *IBM Systems Journal*, 3(42) :428–445, 2003.

[AP07] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Software Eng.*, 33(6) :369–384, 2007.

[ARN12] Mohammad Alrifai, Thomas Risse, and Wolfgang Nejdl. A hybrid approach for efficient web service composition with end-to-end qos constraints. *ACM Trans. Web*, 6(2) :7 :1–7 :31, Jun 2012.

[ASV08] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active xml systems. In *PODS*, pages 221–230, 2008.

[ASV09a] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Modeling and verifying active xml artifacts. *IEEE Data Eng. Bull.*, 32(3) :10–15, 2009.

[ASV09b] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active xml systems. *ACM Trans. Database Syst.*, 34(4), 2009.

[AVFY98] Serge Abiteboul, Victor Vianu, Bradley S. Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *PODS*, pages 179–187, 1998.

[AVFY00] Serge Abiteboul, Victor Vianu, Bradley S. Fordham, and Yelena Yesha. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 61(2) :236–269, 2000.

[BCG$^+$03] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-services that export their behavior. In *ICSOC*, pages 43–58, Dec. 2003.

[BCG$^+$05] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, pages 613–624, 2005.

[BCGP08] Daniela Berardi, Fahima Cheikh, Giuseppe De Giacomo, and Fabio Patrizi. Automatic service composition via simulation. *IJFCS*, 19(2) :429–451, 2008.

[BCK$^+$07] Kamal Bhattacharya, Nathan S. Caswell, Santhosh Kumaran, Anil Nigam, and Frederick Y. Wu. Artifact-centered operational modeling : Lessons from customer engagements. *IBM Systems Journal*, 46(4) :703–721, 2007.

[BCP08] Antonio Brogi, Sara Corfini, and Razvan Popescu. Semantics-based composition-oriented discovery of web services. *ACM Trans. Internet Technol.*, 8(4) :19 :1–19 :39, Oct. 2008.

[BCR13] Walid Belkhir, Yannick Chevalier, and Michaël Rusinowitch. Fresh-variable automata : Application to service composition. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, pages 153–160, 2013.

[BCR14] W. Belkhir, Y. Chevalier, and M. Rusinowitch. Guarded variable automata over infinite alphabets. In *to appear in Journal of Symbolic Computation*, 2014.

[BCT04a] B.Benatallah, F. Casati, and F. Toumani. Analysis and management of web service protocols. In *ER conference, Shanghai, China*, volume 3288 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2004.

[BCT04b] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web service conversation modeling : A cornerstone for e-business automation. *IEEE Internet Computing*, 08(1) :46–54, 2004.

[BCT06] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *DKE*, 58(3) :327–357, 2006.

[BFHS03] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification : a new approach to design and analysis of e-service composition. In *WWW'03*. ACM, 2003.

[BGH$^+$07] Kamal Bhattacharya, Cagdas Evren Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards formal analysis of artifact-centric business process models. In *BPM*, pages 288–304, 2007.

[BGL$^+$05] Kamal Bhattacharya, Robert Guttman, Kelly Lyman, Fenno F. Heath III, Santhosh Kumaran, Prabir Nandi, Frederick Y. Wu, Prasanna Athma, Christoph Freiberg, Lars Johannsen, and Andreas Staudt. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1) :145–162, 2005.

[BLP11] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, pages 142–156, 2011.

[BSBM04] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are two Web Services Compatible ?). In *VLDB TES'04. Toronto, Canada*, 2004.

[BtCS11] Vince Bárány, Balder ten Cate, and Luc Segoufin. Guarded negation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 356–367, 2011.

[CDM13] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis : a database theory perspective. In *PODS*, pages 1–12, 2013.

[CH09] David Cohn and Richard Hull. Business artifacts : A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3) :3–9, 2009.

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1) :114–133, 1981.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[DDHV11] Elio Damaggio, Alin Deutsch, Richard Hull, and Victor Vianu. Automatic verification of data-centric business processes. In *BPM*, pages 3–16, 2011.

[DDV11] Elio Damaggio, Alin Deutsch, and Victor Vianu. Artifact systems with data dependencies and arithmetic. In *ICDT*, pages 66–77, 2011.

[DHPV09] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. *ICDT*, pages 252–267, 2009.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1) :1–30, 2005.

[DSV04] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS*, pages 71–82, 2004.

[FFM$^+$10] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 36(2) :198–215, 2010.

[FGG$^+$08] W. Fan, F. Geerts, W. Gelade, F. Neven, and A. Poggi. Complexity and composition of synthesized web services. In *ACM PODS*, pages 231–240. ACM New York, NY, USA, 2008.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange : semantics and query answering. *Theor. Comput. Sci.*, 336(1) :89–124, 2005.

[GHIS04] Cagdas Evren Gerede, Richard Hull, Oscar H. Ibarra, and Jianwen Su. Automated composition of e-services : lookaheads. In *ICSOC*, pages 252–262, 2004.

[GKS10] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In *LATA*, pages 561–572, 2010.

[GS07] Cagdas E. Gerede and Jianwen Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, pages 181–192, 2007.

[HB03a] R. Hamadi and B. Benatallah. A Petri net-based model for web service composition. *Australasian Database Conference*, pages 191–200, 2003.

[HB03b] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC*, pages 191–200, 2003.

[HCD$^+$13] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 163–174, 2013.

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. pages 453–462. IEEE Computer Society Press, 1995.

[HLL$^+$12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3) :16, 2012.

[HNT08] Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Protocol-based web service composition. In *ICSOC*, pages 38–53, 2008.

[HSV13] Richard Hull, Jianwen Su, and Roman Vaculín. Data management perspectives on business process management : tutorial overview. In *SIGMOD Conference*, pages 943–948, 2013.

[Hul08] R. Hull. artifact-centric business process models : Brief survey of research results and challenges. *Lecture Notes in Computer Science*, 5332/2008 :1152–1163, 2008.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2) :329–363, 1994.

[KLW08] Santhosh Kumaran, Rong Liu, and Frederick Y. Wu. On the duality of information-centric and activity-centric models of business processes. In *CAiSE*, pages 32–47, 2008.

[KM02a] A. Kucera and R. Mayr. Simulation preorder over simple process algebras. *Information and Computation*, 173(2) :184–198, 2002.

[KM02b] Anton ?n Kucera and Richard Mayr. On the complexity of semantic equivalences for pushdown automata and bpa. In *In Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02), volume 2420 of LNCS*, pages 433–445. Springer-Verlag, 2002.

[Kra07] Sacha Krakowiak. Middleware architecture with patterns and frameworks, 2007.

[KRG07] Jochen Malte Küster, Ksenia Ryndina, and Harald Gall. Generation of business process models for object life cycle compliance. In *BPM*, pages 165–181, 2007.

[Loh08] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. *LNCS*, 4937 :77, 2008.

[LPT14] Haizhou Li, François Pinet, and Farouk Toumani. Probabilistic simulation for probabilistic data-aware business processes. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, pages 503–515, 2014.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.

[Min67] M. Minsky. *Computation Finite and Infinite Machines*. Prentice-Hall, 1967.

[MSW11] A. Meyer, S. Smirnov, and M. Weske. Data in business processes. technical report 50, 2011. http ://opus.kobv.de/ubp/volltexte/2011/5304/.

[MW07] Anca Muscholl and Igor Walukiewicz. A lower bound on web services composition. In *FOSSACS*, volume 4423 of *LNCS*, pages 274–287. Springer, 2007.

[NM02] S. Narayanan and S.A. McIlraith. Simulation, verification and automated composition of web services. *WWW'02*, pages 77–88, 2002.

[Pet73] C.A. Petri. Concepts of net theory. In *Proceedings of MFCS*, volume 73, pages 137–146, 1973.

[PF04] S. R. Ponnekanti and A. Fox. Interoperability among Independently Evolving Web Services). In *Middleware '04, Toronto, Canada*, 2004.

[PG09] F. Patrizi and G. De Giacomo. Composition of services that share an infinite-state blackboard (extended abstract). In *IIWEB*, 2009.

[PTB05] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. *ICAPS*, 2005.

[Shm93] Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3) :231–241, 1993.

[Spi00] Marc Spielmann. Verification of relational transducers for electronic commerce. In *PODS*, pages 92–103, 2000.

[vdADO$^+$08] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Techn.*, 8(3), 2008.

[W3C02] W3C. Web service architecture requirements. Technical report, http : //www.w3.org/TR/wsa-reqs/, 2002.

[WMFN04] A. Wombacher, B. Mahleko, P. Fankhauser, and E. Neuhold. Matchmaking for Business Processes based on Choreographies). In *EEE'04, Taipei, Taiwan*, 2004.

[XDMNZ04] A.Y. Halevy X. Dong, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services). In *VLDB'04. Toronto, Canada*, pages 372–383, 2004.

Annexe A

Appendix

Proofs of Chapter 2

Lemma 6 *Each configuration C of the execution of an alternating Turing machine M on the input w has a corresponding configuration in the extended state machine of $S_{duplicator}$*

Démonstration. We will prove that, each configuration C of M correspond to a configuration of the extended state machine of $S_{duplicator}$ and each configuration of the extended state machine of $S_{duplicator}$ correspond to a configuration C of M. The proof is by induction :

– The base : The initial configuration of M is C_0=$qy_1, ..., y_n$. Assume that q is universal. After the part of initialization, $E(S_{duplicator})$ is in an id=(l_q, $\alpha(Lstore)$) where $\alpha(x_i)$=w_i (i.e., the i'th letter of w) and $\alpha(head)$=1. Assume that there exists a move $C_0 \overset{qa/bRq^{'}}{\longrightarrow} C^{'}$, that means y_1=a, so $\alpha(x_1)$=a. From construction of $S_{duplicator}$ there exists a transition $l_q \xrightarrow{g_1^a \ | \ qabq^{'} R_1(\emptyset; x_1, head)} l_{q^{'}}$ where $g_1 : x_1 = a \wedge head = 1$, so $(l_q, \alpha(Lstore)) \xrightarrow{qabq^{'} R_1(\emptyset; \alpha^{'}(x_1), \alpha^{'}(head))} (l_{q^{'}}, \alpha^{'}(Lstore))$ where : (i) $\alpha^{'}(x_1)$= $y_1^{'}$=b, and $\alpha^{'}(head)$=2.

– iteration i : Now Assume that M is in configuration C=$y_1, ..., qy_j, ..., y_n$, and there exists in $E(S_{duplicator})$ an id=($l_q, \alpha(Lstore)$), where y_i=$\alpha(x_i)$ and $\alpha(head)$=j. If $C \overset{qa/bRq^{'}}{\longrightarrow} C^{'}$ exists, then y_j=a, $y_j^{'}$=b and $\alpha(x_j)$=a. From construction of $S_{duplicator}$, we know there is a transition $l_q \xrightarrow{g_1^a \ | \ qabq^{'} R_j(\emptyset; x_j, head)} l_{q^{'}}$ where $g_1 : x_1 = a \wedge head = j$, so there exists in $E(S_{duplicator})$ a transition $(l_q, \alpha(Lstore)) \xrightarrow{qabq^{'} R_j(\emptyset; \alpha^{'}(x_j), \alpha^{'}(head))} (l_{q^{'}}, \alpha^{'}(Lstore))$ where : (i) $\alpha^{'}(x_j)$= $y_j^{'}$=b, and $\alpha^{'}(head)$=j+1.

The same reasoning is used if M is in a configuration where the state is existential, the only difference is $S_{duplicator}$ starts by sending the message $m()$, then executes the atomic process $qabq^{'} R_j(\emptyset; x_j, head)$, this additional transition does not change the values of the variables.
Now, we will prove the second direction, i.e.,each configuration of the extended state machine of $S_{duplicator}$ correspond to a configuration C of M.

– The base : From the construction of $S_{duplicator}$, after the part of initialization the execution of the service reaches an id=($l_q, \alpha(Lstore)$), where q is the initial state of M, Assume that it is universal. $\alpha(x_i)$=w_i and $\alpha(head)$=1. If in $E(S_{duplicator})$ there exists a transition $(l_q, \alpha(Lstore)) \xrightarrow{qabq^{'}R_1(\emptyset;\alpha^{'}(x_1),\alpha^{'}(head))} (l_{q^{'}}, \alpha^{'}(Lstore))$, then $\alpha^{'}(x_1)$=b and $\alpha^{'}(head)$=2. From construction of $S_{duplicator}$, there exists $C_0 \overset{qa/bRq^{'}}{\longrightarrow} C^{'}$ where y_1=a and $y^{'}_1$=b.

– Iteration i : Assume that in $E(S_{duplicator})$ there is an id=$(l_q, \alpha(Lstore))$, where q is an universal state of M and $\alpha(x_i)$=w_i. Assume that $\alpha(head)$=j and there is a configuration of M, C=$y_1, ..., qy_j, ..., y_n$, where $\alpha(x_i)$=y_i. If in $E(S_{duplicator})$, there exists a transition $(l_q, \alpha(Lstore)) \xrightarrow{qabq^{'}R_j(\emptyset;\alpha^{'}(x_j),\alpha^{'}(head))} (l_{q^{'}}, \alpha^{'}(Lstore))$, then $\alpha^{'}(x_j)$=b and $\alpha^{'}(head)$=j+1. From construction of $S_{duplicator}$, there exists $C \overset{qa/bRq^{'}}{\longrightarrow} C^{'}$ where y_j=a and $y^{'}_1$=b.

□

Lemma 7 *Given an alternating Turing machine M working in space bounded by the size of the input w, M has an infinite computation on w iff* $S_{spoiler} \preceq S_{duplicator}$.

Démonstration. The services $S_{spoiler}$ and $S_{duplicator}$ start by initializing the variables. If M has a transition $q \overset{a/bR}{\longrightarrow} q^{'}$ and q universal,then $S_{spoiler}$ has $n-1$ loops : $q_{universal} \xrightarrow{qabq^{'}R_i(\emptyset;x_i,head)} q_{universal}$ and the service $S_{duplicator}$ contains $n-1$ transitions from $l_q \xrightarrow{g^a_i \ | \ qabq^{'}R_i(\emptyset;x_i,head)} l_{q^{'}}$. Hence, the difference with $S_{spoiler}$ is that $S_{duplicator}$ can only execute one transition representing the action $q \overset{a/bR}{\longrightarrow} q^{'}$ if the actual value of x_i=a and the head points on i. Assume that the condition is verified, then if $S_{spoiler}$ chooses to execute any transition different from $qabq^{'}R_i(\emptyset; x_i, head)$, the service $S_{duplicator}$ wins the game by choosing the transition which reaches the state l_{copy}. If $S_{spoiler}$ chooses to execute $qabq^{'}R_i(\emptyset; x_i, head)$, then $S_{duplicator}$ executes $qabq^{'}R_i(\emptyset; x_i, head)$ and the game continue. Now, Assume that the condition is not verified, the service $S_{duplicator}$ is blocked, then $S_{spoiler}$ wins the game by executing $qabq^{'}R_i(\emptyset; x_i, head)$.
If q is existential, $S_{spoiler}$ has a transition $q_{univ} \overset{!m()}{\longrightarrow} q_{exist}$ and $n-1$ transitions from $q_{exist} \xrightarrow{qabq^{'}R_i(\emptyset;x_i,head)} q_{univ}$. $S_{duplicator}$ contains a transition from $l_q \overset{!m()}{\longrightarrow} l_{qbRq^{'}}$ and $n-1$ transitions $l_{qbRq^{'}} \xrightarrow{g^a_i \ | \ qabq^{'}R_i(\emptyset;x_i,head)} l_{q^{'}}$. Assume that the actual values of the variables verify the condition. If $S_{spoiler}$ chooses another action different from sending the message it looses. If it sends the message, it reaches the state q_{exist}. Then $S_{duplicator}$ reaches an intermediate state $l_{qabRq^{'}}$ by sending the message. $S_{spoiler}$ can do any action but if it chooses an action different from $qabq^{'}R_i(\emptyset; x_i, head)$ it loose the game, if it chooses the action $q \overset{a/bR}{\longrightarrow} q^{'}$, then the game continues.
Note that, after sending the message $m()$, $S_{spoiler}$ reaches l_{exist}. This transition can be simulated by many transition of $S_{duplicator}$ because $S_{duplicator}$ may have several transitions labelled with sending $m()$ at l_q. So, there is no simulation if all tests of simulation are false. That means $S_{duplicator}$ is blocked whatever it chooses. If one choice is not blocking the game continue and hence there is simulation iff the duplicator can always chooses a non blocking state for existential transitions of M and does not block for all universal transition of M. Which means M during its execution has always a successor, so there exists an infinite computation. □

Lemma 9 *Let S be a Colombo service and $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}_0^k, \mathbb{F}^k, \Delta^k)$ its k-bounded extended state machine and $E(\mathcal{M}(S))$ the extended state machine of DB-less $\mathcal{M}(S)$, then*

– *If $(q_i, \mathcal{I}_i, \alpha_i) \in \mathbb{Q}^k$ then $\exists$ $(q_i, \alpha_i^{'}) \in \mathbb{Q}_{\mathcal{M}(S)}$ s.t $\alpha^{'}_{i|Lstore} = \alpha_i$ and $\alpha^{'}_{i|DV} = \mathcal{I}_i$ and*

– *$\forall$ $(q_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (q_j, \mathcal{I}_j, \alpha_j)$, $\exists$ $(q_i, \alpha_i^{'}) \xrightarrow{\mu_i^{'}} (q_j^{'}, \alpha_j^{'})$ s.t $\alpha^{'}_{j|Lstore} = \alpha_j$ and $\alpha^{'}_{j|DV} = \mathcal{I}_j$.*

Lemma 9 asserts that for each state in the k-bounded state machine of S there exists a corresponding state in the extended state machine of $\mathcal{M}(S)$ s.t the valuation of DV is equal to database $\mathcal{I}$ and the valuation of variables of $Lstore$ in the two states are equal. The proof is by induction.

Démonstration. **The base** :

1. Assume that $(q_0, \mathcal{I}_0, \alpha_0) \in \mathbb{Q}^k$, from the construction of $\mathcal{M}(S)$ there exists a state $(q_0, \alpha_0^{'}) \in \mathbb{Q}_{\mathcal{M}(S)}$ s.t (i)$\alpha^{'}_{0|Lstore} = \alpha_0 = \emptyset$ and(ii) $\alpha^{'}_{0|DV} = \mathcal{I}_0$. The first point is easy to see, from the definition of an execution of a service, all variables start with null value. The second point come from the construction of $\mathcal{M}(S)$, where there is a transition from q_{init} to q_0 labelled with reception of messages containing values of DV.
2. Let $(q_0, \mathcal{I}_0, \alpha_0) \xrightarrow{\mu_0} (q_j, \mathcal{I}_j, \alpha_j)$ a transition in Δ^k, then from the construction of $\mathcal{M}(S)$ and 1, we know there exists a transition $(q_0, \alpha_0^{'}) \xrightarrow{\mu_0^{'}} (q_j^{'}, \alpha_j^{'})$ where :
   - if $\mu_0 = ?m(\alpha_j(u_1), ..., \alpha_j(u_n))$, then $\mu_0^{'} = ?m(\alpha_j^{'}(u_1), ..., \alpha_j{'}(u_n))$ where :
     - $\alpha^{'}_{j|Lstore-\{v_1,...,v_n\}} = \alpha_{j|Lstore-\{v_1,...,v_n\}}$.
     - $\alpha_j^{'}(v_k) = \alpha_j(v_k)$ for k $\in [1, ..., n]$. This is due to the fact the two substitution have the same infinite co-domain.
     - $\alpha^{'}_{j|DV} = \mathcal{I}_j$ because : $\mathcal{I}_0 = \mathcal{I}_j$, $\alpha^{'}_{0|DV} = \mathcal{I}_0$ and $\alpha^{'}_{j|DV} = \alpha^{'}_{0|DV}$.
   - if $\mu_0 = !m(\alpha_0(u_1), ..., \alpha_0(u_n))$, then $\mu_0^{'} = !m(\alpha_0^{'}(u_1), ..., \alpha_0{'}(u_n))$ where :
     - $\alpha^{'}_{j|Lstore} = \alpha_j$: because $\alpha_0 = \alpha_j$ and $\alpha_0 = \alpha^{'}_{0|Lstore}$ and $\alpha_0^{'} = \alpha_j^{'}$.
     - $\alpha^{'}_{j|DV} = \mathcal{I}_j$: because $\mathcal{I}_0 = \alpha^{'}_{0|DV}$ and $\mathcal{I}_0 = \mathcal{I}_j$ and $\alpha_0^{'} = \alpha_j^{'}$.
   - if $\mu_0 = p(\alpha_0(u_1), ..., \alpha_0(u_n), \alpha_0(DV); \alpha_j(v_1), ..., \alpha_j(v_m), \alpha_j(DV))$, then $\mu_0^{'} = p(\alpha_0^{'}(u_1), ..., \alpha_0^{'}(u_n), \alpha_0^{'}(DV); \alpha_j^{'}(v_1), ..., \alpha_j^{'}(v_m), \alpha_j^{'}(DV))$ where :
     - $\alpha^{'}_{j|DV} = \mathcal{I}_j$: from construction of $\mathcal{M}(S)$ and $\alpha^{'}_{0|Lstore} = \alpha_0$ and $\alpha^{'}_{0|DV} = \mathcal{I}_0$, DV is modified regarding to updates made by p, where the values of variables depends on inputs and the database.
     - $\alpha^{'}_{j|Lstore-\{v_1,...,v_m\}} = \alpha_{j|Lstore-\{v_1,...,v_m\}}$ because : $\alpha^{'}_{i|Lstore-\{v_1,...,v_m\}} = \alpha^{'}_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}} = \alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha^{'}_{i|Lstore-\{v_1,...,v_m\}} = \alpha_{i|Lstore-\{v_1,...,v_m\}}$.
     - $\alpha^{'}_{j|\{v_1,...,v_m\}} = \alpha_{j|\{v_1,...,v_m\}}$: From the construction of p_v and $\alpha_i = \alpha_i^{'}$ and $\alpha^{'}_{i|DV} = \mathcal{I}_0$.

The iteration i :

1. Assume that $(q_i, \mathcal{I}_i, \alpha_i) \in \mathbb{Q}^k$, and there exists $(q_i, \alpha_i^{'}) \in \Delta_{\mathcal{M}(S)}$ where $\mathcal{I}_i = \alpha^{'}_{i|DV}$ and $\alpha_i = \alpha^{'}_{i|Lstore}$, then for each transition $(q_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (q_j, \mathcal{I}_j, \alpha_j)$ in Δ^k, we know there exists a transition $(q_0, \alpha_0^{'}) \xrightarrow{\mu_0^{'}} (q_j^{'}, \alpha_j^{'})$ from the construction of $\mathcal{M}(S)$ and 1 where :

- if μ_i=$?m(\alpha_j(u_1),...,\alpha_j(u_n))$, then $\mu_i^{'}$=$?m(\alpha_j^{'}(u_1),...,\alpha_j{'}(u_n))$ where :
  - $\alpha^{'}_{j|Lstore-\{v_1,...,v_n\}} = \alpha_{j|Lstore-\{v_1,...,v_n\}}$: because $\alpha_{i|Lstore-\{v_1,...,v_m\}} = \alpha^{'}_{i|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}} = \alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha^{'}_{i|Lstore-\{v_1,...,v_m\}} = \alpha^{'}_{j|Lstore-\{v_1,...,v_m\}}$.
  - $\alpha_j^{'}(v_k) = \alpha_j(v_k)$ for k $\in [1,...,n]$. This is due to the fact the two substitution have the same infinite co-domain.
  - $\alpha^{'}_{j|DV} = \mathcal{I}_j$ because : $\mathcal{I}_i$= $\mathcal{I}_j$, $\alpha^{'}_{i|DV} = \mathcal{I}_i$ and $\alpha^{'}_{j|DV} = \alpha^{'}_{i|DV}$.
- if μ_i=$!m(\alpha_i(u_1),...,\alpha_i(u_n))$, then $\mu_i^{'}$=$!m(\alpha_i^{'}(u_1),...,\alpha_i{'}(u_n))$ where :
  - $\alpha^{'}_{j|Lstore} = \alpha_j$: because $\alpha_i = \alpha_j$ and $\alpha_i = \alpha^{'}_{i|Lstore}$ and $\alpha_i^{'}$= $\alpha_j^{'}$.
  - $\alpha^{'}_{j|DV} = \mathcal{I}_j$: because $\mathcal{I}_i = \alpha^{'}_{i|DV}$ and $\mathcal{I}_i = \mathcal{I}_j$ and $\alpha_i^{'}$= $\alpha_j^{'}$.
- if μ_i=$p(\alpha_i(u_1),...,\alpha_i(u_n),\alpha_i(DV);\alpha_j(v_1),...,\alpha_j(v_m),\alpha_j(DV))$, then $\mu_i^{'}$=$p(\alpha_i^{'}(u_1),...,\alpha_i^{'}(u_n),\alpha_i^{'}(DV);\alpha_j^{'}(v_1),...,\alpha_j^{'}(v_m),\alpha_j^{'}(DV))$ where :
  - $\alpha^{'}_{j|DV} = \mathcal{I}_j$: from construction of $\mathcal{M}(S)$ and $\alpha^{'}_{i|Lstore} = \alpha_i$ and $\alpha^{'}_{i|DV}$= $\mathcal{I}_i$, DV are modified regarding to updates made by p, where the values of variables depends on inputs and the database.
  - $\alpha^{'}_{j|Lstore-\{v_1,...,v_m\}} = \alpha_{j|Lstore-\{v_1,...,v_m\}}$ because : $\alpha^{'}_{0|Lstore-\{v_1,...,v_m\}} = \alpha^{'}_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{0|Lstore-\{v_1,...,v_m\}}$= $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha^{'}_{0|Lstore-\{v_1,...,v_m\}} = \alpha_{0|Lstore-\{v_1,...,v_m\}}$.
  - $\alpha^{'}_{j|\{v_1,...,v_m\}} = \alpha_{j|\{v_1,...,v_m\}}$: because α_0= $\alpha_0^{'}$ and $\alpha^{'}_{0|DV} = \mathcal{I}_0$.

□

Lemma10 *Let S be a Colombo service and $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}^k_0, \mathbb{F}^k, \Delta^k)$ its k-bounded extended state machine and $E(\mathcal{M}(S))$ the extended state machine of DB-less $\mathcal{M}(S)$, then*
- *If $(q_i, \alpha_i^{'}) \in \mathbb{Q}_{\mathcal{M}(S)}$ then $\exists$ $(q_i, \mathcal{I}_i, \alpha_i) \in \mathbb{Q}^k$ s.t $\alpha^{'}_{i|Lstore}$=α_i and $\alpha^{'}_{i|DV}$=$\mathcal{I}_i$ and*
- *$\forall$ $(q_i, \alpha_i^{'}) \xrightarrow{\mu_i^{'}} (q_j^{'}, \alpha_j^{'})$, $\exists$ $(q_i, \mathcal{I}_i, \alpha_i) \xrightarrow{\mu_i} (q_j, \mathcal{I}_j, \alpha_j)$ s.t $\alpha^{'}_{j|Lstore} = \alpha_j$ and $\alpha^{'}_{j|DV} = \mathcal{I}_j$.*

Démonstration. The proof of this lemma is in the same spirit of the previous one, we will prove by induction the correspondence between the two extended state machines :
The base :

1. Assume that $(q_0, \alpha_0^{'}) \in \mathbb{Q}_{\mathcal{M}(S)}$, from the construction of $\mathcal{M}(S)$ there exists a state $(q_0, \mathcal{I}_0, \alpha_0) \in \mathbb{Q}^k$ s.t (i) $\alpha^{'}_{0|Lstore} = \alpha_0$= $\emptyset$ and (ii) $\alpha^{'}_{0|DV} = \mathcal{I}_0$. (i) comes from the definition of an execution of a service, all variables of Lstore start with null values. (ii) comes from the construction of $\mathcal{M}(S)$, where there is a transition from q_{init} to q_0 labelled with reception of messages over all variables of DV, because the variables and the database are range over the same infinite domain, necessarily there exists a valuation of the variables of the message $\alpha_0^{'}(DV)$=$\mathcal{I}_0$.

2. Let $(q_0, \alpha_0^{'}) \xrightarrow{\mu_0^{'}} (q_j^{'}, \alpha_j^{'})$ a transition in $\Delta_{\mathcal{M}(S)}$, then from the construction of $\mathcal{M}(S)$ and 1, we know there exists a transition $(q_0, \mathcal{I}_0, \alpha_0) \xrightarrow{\mu_0} (q_j, \mathcal{I}_j, \alpha_j)$ in Δ^k where :
- if $\mu_0^{'}$=$?m(\alpha_j^{'}(u_1),...,\alpha_j{'}(u_n))$, then μ_0=$?m(\alpha_j(u_1),...,\alpha_j(u_n))$ where :
  - $\alpha_{j|Lstore-\{v_1,...,v_n\}} = \alpha^{'}_{j|Lstore-\{v_1,...,v_n\}}$.
  - $\alpha_j(v_k) = \alpha_j^{'}(v_k)$ for k $\in [1,...,n]$. This is due to the fact the two substitution have the same infinite co-domain.
  - $\mathcal{I}_j = \alpha^{'}_{j|DV}$ because : $\alpha^{'}_{j|DV} = \alpha^{'}_{0|DV}$, $\alpha^{'}_{0|DV} = \mathcal{I}_0$ and $\mathcal{I}_0$= $\mathcal{I}_j$.

- if $\mu_0^{'}$=$!m(\alpha_0^{'}(u_1),...,\alpha_0{'}(u_n))$, then μ_0=$!m(\alpha_0(u_1),...,\alpha_0(u_n))$ where :
  - $\alpha_j=\alpha_{j|Lstore}^{'}$: because $\alpha_0^{'}$= $\alpha_j^{'}$ and $\alpha_0=\alpha_{0|Lstore}^{'}$ and $\alpha_0=\alpha_j$.
  - $\mathcal{I}_j$ =$\alpha_{j|DV}^{'}$: because $\alpha_0^{'}$= $\alpha_j^{'}$ and $\mathcal{I}_0=\mathcal{I}_j$ and $\mathcal{I}_0=\alpha_{0|DV}^{'}$.
- if $\mu_0^{'}$=$p(\alpha_0^{'}(u_1),...,\alpha_0^{'}(u_n),\alpha_0^{'}(DV);\alpha_j^{'}(v_1),...,\alpha_j^{'}(v_m),\alpha_j^{'}(DV))$, then μ_0=$p(\alpha_0(u_1),...,\alpha_0(u_n),\alpha_0(DV);\alpha_j(v_1),...,\alpha_j(v_m),\alpha_j(DV))$ where :
  - $\mathcal{I}_j$ =$\alpha_{j|DV}^{'}$: from construction of $\mathcal{M}(S)$ and $\alpha_{0|Lstore}^{'}=\alpha_0$ and $\alpha_{0|DV}^{'}$= $\mathcal{I}_0$, DV is modified regarding to updates made by p, where the values of variables depends on inputs and the database.
  - $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ =$\alpha_{j|Lstore-\{v_1,...,v_m\}}^{'}$ because : $\alpha_{i|Lstore-\{v_1,...,v_m\}}^{'}$ $=$ $\alpha_{i|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}}$= $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}}^{'}=\alpha_{j|Lstore-\{v_1,...,v_m\}}^{'}$.
  - $\alpha_{j|\{v_1,...,v_m\}}=\alpha_{j|\{v_1,...,v_m\}}^{'}$: From the construction of p_v and $\alpha_{i|DV}^{'}=\mathcal{I}_0$ and α_i= $\alpha_i^{'}$.

The iteration i :

1. Assume that $(q_i,\alpha_i^{'})\in\Delta_{\mathcal{M}(S)}$, and there exists $(q_i,\mathcal{I}_i,\alpha_i)\in\mathbb{Q}^k$ where $\alpha_{i|DV}^{'}=\mathcal{I}_i$ and $\alpha_{i|Lstore}^{'}=\alpha_i$, then for each transition $(q_0,\alpha_0^{'})\xrightarrow{\mu_0^{'}}(q_j^{'},\alpha_j^{'})$, we know there exists a transition $(q_i,\mathcal{I}_i,\alpha_i)\xrightarrow{\mu_i}(q_j,\mathcal{I}_j,\alpha_j)$ in Δ^k from the construction of $\mathcal{M}(S)$ and 1 where :
- if $\mu_i^{'}$=$?m(\alpha_j^{'}(u_1),...,\alpha_j{'}(u_n))$, then μ_i=$?m(\alpha_j(u_1),...,\alpha_j(u_n))$ where :
  - $\alpha_{j|Lstore-\{v_1,...,v_n\}}$ $=$ $\alpha_{j|Lstore-\{v_1,...,v_n\}}^{'}$: because $\alpha_{j|Lstore-\{v_1,...,v_m\}}^{'}$ $=$ $\alpha_{i|Lstore-\{v_1,...,v_m\}}^{'}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}}$ $=$ $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{i|Lstore-\{v_1,...,v_m\}}^{'}=\alpha_{i|Lstore-\{v_1,...,v_m\}}$.
  - $\alpha_j(v_k)=\alpha_j^{'}(v_k)$ for k $\in[1,...,n]$. This is due to the fact the two substitution have the same infinite co-domain.
  - $\mathcal{I}_j=\alpha_{j|DV}^{'}$ because : $\alpha_{j|DV}^{'}=\alpha_{i|DV}^{'}$, $\alpha_{i|DV}^{'}=\mathcal{I}_i$ and $\mathcal{I}_i$= $\mathcal{I}_j$.
- if $\mu_i^{'}$=$!m(\alpha_i^{'}(u_1),...,\alpha_i{'}(u_n))$, then μ_i=$!m(\alpha_i(u_1),...,\alpha_i(u_n))$ where :
  - $\alpha_j=\alpha_{j|Lstore}^{'}$: because $\alpha_i^{'}$= $\alpha_j^{'}$ and $\alpha_i=\alpha_{i|Lstore}^{'}$ and $\alpha_i=\alpha_j$.
  - $\mathcal{I}_j=\alpha_{j|DV}^{'}$: because $\alpha_i^{'}$= $\alpha_j^{'}$ and $\mathcal{I}_i=\mathcal{I}_j$ and $\mathcal{I}_i=\alpha_{i|DV}^{'}$.
- if $\mu_i^{'}$=$p(\alpha_i^{'}(u_1),...,\alpha_i^{'}(u_n),\alpha_i^{'}(DV);\alpha_j^{'}(v_1),...,\alpha_j^{'}(v_m),\alpha_j^{'}(DV))$, then μ_i=$p(\alpha_i(u_1),...,\alpha_i(u_n),\alpha_i(DV);\alpha_j(v_1),...,\alpha_j(v_m),\alpha_j(DV))$ where :
  - $\mathcal{I}_j=\alpha_{j|DV}^{'}$: from construction of $\mathcal{M}(S)$ and $\alpha_{i|Lstore}^{'}=\alpha_i$ and $\alpha_{i|DV}^{'}$= $\mathcal{I}_i$, DV are modified regarding to updates made by p, where the values of variables depends on inputs and the database.
  - $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ $=$ $\alpha_{j|Lstore-\{v_1,...,v_m\}}^{'}$ because : $\alpha_{0|Lstore-\{v_1,...,v_m\}}^{'}$ $=$ $\alpha_{0|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{0|Lstore-\{v_1,...,v_m\}}$= $\alpha_{j|Lstore-\{v_1,...,v_m\}}$ and $\alpha_{0|Lstore-\{v_1,...,v_m\}}^{'}=\alpha_{j|Lstore-\{v_1,...,v_m\}}^{'}$.
  - $\alpha_{j|\{v_1,...,v_m\}}=\alpha_{j|\{v_1,...,v_m\}}^{'}$: because $\alpha_{0|DV}^{'}=\mathcal{I}_0$ and α_0= $\alpha_0^{'}$.

□

Lemma 11 *Each configuration C of the execution of an alternating Turing machine M on the input w has a corresponding configuration in the extended state machine of $S_{duplicator}$.*

Démonstration. We will prove that, each configuration C of M correspond to a configuration of the extended state machine of $S_{duplicator}$ and each configuration of the extended state machine of $S_{duplicator}$ correspond to a configuration C of M. Here a configuration C of M is $y_1, ..., qy_j, ..., y_{2^n}$, where the head points on the j'th cell. The proof is by induction :

– the base : Assume that the initial configuration of M is $C_0{=}qy_1, ..., y_{2^n}$ and q universal, from construction of $S_{duplicator}$, after checking the database and initializing n first tuples with the word w, $E(S_{duplicator})$ is in id=$(l_q, \alpha(Lstore), I)$, where the binary number $\alpha(x_1)...\alpha(x_n)$ points on the first tuple and $f^R_{n+1}(\alpha(x_1, ..., \alpha(x_n)){=}y_1$. Assume that $C_0 \overset{qabRq'}{\longrightarrow} C'$ exists, that means y_1=a in C_0, so $f^R_{n+1}(\alpha(x_1), ..., \alpha(x_n) = $ a. From construction of $S_{duplicator}$ there is a transition $l_q \xrightarrow{\textsf{true} \ | \ get\_cell_{qabq'R}(x_1,...,x_n;letter)} l'_{qbdq'}$ so id $\xrightarrow{get\_cell_{qabq'R}(x_1,...,x_n;letter)} id'$ exist where letter in id' is equal to a. There is also a transition $l'_{qbdq'} \xrightarrow{letter=a \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'}$, so id' $\xrightarrow{set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset} id''$ because letter=a. y_1 in C' is equal to b and the head point in the second cell. $f^R_{n+1}(x_1, ..., x_n)$=b in id''. in $S_{duplicator}$ there is a transition $l''_{qbdq'} \xrightarrow{\neg(x_1=1\wedge...\wedge x_n=1) \ | \ NEXT(x_1,...,x_n;x_1,...,x_n)} l_{q'}$ so there is a transition id'' $\xrightarrow{NEXT(x_1,...,x_n;x_1,...,x_n)} id'''$, where the binary number $x_1...x_n$ in id''' is equal to 2. We can conclude that C' is encoded in id'''.
If $C_0 \overset{qa/bRq'}{\longrightarrow} C'$ does not exist, then letter is different from a in id' and the condition of $l'_{qbdq'} \xrightarrow{letter=a \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'}$ is not verified so the execution of $S_{duplicator}$ blocks.

– the iteration i : Now Assume that the execution of M is in configuration $C=$ $y_1, ..., qy_j, ..., y_{2^n}$ where q is universal and there exists an id in $E(S_{duplicator})$ where the binary number $x_1...x_n$ is equal to j and the control state is l_q. Assume that $C \overset{qabRq'}{\longrightarrow} C'$ exists, that means y_j=a in C, so $f^R_{n+1}(\alpha(x_1), ..., \alpha(x_n)) = $ a. From construction of $S_{duplicator}$ there is a transition $l_q \xrightarrow{\textsf{true} \ | \ get\_cell_{qabq'R}(x_1,...,x_n;letter)} l'_{qbdq'}$ so id $\xrightarrow{get\_cell_{qabq'R}(x_1,...,x_n;letter)} id'$ exist where letter in id' is equal to a. There is also a transition $l'_{qbdq'} \xrightarrow{letter=a \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'}$, so id' $\xrightarrow{set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset} id''$ because letter=a. y_j in C' is equal to b and the head point in the j +1'th cell. $f^R_{n+1}(x_1, ..., x_n)$=b in id'' in $S_{duplicator}$ there is a transition $l''_{qbdq'} \xrightarrow{\neg(x_1=1\wedge...\wedge x_n=1) \ | \ NEXT(x_1,...,x_n;x_1,...,x_n)} l_{q'}$) so there is a transition id'' $\xrightarrow{NEXT(x_1,...,x_n;x_1,...,x_n)} id'''$, where the binary number $x_1...x_n$ in id''' is equal to $j+1$. We can conclude that C' is encoded in id'''.

Now, we will prove the second direction, i.e.,each configuration of the extended state machine of $S_{duplicator}$ correspond to a configuration C of M.

– the base : From construction of $S_{duplicator}$, after the part if initialisation, the execution of the service is in an id, with the control state l_q where q is the initial state of M, Assume that it is universal. all x_i are equal to zero, and the n first tuples contains the letters of the input word w, then id correspond to C_0 in the execution of M, and $y_1{=}f^R_{n+1}(\alpha(x_1, ..., \alpha(x_n))$. If there exists in $E(S_{duplicator})$ transitions id $\xrightarrow{get\_cell_{qabq'R}(x_1,...,x_n;letter)} id' \xrightarrow{set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset} id'' \xrightarrow{NEXT(x_1,...,x_n;x_1,...,x_n)}$ id''', then from construction of $S_{duplicator}$, there exists a transition $q \overset{a/bR}{\longrightarrow} q'$ in M.

We know also $f^R_{n+1}(\alpha(x_1,...,\alpha(x_n))$=a in id and letter=a in d' so y_1=a. Then $C_0 \overset{qabRq'}{\longrightarrow} C'$. $f^R_{n+1}(\alpha''(x_1,...,\alpha''(x_n))$=b in id''. Then y_1 in C'=b, and the head point on 2, because after executing next $x_1...x_n$=1.

– iteration i : Assume that $E(S_{duplicator})$ contains an id where the control state correspond to an universal state of the machine and id correspond to a configuration C of the machine. Let a set of transition id $\xrightarrow{get\_cell_{qabq'R}(x_1,...,x_n;letter)}$ $id' \xrightarrow{set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} id'' \xrightarrow{NEXT(x_1,...,x_n;x_1,...,x_n)} id'''$. From construction of $S_{duplicator}$, there is a transition $q \overset{a/bR}{\longrightarrow} q'$ in M. y_j=$f^R_{n+1}(\alpha(x_1,...,\alpha(x_n))$ in id. letter=a in d' so y_j=a, then $C \overset{qabRq'}{\longrightarrow} C'$, $f^R_{n+1}(\alpha''(x_1,...,\alpha''(x_n))$=b in id''. Then y_j in C'=b, and the head point on j+1, because after executing NEXT, the binary number $x_1...x_n$=j.

The same reasoning is used where q is existential with the difference that $S_{duplicator}$ has an additional transition before executing get_cell, it send the message $m()$, which does not change the values of the variables nor the database. Also for the action labelled with L, we just replace NEXT by PREVIOUS. □

Lemma 12 *Given an alternating Turing machine M working in space exponentially bounded by the size n of the input word w. M has an infinite computation on w iff $S_{spoiler}$ $\preceq$ $S_{duplicator}$.*

Démonstration. $S_{spoiler}$, $S_{duplicator}$ start by checking the database and initializing the n first tuple with the input word w. If M has a transition $q \overset{a/bR}{\longrightarrow} q'$ and q universal,then $S_{spoiler}$ has a loop : $l_\forall \xrightarrow{\text{true} \ | \ get\_cell_{qabq'R}(x_1,...,x_n;letter)} l_{qbdq'} \xrightarrow{\text{true} \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)}$ $l'_{qbdq'} \xrightarrow{\neg(x_1=1\wedge...\wedge x_n=1) \ | \ NEXT(x_1,...,x_n;x_1,...,x_n)} l_\forall$

and the service $S_{duplicator}$ contains transitions $l_q \xrightarrow{\text{true} \ | \ get\_cell_{qabq'R}(x_1,...,x_n;letter)}$ $l'_{qbdq'} \xrightarrow{letter=a \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'} \xrightarrow{\neg(x_1=1\wedge...\wedge x_n=1) \ | \ NEXT(x_1,...,x_n;x_1,...,x_n)} l_{q'}$.
So the difference with $S_{spoiler}$ is that, $S_{duplicator}$ can only execute the transition $l'_{qbdq'} \xrightarrow{letter=a \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'}$ if $x_1...x_n$ points on a tuple with value of W=a. Assume that the condition is verified, then if $S_{spoiler}$ chose to execute any transition with label different from $get\_cell_{qabq'R}(x_1,...,x_n;letter)$, $S_{duplicator}$ wins the game by choosing the transition which reaches the state l_{copy}, if $S_{spoiler}$ chooses to execute $get\_cell_{qabq'R}(x_1,...,x_n;letter)$, then $S_{duplicator}$ execute $get\_cell_{qabq'R}(x_1,...,x_n;letter)$ and it can execute $set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)$ and the game continue. Now, assume that the condition is not verified, then if $S_{spoiler}$ chose to execute any transition with label different from $get\_cell_{qabq'R}(x_1,...,x_n;letter)$, $S_{duplicator}$ wins the game by choosing the transition which reaches the state l_{copy}, if $S_{spoiler}$ chooses to execute $get\_cell_{qabq'R}(x_1,...,x_n;letter)$, then $S_{duplicator}$ execute $get\_cell_{qabq'R}(x_1,...,x_n;letter)$ but it can not execute $set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)$, because the condition is not verified, so there is not simulation.
If M has a transition $q \overset{a/bR}{\longrightarrow} q'$ and q existential, $S_{spoiler}$ has transitions $l_\forall \overset{\text{true},!m()}{\longrightarrow} l_\exists \xrightarrow{\text{true} \ | \ get\_cell_{qabq'R}(x_1,...,x_n;letter)} l_{qbdq'} \xrightarrow{\text{true} \ | \ set\_cell_{qabq'R}(x_1,...,x_n,b)}$ $l'_{qbdq'} \xrightarrow{(x_1=1\wedge...\wedge x_n=1) \ | \ NEXT(x_1,...,x_n;x_1,...,x_n)} l_\forall$. $S_{duplicator}$ has transitions

$l_q \xrightarrow{\text{true} \;|\; !m()} choice_{qbdq'}) \xrightarrow{\text{true} \;|\; get\_cell_{qabq'R}(x_1,...,x_n;letter)}$
$l'_{qbdq'} \xrightarrow{letter=a \;|\; set\_cell_{qabq'R}(x_1,...,x_n,b)} l''_{qbdq'} \xrightarrow{(x_1=1\wedge...\wedge x_n=1) \;|\; NEXT(x_1,...,x_n;x_1,...,x_n)} l_{q'}$.
So the difference with $S_{spoiler}$ is that, $S_{duplicator}$ can only execute the transition $l'_{qbdq'} \xrightarrow{letter=a \;|\; set\_cell_{qabq'R}(x_1,...,x_n,b;\emptyset)} l''_{qbdq'}$ if $x_1...x_n$ points on a tuple with value of W=a. Assume that the condition is verified. If $S_{spoiler}$ does not execute the transition with sending message, it looses the simulation, because $S_{duplicator}$ can execute the transition to l_{copy}. Assume that $S_{spoiler}$ sends the message $m()$, then $S_{duplicator}$ also sends the message. At this step $S_{spoiler}$ reaches the state $l_\exists$, if it chooses a transition different from $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$, $S_{duplicator}$ wins the game by choosing the transition which reaches the state l_{copy}. If $S_{spoiler}$ chooses to execute $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$, then $S_{duplicator}$ executes $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$ and it can execute $set\_cell_{qabq'R}(x_1, ..., x_n, b; \emptyset)$ and the game continue. Now, assume that the condition is not verified, if $S_{spoiler}$ at state $l_\forall$ does not choose sending the message it loose the simulation. Assume that it sends the message and reaches $l_\exists$, then $S_{duplicator}$ also sends the message. Then if $S_{spoiler}$ choose to execute any transition with label different from $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$, $S_{duplicator}$ wins the game by choosing the transition which reaches the state l_{copy}. If $S_{spoiler}$ chooses to execute $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$, then $S_{duplicator}$ executes $get\_cell_{qabq'R}(x_1, ..., x_n; letter)$ but it can not execute $set\_cell_{qabq'R}(x_1, ..., x_n, b; \emptyset)$, because the condition is not verified, so there is not simulation. Because when $S_{spoiler}$ send the message $m()$, there is many transition at state l_q which can send the message $m()$, there is no simulation if all transition of l_q labelled with sending $m()$ will block. □

Proofs of Chapter 3

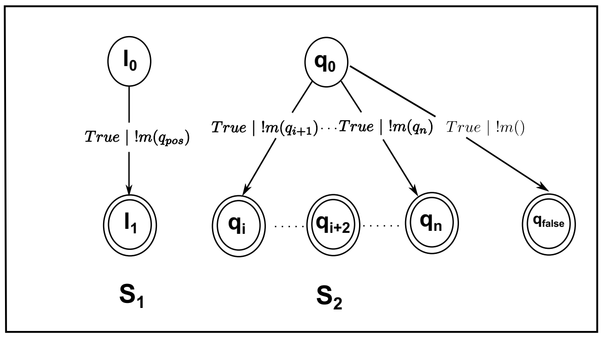


FIGURE A.1 – connection between simulation and the language $\mathcal{L}_S$

Lemma 27. *Let P be a formula expressed in the language $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$, where $\mathcal{L}_S$ is a boolean query language. Then, there exists two $(\emptyset, \mathcal{L}_S, \emptyset)$ services S_1 and S_2 such that the formula P is satisfiable iff $S_1 \not\preceq S_2$*

Démonstration. the proof follow the same spirit as the proof of lemma 18. It is based on a reduction from the problem of testing satisfiablity of a formula to the problem of checking simulation between two $(\emptyset, \mathcal{L}_S, \emptyset)$ services. P is of the form $q_1() \wedge q_2()... \wedge q_i() \wedge \neg q_{i+1}() \wedge ....\neg q_n()$ where each q_j where $j \in [1, n]$ is a boolean query expressed in the language $\mathcal{L}_S$. For $k \in [1, i]$, q_k is a positive boolean query of the form $q_k()$:-$body_k$. We construct the boolean query q_{pos}=$\bigwedge_{k\in[1,i]} body_k$. q_{pos} still a boolean query expressed in the language $\mathcal{L}_S$. The figure A.1 depicts the test of simulation constructed, where the service S_1 sends the message $m(q_{pos})$. The service S_2 will have $n-i+1$ transitions (i.e. the number of negated queries plus one), where each transition is labbeled with $m(q_k)$ with $k \in [i+1, n]$. The last transition of S_2 is labbeled with $!m()$ (i.e., send the empty message). Hence $S_1 \not\preceq S_2$ iff there exists an instance I such that $I \models q_{pos}$ and for each $k \in [i+1, n]$ $I \not\models q_k$. So, the formula P is satisfiable. Hence, Simulation in $(\emptyset, \mathcal{L}_S, \emptyset)$ services is undecidable if satisfiability in $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable. Note that the transition labbeled with $!m()$ is used to handle the case where S_1 chooses an instance $I \not\models q_{pos}$ and $I \models q_k$ for $k \in [i+1, n]$. □

Theorem 10 *Simulation in $(\emptyset, \mathcal{L}_S, \emptyset)$ services is undecidable if satisfiability of formula in $\mathcal{L}_S \cup \{\wedge^b, \neg^b\}$ is undecidable, where $\mathcal{L}_S$ is a boolean query language.*

Démonstration. The theorem is a direct consequence of lemma 27. □

Complexity of the simulation for $(\emptyset, \mathcal{L}_{LP}, \emptyset)$

Problem 5. **CO-SEND** $(\emptyset, \mathcal{L}_{LP}, \emptyset)$
Inputs : two $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ services S and S'. *Question* : $S \not\preceq S'$?

Proposition 4 *the problem* CO-SEND *$(\emptyset, \mathcal{L}_{LP}, \emptyset)$ is in Σ_2^p*

Démonstration. Let S and S' be two $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ services and let $\mathcal{P}_\mathcal{Q}$ be the set of partitions of the propositional logic queries (i.e, boolean queries containing only constants) used in S or in S'. Given a partition $p \in \mathcal{P}_\mathcal{Q}$ and an oracle checking the satisfiability of partitions of $\mathcal{P}_\mathcal{Q}$, it is possible to check in polynomial time whether S is not simulated by S'. Indeed, it is sufficient to check the consistency of the partition p and then check the simulation between the two finite state machines $FSM_p(S)$ and $FSM_p(S')$. Because satisfiability of a propositional logic formula is NP-complete, CO-SEND $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ is in Σ_2^p. □

We will prove the NP-hardness of the problem CO-SIM $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ using a reduction from . The reduction is nearly the same as for the complexity of simulation for $(\mathcal{L}_{PL}, \emptyset, \emptyset)$ services. Starting from an instance φ of 3-SAT problem using n boolean variables $\{x_1, ..., x_n\}$, we construct a test of simulation between two $(\emptyset, \mathcal{L}_{PL}, \emptyset)$ services namely $S_{spoiler}$ and $S_{duplicator}$.

Lemma 28. *Given a* 3-SAT *problem instance with n boolean variables, the problem has a solution iff $S_{3SAT-spoiler} \not\preceq S_{3SAT-duplicator}$, where $S_{3SAT-spoiler}$ and $S_{3SAT-duplicator}$ are $(\emptyset, \mathcal{L}_{PL}, \emptyset)$ services.*

Démonstration. Let the formula φ be a 3-SAT problem instance with n variables. The idea of the proof is that, $S_{3SAT-spoiler}$ will have a transition which sends the message $m(q_\varphi)$. q_φ is a boolean query having φ as body. $S_{3SAT-duplicator}$ will have a transition which sends the empty message $!m()$. Hence, if there exists a database instance I over $\mathcal{W}_g$ such that φ is true then $S_{3SAT-spoiler}$ sends *true* in the message $m()$ and $S_{3SAT-duplicator}$ sends m empty. Then, there is no simulation. Now we will detail the construction.
$\mathcal{W}_g$ will contain n boolean relational schema $\{R_1, ..., R_n\}$. An instance I over $\mathcal{W}_g$ corresponds to a set of instance $\{I_{R_1}, ..., I_{R_n}\}$. The relations R do not have any attribute, and there are only two possible instance : one containing the empty tuple, then we say R is evaluated to true, the other instance is empty then we say R is evaluated to false.

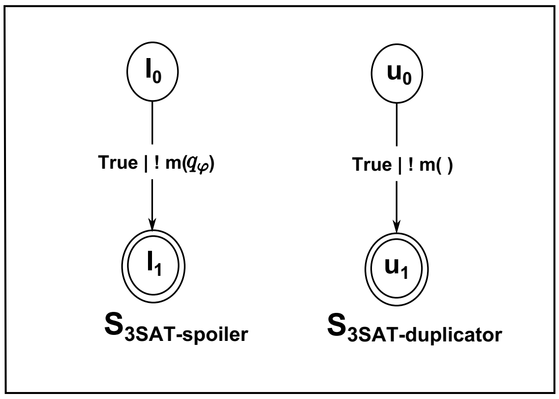


FIGURE A.2 – Reduction from 3-SAT problem to simulation.

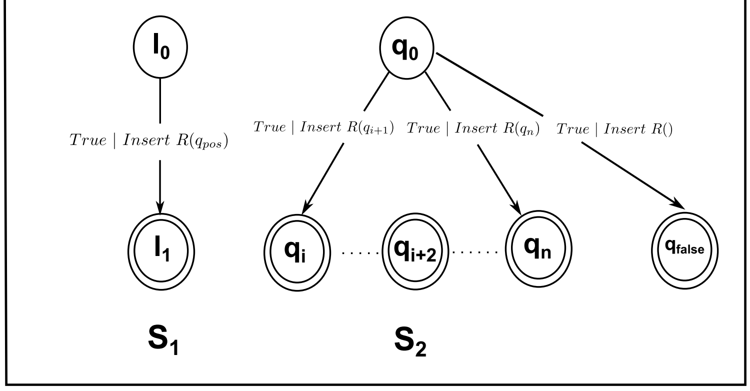
$S_{3SAT-spoiler}$ will contain only one transition $(l_0, True, !m(q_\varphi), l_1)$ and $S_{3SAT-duplicator}$ contains one transition $(u_0, True, !m(), u_1)$. A literal x_i (or its negation $\neg x_i$) in φ will be transformed to $R_i()$ ($\neg R_i()$) in q_φ.
If there exists an instance $I \models \varphi$ (hence, 3-SAT has a solution), then $S_{3SAT-spoiler}$ sends the message $m(true)$ while $S_{3SAT-duplicator}$ sends the empty message $!m()$. Hence, $S_{3SAT-spoiler} \not\preceq S_{3SAT-duplicator}$. If 3-SAT does not accept any solution, there is no database instance I which satisfies the query q_φ, hence there is simulation. The figure A.2 depicts the test of simulation between $S_{3SAT-spoiler}$ and $S_{3SAT-duplicator}$. □

Theorem 12 CO-SEND $(\emptyset, \mathcal{L}_{LP}, \emptyset)$ is NP-HARD

Démonstration. From lemma 28 and knowing that the 3-SAT problem is NP-HARD [Coo71]. □

connection between simulation for $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ and satisfiability of $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$

Lemma 29. *Let P be a formula expressed in the language $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$, where $\mathcal{L}_I$ is a boolean query language. Then, there exists two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services S_1 and S_2 such that the formula P is satisfiable iff $S_1 \not\preceq S_2$.*



l_0
$True \mid Insert\ R(q_{pos})$
l_1
q_0
$True \mid Insert\ R(q_{i+1})$
$True \mid Insert\ R(q_n)$
$True \mid Insert\ R()$
q_i
q_{i+2}
q_n
q_{false}
S_1
S_2

FIGURE A.3 – Connection between simulation of $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services and the language $\mathcal{L}_I$

Démonstration. the proof follow the same spirit as the proof of lemma 18. It is based on a reduction from the problem of testing satisfiablity of a formula to the problem of checking simulation between two $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services. P is of the form $q_1() \wedge q_2() ... \wedge q_i() \wedge \neg q_{i+1}() \wedge .... \neg q_n()$ where each q_j with $j \in [1, n]$ is a boolean query expressed in the language $\mathcal{L}_I$. For $k \in [1, i]$, q_k is a positive boolean query of the form $q_k()$:-$body_k$. We construct the boolean query q_{pos}=$\bigwedge_{k \in [1,i]} body_k$. q_{pos} still a boolean query expressed in the language $\mathcal{L}_I$. The figure A.3 depicts the test of simulation constructed, where the service S_1 inserts the result of $q_{pos}()$ in R. The service S_2 will have $n - i + 1$ transitions (i.e. the number of negated queries plus one), where each transition inserts the result of $q_k()$ in R with $k \in [i + 1, n]$. The last transition of S_2 inserts nothing in R. Hence $S_1 \not\preceq S_2$ iff there exists an instance I such that $I \models q_{pos}$ and for each $k \in [i+1, n]$ $I \not\models q_k$. So, the formula P is satisfiable. Hence, Simulation in $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ services is undecidable if satisfiability in $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ is undecidable. Note that the last transition of S_2 is used to handle the case where S_1 chooses an instance $I \not\models q_{pos}$ and $I \models q_k$ for $k \in [i + 1, n]$ or an instance of R containing $True$. □

Theorem 13 *Simulation in* $(\emptyset, \emptyset, \mathcal{L}_U^{insert(\mathcal{L}_I)})$ *services is undecidable if checking satisfiability of formulas in* $\mathcal{L}_I \cup \{\wedge^b, \neg^b\}$ *is undecidable.*

Démonstration. From lemma 29. □

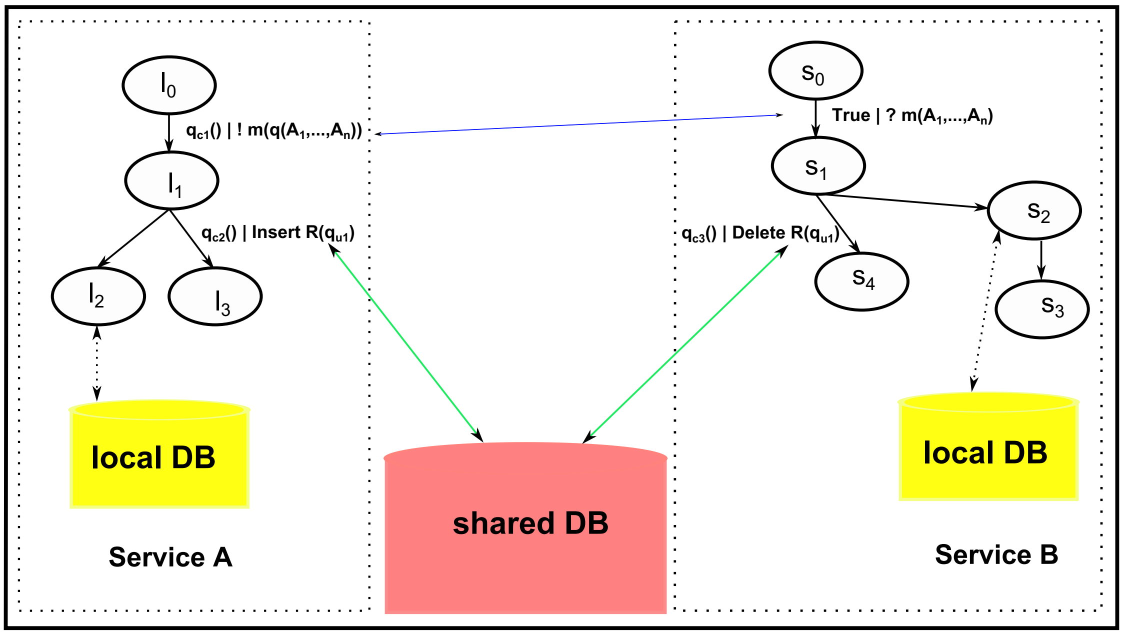


FIGURE 1.1 – Generic web service framework.

(e.g., the service A sends a message m, which contains a result of the query q). A query q can be defined over the local as well as the shared database. The transitions are guarded by boolean queries (q_{ci}). Finally, services can modify the databases using *update queries* (e.g., the service A inserts the result of the query q_{u1} into the relation R). In this thesis, we focus our attention on insert queries and we do not consider delete and modify queries.

In order to isolate and study the impact of the different parameters of the generic model on the simulation preorder, we investigate the decidability and complexity issues of the simulation for various classes of our generic model. Each class is characterized by :
- the type of actions supported by the model, e.g., the service can only send messages, or only insert in the database, ... etc,
- the languages used to instantiate respectively $\mathcal{L}_T$, $\mathcal{L}_U$ and $\mathcal{L}_S$,
- the presence or not of the local database (i.e., in the presence of local database, we study weak simulation).

Table 1.2 summarizes the considered sub-classes of the generic model as well as the obtained results. We consider more precisely the following classes :
- Update-free services. This class represents services which are not able to make modifications over the databases. The class of update-free service is decomposed into two sub-classes :
  - Guarded services, this class enables to focus on the role played by the language of guards ($\mathcal{L}_T$) on the decidability of the simulation relation. Our main result regarding this class lies in a full characterization of the decidability of simulation in terms of the decidability of checking satisfiability of formulas expressed in the language $\mathcal{L}_T$ augmented with a restricted form of negation. We denote this language $\mathcal{L}_T \cup \{\wedge^b, \neg^b\}$ (i.e., the conjunction and negation is applied on boolean $\mathcal{L}_T$ formulas). As for the case of $Colombo^{DB=\emptyset}$, we use a finite symbolic representation of update-free services by partitioning the original infinite state space into a finite number of equivalence classes.
  - Send services. This class represents update-free services which send the results of

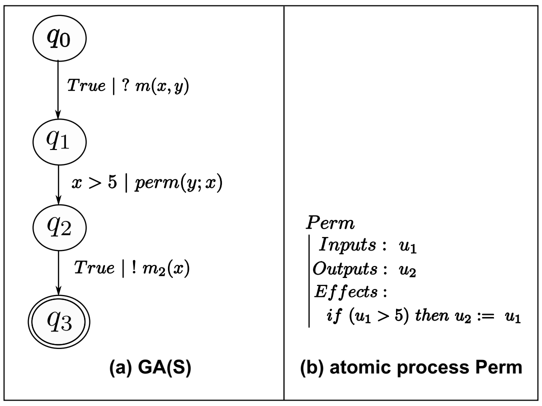


Figure 2.11 – A $Colombo^{db=\emptyset}$ service S.

Hence, the set of elementary intervals over K is :

$$I_K = \{[\omega,\omega],]-\infty,5[, [5,5],]5,+\infty[\}$$

while the set $R_g(X,K)$ includes, among others, the following regions :
- $r_\omega = ([\omega],[\omega],\{x=\omega, y=\omega\}$
- $r_1 = ([\omega],]-\infty,5[, \{x=\omega, y=y\}$
- $r_2 = (]5,+\infty[,]5,+\infty[, \{y<x\}$
- $r_3 = (]5,+\infty[,]5,+\infty[, \{y=x\}$
- $r_4 = (]5,+\infty[,]5,+\infty[, \{x<y\}$
- $r_5 = (]5,+\infty[, [5,5], \{y<x\}$
- $r_6 = (]5,+\infty[,]-\infty,5[, \{y<x\}$
- ...

The corresponding region automaton R^S is depicted at figure 2.12. The initial state of R^S is made of the pair (q_0, r_ω). We illustrate below the cases (a), (b) and (c) of definition 6 on this region automaton.
- the transition $(q_0, ?m1(x,y), q_1)$ of $GA(S)$ (c.f., figure 2.11), is translated into a set of transitions $((q_0,r_\omega), ?m1(x,y), (q,r))$ with $r \in R_g(X,K)$ (case (b) of definition 6). This captures the fact that on a reception of a message $?m1(x,y)$, any new values may be associated to the variables x and y.
- the transition $(q_1, x > 5 \mid Perm(y;x), q_2)$ of $GA(S)$, enables to a create new transition from the state (q_1,r_2) of R^S as illustrated below :
  - $((q_1,r_2), Perm(y;x), (q_2,r_3))$, this is because the region r_2 satisfies both the guard $x > 5$ of the transition and the condition $u_1 > 5$ of the atomic process (case (c-1) of definition 6). Hence, in this case the atomic process Perm is executed. The atomic process Perm assigns variable y to the variable x, hence the region automata moves to a region where $\tau_x := \tau y$ and requires to have $x = y$ in the associated v-order. In our example, region r_3 satisfies both conditions.
  - $((q_1,r_5), Perm(y;x), (q_2,r_5))$, this is because the region r_5 satisfies the guard $x > 5$ of the transition but does not satisfy the condition $u_1 > 5$ of the atomic process Perm (case (c-2) of definition 6). According to the Colombo semantics, the transition is fired but the atomic process Perm execute a *no-op* operation (no operation). As a consequence, the region automata moves to state q_2 while staying in the same region r_5.
- the transition $((q_2,r_5), !m2(x,), (q_3,r_5))$ (case (a) of definition 6). A send of a message does not modify values of the variables, hence upon sending the message $!m2(x)$,

the region automaton R^S moves into a new state (q_3, r_5) while staying in the same region r_5.

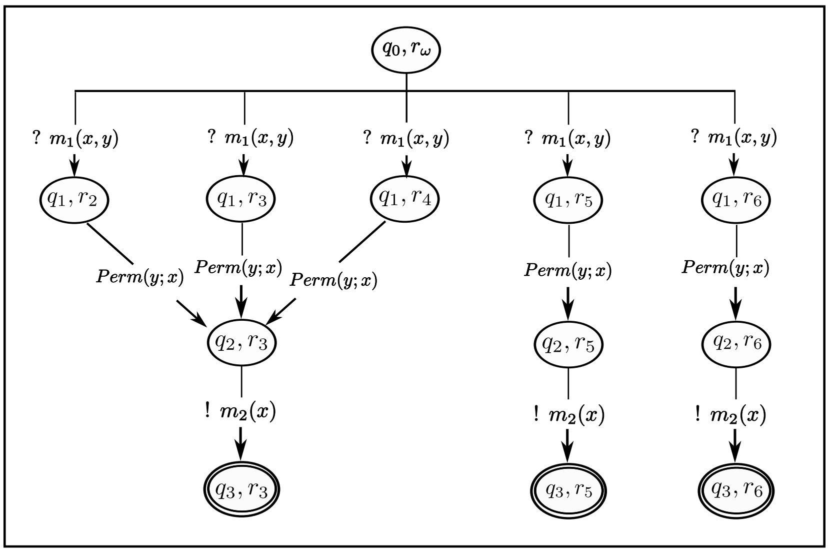


FIGURE 2.12 – A region automaton R^S.

In the following we show that the region automata R^S constitutes a compact representation of the extended state machine of $E(S)$ and hence it faithfully abstracts the original Colombo service S. To do so, we define the notion of unfolding of a region automaton $U_{nfold}(R^S)$ as given below.

Definition 7. *(unfolding of region automata) Let* $R^S = (Q^S, q_0^S, F^S, \delta^S, R_g(X,K))$ *be a region automata of a service* S. *The associated extended state machine, noted* $U_{nfold}(R^S)$, *is a tuple* $U_{nfold}(R^S) = (\mathbb{Q}^g, \mathbb{Q}_0^g, \mathbb{F}^g, \Delta^g)$ *where :*

- $\mathbb{Q}^g = \bigcup_{r \in R_g(X,K)} \{(q,\alpha)\ s.t\ (q,r) \in Q^S, \alpha \in r\}$.
- $\mathbb{Q}_0^g = \{(q_0, \alpha_w)\}$, *with* $\alpha_w(x) = \omega$, $\forall x \in LStore(S)$.
- $\mathbb{F}^g = \bigcup_{r \in R_g(X,K)} \{(q,\alpha)\ s.t\ (q,r) \in F^S, \alpha \in r\}$..
- $\forall (q,r) \xrightarrow{\mu_i} (q',r') \in \delta^S$, *a new transition* $(q,\alpha) \xrightarrow{\mu_i} (q',\alpha')$ *is added to* Δ *such that* $\alpha \in r, \alpha' \in r'$ *and :*
  (a) *if* $\mu = !m(v_1, \ldots, v_m)$, *then* $\alpha' = \alpha$.
  (b) *if* $\mu = ?m(v_1, \ldots, v_m)$ *then* $\forall x \in LStore(S) \setminus \{v_1, \ldots, v_m\}$, *we have* $\alpha'(x) = \alpha(x)$.
  (c) *If* $\mu = p(u_1, \ldots, u_n; v_1, \ldots, v_m, \{c, E\})$, *we have two cases :*
  (c-1) *if* $r \wedge \theta \wedge c$ *is consistent then* $\forall x \in LStore(S) \setminus \{v_1, \ldots, v_m\}$, *we have* $\alpha'(x) = \alpha(x)$ *and for each* $i \in [1,m]$, *we have :*
    – *If* $v_i := k \in E$, *with* $k \in \mathcal{D} \cup \{\omega\}$, *then* $\alpha'(v_i) = k$
    – *If* $v_i := u_j \in E$ *then* $\alpha'(v_i) = \alpha(u_i)$
  (c-2) *if* $r \wedge \theta \wedge \neg\ c$ *is consistent, then* $\alpha' = \alpha$.

A run of $U_{nfold}(R^S)$ is any finite path from an initial configuration of $E(RS)$ to one of its final configurations.

Example 18. Figure 2.13(b) depicts part of the extended automata obtained by unfolding the region automata of figure 2.13(a) which corresponds to a fragment of the region automata of figure 2.12.

Alternating Turing machine M An alternating Turing machine M [CKS81] is a tuple $(\mathbb{Q}, q_0, \Gamma, \delta, mode)$ where :

- $\mathbb{Q}$ is the set of control states.
- q_0 is the initial state.
- Γ is the set of tape symbols.
- $mode : \mathbb{Q} \longrightarrow \{\forall, \exists,$ accept , reject $\}$ is the labelling function of control state.
- $\delta : \mathbb{Q}$ x $\Gamma \longrightarrow \mathcal{P}(\mathbb{Q}$ x Γ x $\{L, R\})$.

A configuration C of M is of the form $y_1, ..., qy_j, ..., y_n$, where q is a state of the machine, and the head points actually on the j'th letter on the tape (i.e., y_i are the letters of the word on the tape). A transition $qa \longrightarrow bRq^{'}$ is applicable from a configuration C if the letter pointed by the head is equal to a ($y_j{=}a$), then the successor C' of C is equal to $y^{'}_1, ...y^{'}_j, q^{'} y^{'}_{j+1}, ..., y^{'}_n$ s.t $y_k{=}\ y^{'}_k$ for k $\in$ [1,n] and k $\neq$ j and $y^{'}_j = b$. We note this step $C \overset{qa/bRq^{'}}{\longrightarrow} C'$ or $(y_1, ..., qy_j, ..., y_n) \overset{qa/bRq^{'}}{\longrightarrow} (y^{'}_1, ...y^{'}_j, q^{'} y^{'}_{j+1}, ..., y^{'}_n)$. The machine M starts on $C_0 = qy_1, ..., y_n$, where $y_i{=}w_i$, the i'th letter of the input word w.

The definition of acceptance of an alternating Turing machine is recursive :

- If the configuration C is in an accepting control state q, then C is accepting.
- If the configuration C is in an rejecting control state q, then C is rejecting.
- If the configuration C is in a universal control state q, then C is accepting if all the configurations reachable from C in one step are accepting and rejecting if some configurations reachable from C in one step are rejecting.
- If the configuration C is in an existential control state q, then C is accepting if some configurations reachable in one step are accepting and rejecting when all configurations reachable in one step are rejecting (the case of classical non-deterministic Turing machine correspond to an alternating machine where all states are existential).

M is said to accept an input word w if the initial configuration of M is accepting, and to reject w if the initial configuration is rejecting. A configuration reachable in one step from configuration C is called a *successor* of C and the set of *successors* of C is denoted $successors(C)$.

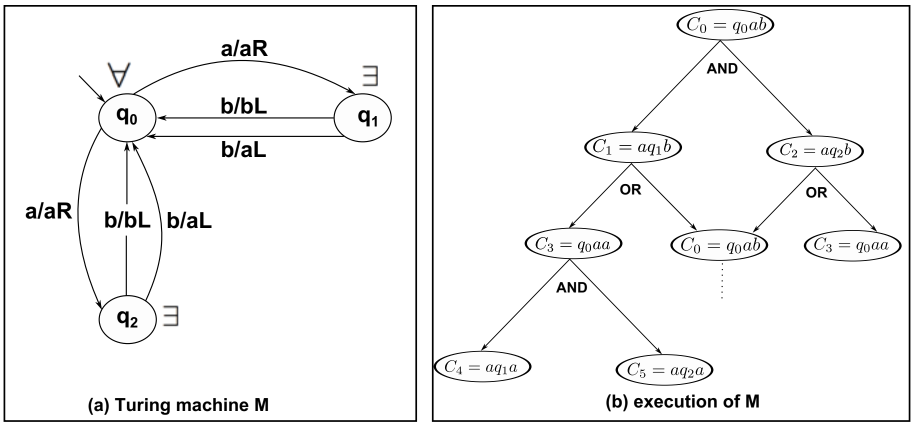


Figure 2.14 – Alternating Turing machine M.

We consider the problem of the existence of an infinite execution of an alternating Turing machine M on an input word $w = y_1, ..., y_n$, where y_i's are letters from Γ. That is given a word w as input, M can make choices of existential transitions such that whatever the transitions chosen by universal states the machine continues the execution. Assume

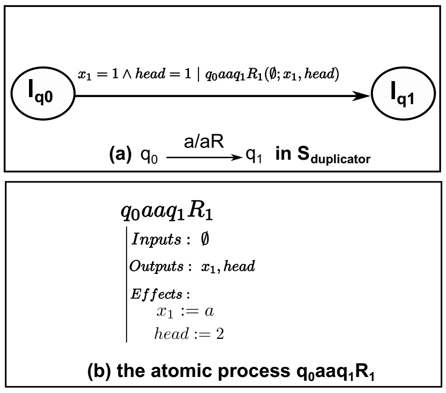

l_{q0}
$x_1 = 1 \wedge head = 1 \mid q_0aaq_1R_1(\emptyset; x_1, head)$
l_{q1}
(a) $q_0 \xrightarrow{a/aR} q_1$ in $S_{duplicator}$
$q_0aaq_1R_1$
Inputs : $\emptyset$
Outputs : $x_1, head$
Effects :
$x_1 := a$
$head := 2$
(b) the atomic process $q_0aaq_1R_1$

Figure 2.15 – A transition in $S_{duplicator}$ corresponding to a transition of M.

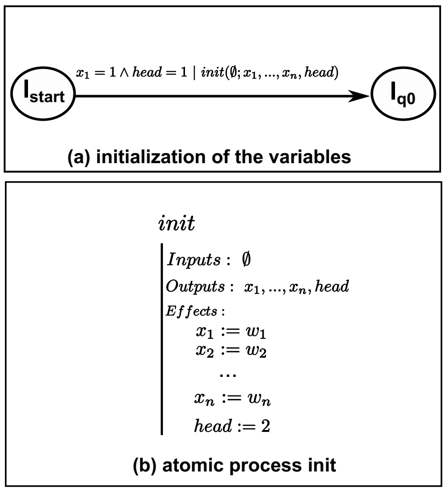

l_{start}
$x_1 = 1 \wedge head = 1 \mid init(\emptyset; x_1, ..., x_n, head)$
l_{q0}
(a) initialization of the variables
init
Inputs : $\emptyset$
Outputs : $x_1, ..., x_n, head$
Effects :
$x_1 := w_1$
$x_2 := w_2$
...
$x_n := w_n$
$head := 2$
(b) atomic process init

Figure 2.16 – initialization of variables in $S_{duplicator}$.

Example 21. Figure 2.16 depicts the initialization of the service $S_{duplicator}$ corresponding to the machine M of the example 19, where x_1 :=a, x_2 := b and $head$:=1.

Before giving the construction of $S_{duplicator}$, we need to introduce some notations :

– $\mathcal{P}$ is the set of all atomic processes used to encode actions of the machine M, it contains the following sets :
  – $\{qabq'R_i(\emptyset; x_i, head) \mid q \xrightarrow{a/bR} q' \;\; in \;\; M \;\; and \;\; i \in [1, n-1]\}$
    For each transition of the machine labelled with a move to the right, we create n-1 atomic processes to encode it.
  – $\{qabq'L_i(\emptyset; x_i, head) \mid q \xrightarrow{a/bL} q' \;\; in \;\; M \;\; and \;\; i \in [2, n]\}$
    For each transition of the machine labelled with a move to the left, we create n-1 atomic processes to encode it.

  The atomic process $qabq'R_i(\emptyset; x_i, head)$ has no condition, it assigns to x_i the value b and increments the head. The atomic process $qabq'L_i(\emptyset; x_i, head)$ has no conditions, it assigns to x_i the value b and decrements the head.
– g_i^a is a condition of the form x_i =a $\wedge$ $head$= i. It will be used as guard on transitions of $S_{duplicator}$.

The incrementation is not allowed in the definition of the Colombo model. When defining the effects of the atomic process, we write the result of the sum rather than the operation of incrementation. For example, in the atomic process $qabq'R_1(\emptyset; x_1, head)$, Note that, if the machine reads or writes the special blank character B during a transition, then we replace the constants a,b by the special symbol ω, in the construction of the corresponding transition.

$S_{duplicator}$ starts by initializing the variables representing the cells with the input word. If M has a transition $q \overset{a/bR}{\longrightarrow} q^{'}$ and q is a universal state, then the service contains $n-1$ transitions from l_q to $l_{q^{'}}$ labelled with condition/action : if x_i=a and the head points on i then we can execute the atomic process which modifies x_i to b and increments the head. So, $S_{duplicator}$ can only execute the atomic process representing the transition $q \overset{a/bR}{\longrightarrow} q^{'}$ if the actual value of x_i=a and the head points on i. Note that, for any actual valuation of variables, there is only one transition from the "n-1" transitions which can be executed. This is due to the guards where several x_i can verify the condition but the head points only to one cell.
If q is an existential state, then $S_{duplicator}$ sends a message m before executing the atomic process. The state l_{copy} contains a set of self loop labelled with all atomic processes $\mathcal{P}$ and $!m()$ (if $S_{duplicator}$ reaches this state, it wins the simulation). All transitions which reach the state l_{copy} are used to prevent $S_{spoiler}$ from cheating during the test of simulation.

The next lemma asserts that each configuration of M on the input word w has a corresponding configuration in the extended state machine of $S_{duplicator}$. The proof is obtained by induction (details are given in appendix A).

Lemma 6. *Each configuration C of the execution of an alternating Turing machine M on an input w has a corresponding configuration in the extended state machine of $S_{duplicator}$.*

Example 22. The Figure 2.17 depicts the part of service $S_{duplicator}$ corresponding to the transition $q_0 \overset{a/aR}{\longrightarrow} q_1$ where q_0 is universal, and the two transitions $q_1 \overset{b/bL}{\longrightarrow} q_0$ and $q_1 \overset{b/aL}{\longrightarrow} q_0$ where q_1 is an existential state of the machine M of example 19.

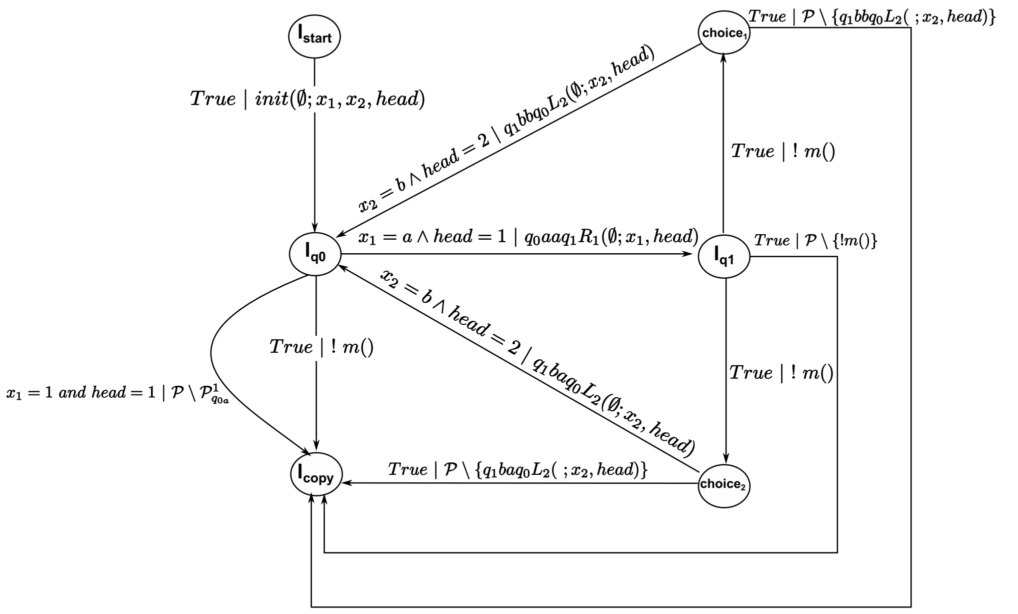

l_{start}
$True \mid init(\emptyset; x_1, x_2, head)$
$choice_1$
$True \mid \mathcal{P} \setminus \{q_1bbq_0L_2(; x_2, head)\}$
$x_2 = b \wedge head = 2 \mid q_1bbq_0L_2(\emptyset; x_2, head)$
$True \mid ! \; m()$
l_{q0}
$x_1 = a \wedge head = 1 \mid q_0aaq_1R_1(\emptyset; x_1, head)$
l_{q1}
$True \mid \mathcal{P} \setminus \{!m()\}$
$x_2 = b \wedge head = 2 \mid q_1baq_0L_2(\emptyset; x_2, head)$
$True \mid ! \; m()$
$True \mid ! \; m()$
$x_1 = 1 \; and \; head = 1 \mid \mathcal{P} \setminus \mathcal{P}^1_{q_{0a}}$
l_{copy}
$True \mid \mathcal{P} \setminus \{q_1baq_0L_2(; x_2, head)\}$
$choice_2$

Figure 2.17 – A part of the service $S_{duplicator}$.

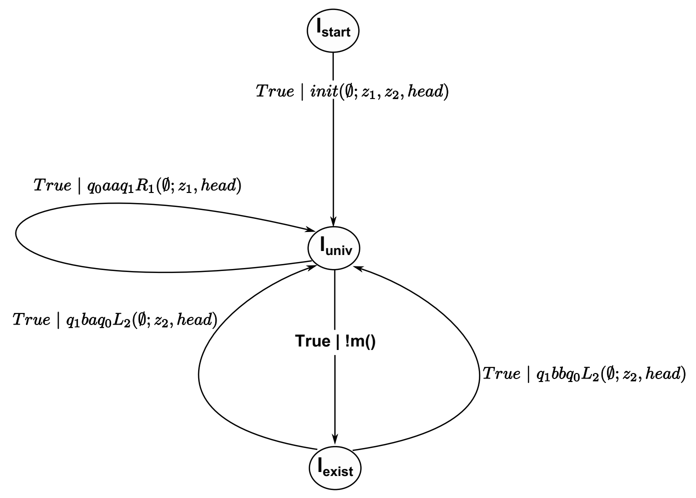

l_{start}
$True \mid init(\emptyset; z_1, z_2, head)$
$True \mid q_0aaq_1R_1(\emptyset; z_1, head)$
l_{univ}
$True \mid q_1baq_0L_2(\emptyset; z_2, head)$
True | !m()
$True \mid q_1bbq_0L_2(\emptyset; z_2, head)$
l_{exist}

Figure 2.18 – part of $S_{spoiler}$.

Lemma 8. *Given two DB-less Colombo services S, $S^{'}$, checking whether $S \preceq S^{'}$ is* EXPTIME-*hard.*

Hence, the following theorem can now be claimed from proposition 1 and lemma 8

Theorem 5. *Given two DB-less Colombo services S, $S^{'}$, checking whether $S \preceq S^{'}$ is* EXPTIME-*complete.*

2.5 Decidability of simulation in $Colombo^{bound}$

We study in this section the simulation problem in the setting of a Colombo model with a *bounded* global database (i.e., the size of the instance over $\mathcal{W}$ is at most equal to a constant k). Given two services S and $S^{'}$, S is *k-bounded simulated* by $S^{'}$ means that $S^{'}$ is able to reproduce the behavior of S on all executions where the size of the database is at most equal to k. We will prove that the simulation is decidable in this setting by providing a reduction to a test of simulation between two DB-less $Colombo^{DB=\emptyset}$ services. This is done by encoding the bounded database using a finite set of variables. First we start by giving the definition of *k-bounded extended state machines*, which is used to capture the notion of k-bounded simulation. Then we give the construction of the DB-less service and prove the equivalence of the two tests.

2.5.1 k-bounded extended state machine $E^k(S)$ and k-bounded simulation

Let k be an integer. We call a database instance I k-bounded if $|I| \leqslant k$. The k-bounded extended state machine $E^k(S)$ of a Colombo service S is the extended state machine $E(S)$ of S restricted to configurations having k-bounded instances.

Definition 9. *Let S be a Colombo service and $E(S) = (\mathbb{Q}, \mathbb{Q}_0, \mathbb{F}, \Delta)$ the associated extended state machine, then $E^k(S) = (\mathbb{Q}^k, \mathbb{Q}_0^k, \mathbb{F}^k, \Delta^k)$ is the k-bounded extended state machine of S where :*
– $\mathbb{Q}^k = \{(l, \mathcal{I}, \alpha) \mid (l, \mathcal{I}, \alpha) \in \mathbb{Q}$ *and* $|\mathcal{I}| \leq k\}$.

The k-bounded extended state machine of S is the part of $E(S)$ where all configurations contain only k-bounded databases. Like $E(S)$, a run σ of $E^k(S)$ is a finite sequence

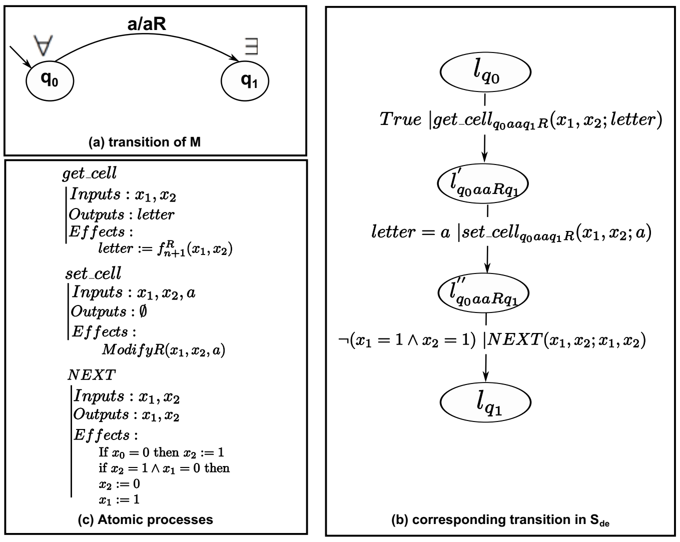


FIGURE 2.24 – transitions corresponding to $q_0 \xrightarrow{a/aR} q_1$ in M.

– finally moves to the next tuple by executing a binary addition on $x_1 ... x_n$.

Example 26. Figure 2.24(a) depicts a transition of the machine M of example 19. If the actual value of the cell pointed by the head is equal to a and the machine is in the state q_0, the machine writes a, and moves to the next cell and reaches the state q_1. Suppose the input word is ab, so n=2. The part of $S_{duplicator}$ representing this transition starts by storing the value of the attribute W corresponding to the tuple identified with the key x_1x_2 in the variable letter (i.e., $letter := f^R_{n+1}(x_1, x_2)$) using the atomic process *get_cell*. Then, the service tests if letter = a and writes in the current tuple the new value of W with *set_cell*. After that, the service increments the binary number x_1x_2 using the atomic process NEXT. As a consequence, x_1x_2 points on the next tuple. The guard $\neg(x_1 = 1 \wedge x_2 = 1)$ prevents a move to the right if the service points on the last cell. Note that, when encoding a transition of M, the service $S_{spoiler}$ will not contain the guard letter = a, because $S_{spoiler}$ will encode all transitions that the machine can do infinitely often.

The services $S_{duplicator}$ and $S_{spoiler}$ will start with an initialization part where they :

1. Check if all tuples identified with key from $(0, ..., 0)$ to $(1, ..., 1)$ contain the symbol B, which means the 2^n cells are empty. In following, we will call the database instances which satisfy this condition *standard* instances and those that do not satisfy it *non-standard* instances.
2. initialize the n first tuples with the n letters of the input word w.

Example 27. Continuing with our example, figure 2.25 depicts the initialization part of the two services. The services start by assigning zero to x_1 and x_2, then check if the value of the attribute W of the actual tuple identified with the key x_1x_2 is equal to B. If x_1x_2 points on an empty tuple and it is not the last tuple (key equal 11), the services increment the key and test the next tuple. If one of them does not contain B, then the database is *non-standard* and there is simulation. If all tuples ranged from 00 to 11 contain B, the services reinitialize the variables to zero.

For all executions starting with a *non-standard* database, $S_{spoiler} \preceq S_{duplicator}$ is true, because the two services have the same initialization part. Figure 2.26 depicts examples of

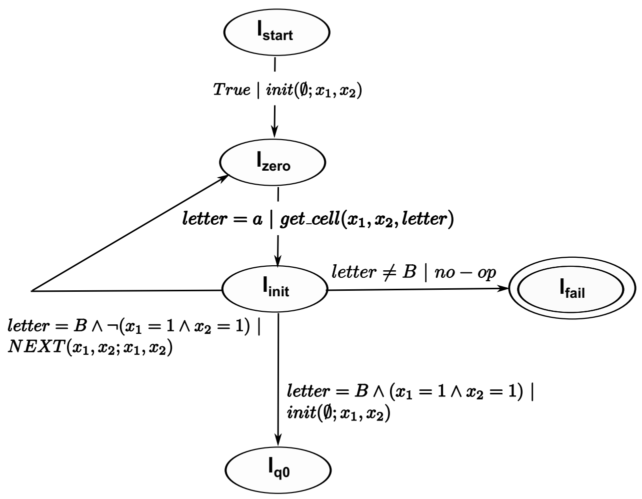
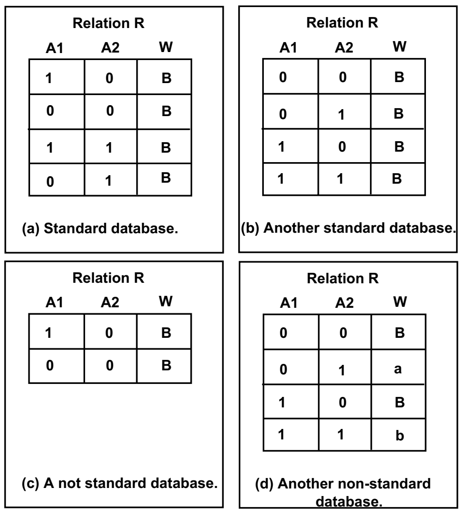

l_{start}
$True \mid init(\emptyset; x_1, x_2)$
l_{zero}
$letter = a \mid get\_cell(x_1, x_2, letter)$
l_{init}
$letter \neq B \mid no - op$
l_{fail}
$letter = B \wedge \neg(x_1 = 1 \wedge x_2 = 1) \mid NEXT(x_1, x_2; x_1, x_2)$
$letter = B \wedge (x_1 = 1 \wedge x_2 = 1) \mid init(\emptyset; x_1, x_2)$
l_{q0}

Figure 2.25 – initialization part of $S_{duplicator}$ and $S_{spoiler}$.

standard and *non-standard* databases. As we can see, the order of tuples is not important for standard databases (Figure 2.26(a) and figure 2.26(b)). The database depicted at figure 2.26(c) fails in the initialization part because $f_3^R(1,1)$ and $f_3^R(0,1)$ are equal to ω, and the database depicted at figure 2.26(d) is *non-standard* because there are tuples with values different from B for the attribute W.


Relation R

A1	A2	W
1	0	B
0	0	B
1	1	B
0	1	B

(a) Standard database.

Relation R

A1	A2	W
0	0	B
0	1	B
1	0	B
1	1	B

(b) Another standard database.

Relation R

A1	A2	W
1	0	B
0	0	B

(c) A not standard database.

Relation R

A1	A2	W
0	0	B
0	1	a
1	0	B
1	1	b

(d) Another non-standard database.

Figure 2.26 – Standard database.

Now we will give the formal definition of atomic processes.

Atomic processes $\mathcal{P}$ is the set of all atomic processes used to encode the actions of the machine M :

- for each transition $q \xrightarrow{a/bR} q'$ in M :
  - $get\_cell_{qabq'R}(x_1, ..., x_n; letter, CE)$ is an atomic process with one conditional effect :
    - θ : true.
    - ev : letter $:= f_{n+1}^R(x_1, ..., x_n)$.
  - $set\_cell_{qabq'R}(x_1, ..., x_n, b)$ is an atomic process with one conditional effect :