Constructive Verification for Component-based Systems

Thanh-Hung Nguyen

Univeristé de Grenoble

T H È S E

pour obtenir le grade de

Docteur de l'Université de Grenoble

Spécialité : Informatique

préparée au laboratoire VERIMAG

dans le cadre de l'**École Doctorale Mathématiques, Sciences et Technologies de L'Information, Informatique**

présentée et soutenue publiquement par

Thanh-Hung NGUYEN

le 27 Mai 2010

(DRAFT)

Vérification Constructive des Systèmes à base de Composants (Constructive Verification for Component-based Systems)

JURY

Président	Roland GROZ	INP Grenoble, France
Rapporteurs	Parosh Aziz ABDULLA	Uppsala University, Sweden
	Ahmed BOUAJJANI	University of Paris 7, France
	Shankar NATARAJAN	SRI's Computer Science Laboratory, USA
Examinateurs	Marius BOZGA	CNRS, Verimag Laboratory , France
	Klaus HAVELUND	Jet Propulsion Laboratory, USA
Directeurs de thèse	Saddek BENSALEM	UJF, Verimag Laboratory, France
	Joseph SIFAKIS	CNRS, Verimag Laboratory, France

Constructive Verification for Component-based Systems
Vérification Constructive des Systèmes à base de Composants

Abstract

The goal of the thesis is to develop theory, methods and tools for the compositional and incremental verification for component-based systems. We propose a compositional verification method for proving safety properties. The method is based on the use of two kinds of invariants: *component invariants* which express local aspects of systems and *interaction invariants* which characterize global constraints induced by synchronizations between components. We provide efficient methods for computing these invariants. We also propose a new technique that takes the incremental design of the system into account. The integration of verification into design process allows to detect an error as soon as it appears. Moreover, the technique helps to avoid redoing all the verification process by reusing intermediate verification results. It takes advantage of the system structure for coping with complexity of the global verification and therefore, reduces significantly the cost of verification in both time and memory usage. The methods have been implemented in D-Finder tool-set. The experimental results obtained on non trivial examples and case studies show the efficiency of our methods as well as the capacity of D-Finder.

Key words: BIP, compositional verification, incremental verification, incremental design, invariant, deadlock detection, static analysis, D-Finder.

Associated Papers

Several Chapters in this thesis appeared in several papers in form of articles or of Verimag technical reports.

The compositional method in Chapter 2 appeared in the paper [BBNS08] in ATVA 2008 (6th International Symposium on Automated Technology for Verification and Analysis, October 20-23, 2008, Seoul, South Korea). A journal version [BBNS10] has been accepted for publication in IET Software Journal, Special Issue on Automated Compositional Verification: Techniques, Applications, and Empirical Studies, 2010.

The incremental construction and verification methods together with the results in Chapter 3 appeared in two papers: paper [BBL+09] has been accepted for TASE 2010 (4th IEEE International Symposium on Theoretical Aspects of Software Engineering) and paper [BLN+10] has been submitted to FMCAD 2010 (Formal Methods in Computer Aided Design).

The D-Finder tool-set in Chapter 5 appeared in the tool paper [BBNS09] in CAV 2009 (21st International Conference on Computer Aided Verification, June 26 - July 02, 2009, Grenoble, France).

An application of the method on a robotic system presented in Chapter 6 appeared in the paper [BGL+08] in ECAI 2008 (18th European Conference on Artificial Intelligence, July 21-25, 2008, Patras, Greece) and then the journal version [BGI+09] is published in Special Issue on Software Engineering for Robotics of the IEEE Robotics and Automation Magazine Vol. 16, No. 1, Pages 67-77, March 2009.

Contents

List of Figures

List of Tables

Part I

Context

Introduction

Problems and Needs

Computer science nowadays plays an important role in the development of science and of society. Despite of being a new domain, it appears almost everywhere, from small systems such as a cell phone, a music player to huge systems such as a plane, a spacecraft. The increasing need of computer systems increases also the need of their reliability, correct behavior, etc.

Constructing correct systems is an essential requirement crossing all areas. A system is correct if it behaves following an expected manner or in other words, it satisfies an explicit set of requirements. The expected behavior is called *formal specification* of the system and is often expressed as a set of desired requirements. The *correctness* of a system is assigned if the system is correct with respect to its specification. Ensuring the correctness is specially important for critical systems since their failure or malfunction may cause a catastrophe such as loss of human life, high economical costs or environmental harm. For example, the crash of Ariane 5, an European expendable launch system, is one of the most infamous computer bugs in history and resulted in the loss of more than 370 millions dollars.

Unfortunately, with the growing of the demand for scalability and complexity of systems, it becomes more and more difficult to design correctly their models. The scale and complexity not only increases potential violations of desired properties but also makes them harder to detect and to handle. The construction of a system that operates reliably despite of complexity is highly desirable but also not always feasible. Therefore the check of the correctness of the system is essential and important to ensure that all requirements are respected. The check process is based on techniques for detecting property violations of the model and the correctness is achieved by the absence of such violations.

There are two main approaches for detecting property violations of a system: *formal testing* and *formal verification*. Consider a model of a system, an environment in which the system interacts, and some properties that the designed system is expected to guarantee, one can choose one of the following approaches depending on their goal:

- **Formal testing** [Bei90, Tre90, Mye04, FFMR07] is a method used to find defects on a system implementation, either during the development or after the complete construction of the system. To do that, testing generates some inputs from environment (test cases) and executes the system to determine whether it produces the required results. Testing, depending on method employed, can be used at any time of development process. It is a quick and direct way to detect bugs or violations in the system. However, testing is not capable of covering all the possibilities that may happen while running the system in reality. The number of possible situations is usually so large that we can test a tiny proportion of them. The absence of property violations provided by testing does not imply the correctness of the system.

- **Formal verification** [UP83, BM79, QS82, CE81] can both search for input patterns which violate the desired properties or prove the correctness of the system if such input patterns do not exist. In contrast to formal testing, formal verification covers all the possibilities that the system can behave, hence it proves the correctness of the system in the case of the absence of property violation. It relies on the use of mathematical techniques to prove or disprove the correctness of a design with respect to a certain formal specification. Formal verification has been successfully applied to verify both software and hardware systems. The verification of these systems is done by providing a formal proof on an abstract mathematic model of the system. The mathematic objects that are often used to model these systems are: *labeled transition systems*, *petri nets*, *finite state machines*, *boolean formula*, etc.

The goal of our work is to provide a theory, methods and tools for achieving the correctness of systems. We focus on the formal verification since, as mentioned above, it allows to achieve the correctness of systems. We provide below a brief description of the current state of the art in formal verification.

State of the Art in Formal Verification

There are basically two main directions in formal verification - *theorem proving* and *model-checking*.

Theorem proving [BM79, BM88, GH93, GM93, ORR+96] uses mathematical proofs to show that a system satisfies its requirements. In this approach, both system and requirement are expressed in form of formulas in some mathematical logic. This logic defines a set of axioms and inference rules that theorem proving uses, together with intermediate lemmas, to find a proof of a desired property. An advantage of theorem proving is the capacity of dealing directly with infinite systems by using techniques like structure induction. However, one can not get counterexample when the proof fails. Although many theorem prover tools have been developed to support building and checking the proofs, the finding of these proofs is not always feasible and may require a lot of expert intervention from users, that makes the theorem proving process slow and often error-prone. The lack of automation prevents this approach from being largely used in the industrial context.

Model-checking [QS82, CE81, CGP99] is a fully automatic verification technique which relies on building a finite model of the system and checking whether that model meets its specification. Usually the system is modeled as a finite state machine and the specification is expressed as a temporal logic formula. The temporal logic allows describing the change over time and is therefore suitable for most of the necessary correctness properties such as *safety* properties (*always*, i.e. something bad never happens), *liveness* properties (*eventually*, i.e. something good happens), etc.. The check is performed by using an efficient search procedure on the exhaustive state space graph of the system's model. Model checking has been extensively developed and used for verifying both software and hardware systems. The automatic nature of model-checking makes it attractive for practical use in the industry. It can be also used for checking partial specification when the system has not been completely specified. Moreover, if a property does not hold, model checking provides a counterexample, a path through the model that reveals this violation.

Despite having been successfully applied in the industrial community, there are several problems which make model-checking difficult for verifying large systems:

- Model checking needs to explore the entire state space of the models of systems, therefore it is not suitable for infinite-state systems. However, a big number of important systems are infinite, and the exploration of the state space of their infinite-state models is not possible or requires extra pre-processing (such as abstraction) to make it feasible.

- Even with finite-state systems, model checking is not always scalable. The systems nowadays become more and more complex with a large number of parallel processes. Model checking techniques examine all possible paths through the system's model to determine whether or not the property being verified is violated, and that is the source of difficulty because the number of possibilities in the global model is exponential in the number of component processes. This is called *state space explosion* problem in Model-Checking.

- Model checking is not a priori guarantee of correctness. It helps developer detect, understand and then fix inconsistencies, ambiguities, bugs, etc., in the model of the system. However if errors or bugs are found after the complete construction of the system's model, we might have to reconstruct or modify the global model which may cause significant loss of time, money or human resource.

A lot of work has been done to overcome the problems in model-checking, specially the state space explosion problem. The major goal is to make the formal verification scalable in order to increase the size of the systems that can be handled. They can in general be categorized into two approaches: optimization/improvement of model-checking algorithms and compositional reasoning.

Model-Checking Improvement

Symbolic Model Checking

A well-known improvement of model-checking algorithms is *symbolic model-checking* [McM93, BCM+90]. This method represents implicitly the state graph as a formula in propositional logic instead of building it explicitly. It uses boolean encoding for representing state machine and set of states, therefore allows manipulating a set of states rather than a single state in the explicit enumeration of states. All operations are handled as boolean functions by using the Binary Decision Diagrams (BDDs).

A symbolic representation based on BDDs provides a canonical form for boolean formula that is more compact than conjunctive and disjunctive normal form. The use of BDDs allows to verify extremely large systems having up to 10^{120} states. It is also successful in verifying several systems of industrial complexity. The property to be verified is evaluated recursively by iterative fixed-point computations on the reachable state space. More precisely, the rough procedure for checking a safety property is : initially, the set of initial states is represented as a BDD from which an iterative process starts. At each step i, the iteration adds to the BDDs the set of states that can be reached in i steps from the initial states and intersects the new states with the set of states violating the property. If the intersection is not empty, it means that an violation has been detected. The process terminates when there is no more new reachable states or an error is found. If the process terminates without any error, the property holds; otherwise, a counterexample is provided.

In spite of such success, symbolic model checking still has its limitations due to the size of BDDs. In some cases the BDD representation can be exponential in the size of system description. Moreover, BDDs are very sensitive to ordering of variables. Finding a good ordering, which yields the smallest BDD for a given formula is an NP-complete problem.

Bounded Model Checking

An alternative method that can avoid the state space explosion is *Bounded Model Checking* (BMC) [BCCZ99, CBRZ01] proposed by Biere et al. in 1999. The basic idea of the method is to search for a counterexample within a bounded number steps of executions. Given a bounded execution length k, BMC constructs a propositional formula that represents the set of initial states and the states that can be reached from initial states within k steps. It also constructs a formula expressing the violation of a property P in one of these k steps. Then the conjunction of two formulas is checked. If the conjunction is satisfiable, BMC provides a counterexample of length at most k. If it is not satisfiable, we can either increase k until a violation is found or stop if time or memory constraints are exceeded.

The BMC problem is reduced to a propositional satisfiability problem that determines whether a propositional formula in conjunctive normal form has a truth assignment that makes the formula true. This problem can be efficiently solved by powerful SAT tools. BMC and BDD-based symbolic model-checking are incomparable. There are several case studies that can not be verified by symbolic model checking but can be verified by BMC and vice-versa. A disadvantage of BMC is the incompleteness of the method. The absence of error after k execution steps does not prove the correctness. A possible solution is the

finding of the longest shortest path between any two states (the maximum value k_{max} of k). The states obtained in k_{max} steps allows covering all the reachable states since the the path from the initial states to any reachable state is always shorter than or equal to k_{max}. However finding such k_{max} is extremely hard.

Partial order reduction

Partial order reduction [God91, GW92, Pel94, CGP99] is a verification method that simplifies the size of state space to be searched by a model-checking algorithm. It exploits the commutativity and concurrently executed transitions, which result in the same state when executed in different orders. Intuitively, if two transitions t_1 and t_2 are executed consequently but in any order, the system arrives in the same state, so it is not necessary to consider both the t_1t_2 and t_2t_1 interleavings.

Compositional Reasoning

The second approach is *compositional reasoning* [CLM89, CMP94, Lon93] that verifies each component of the system in isolation and allows global properties to be inferred about the entire system. The basic idea is the use of divide-and-conquer approach: the system is decomposed into subsystems and these subsystems are analyzed individually. Since subsystems are smaller than the whole system, the individual analysis of the subsystems reduce the effects of the state space explosion problem. The guarantee of global property is then determined by composing the results of these individual analysis. Since through this thesis, we propose a compositional method for the verification of component-based systems, we will focus on compositional approach by presenting below several existing compositional methods.

Abstraction

Abstraction [CC77, Lon93, CGL94, DF95, LGS+95] is a popular technique which verifies properties on a system by firstly simplifying it. The simplification is often based on the conservative aggregation of states. The simplified system, which is called *abstract system*, is usually smaller than the original system (*concrete system*), so the state space is reduced. For a system obtained from the parallel composition of a set of components, i.e $S = B_1 \parallel \cdots \parallel B_n$, the compositional abstraction first computes, for each component B_a, an abstract component B_i^a, then it composes the abstract components $S^a = B_1^a \parallel \cdots \parallel B_n^a$ to obtain an abstract system S^a of S. The abstraction is required to be sound, i.e. the properties that hold on the abstract system also hold on the concrete system. However, the abstraction is often not complete, i.e. not all true properties of the concrete system are also true on the abstract system so that a process of abstraction refinement may be necessary.

Assume-guarantee

Assume-guarantee [MC81, Jon83, Pnu85, HQR98, dRdBH+00, GPB02, CGP03] is a semi-automatic compositional approach that decomposes properties into two parts. One is an

assumption about the global behavior of the environment and the other is a property guaranteed by the component when the assumption about its environment holds. The assumption is needed since when a subsystem is verified it may be necessary to assume that the environment behaves in a certain manner. Consider a system S which is decomposed into two subsystems S_1 and S_2. P is a property to be verified on the parallel composition of S_1 and S_2, denoted by $S_1 \parallel S_2$. The basic assume-guarantee rule is as follows:

$$\frac{\begin{array}{c}\langle A\rangle S_1 \langle P\rangle \\ \langle true\rangle S_2 \langle A\rangle\end{array}}{\langle true\rangle S_1 \parallel S_2 \langle P\rangle}$$

That is, if under assumption A, subsystem S_1 satisfies property P and A is satisfied by subsystem S_2, then the system resulting from the parallel composition $S_1 \parallel S_2$ satisfies the property P. Even though it is widely touted, many issues make the application of assume-guarantee rules difficult. They are discussed in detail in a recent paper [CAC08]. The paper provides an evaluation of automated assume-guarantee techniques. In many cases, the verification based on assume-guarantee is not better than monolithic verification in time and memory usage. The main difficulty is finding decompositions into sub-systems in the case of many parallel sub-systems $S_1 \parallel \cdots \parallel S_n$. The verification performance depends on the way of decomposition but finding a good decomposition is not always feasible. Another problem is choosing adequate assumptions for a particular decomposition. The assumption should be weak enough to be satisfied by a sub-system but strong enough to prove the global property.

Interface processes

[CLM89] proposes a method for reducing the complexity of temporal logic model checking in systems composed of many parallel processes. It minimizes the global state transition graph by focusing on the communication among the component processes. The method models the environment of a process by another process called an interface process. In interface process, only variables involved in the interface between two components are considered and events that do not relate to the communication variables are eliminated. Therefore, the interface process is often smaller than the original process but it preserves properties that refer to interface variables. This method is specially suitable for loosely coupled systems through a small number of interacting variables. However, the method is not very efficient with the tightly coupled systems because the interface process may not be smaller than the original process. And the method has difficulty in handling more general properties involving temporal assertions about several processes.

Partitioned Transition Relations

In model checking, the set of reachable states from initial states (or co-reachable from bad states) is obtained by computing the set of successors (or predecessors). This process requires the construction of the transition relation of the global systems and that is a source of difficulties. [BCL91] provide methods for computing that set by using the transition

relations of each component separately during traversal of the state graph. The set of states in the global graph is then obtained by combining the individual results. More precisely, model-checking requires the computation of the image or pre-image of a set of states under a transition relation. For example, if $S(V)$ is a set of states depending on a set of variables V, and $N(V, V')$ is a transition relation relating the current state variables V and next state variables V', then the image of S is given by $\exists V[S(V) \wedge N(V, V')]$. The computation of the value of a large formula with many quantifiers is quite expensive. The method called *partitioned transition relations* in [BCL91] deals with this problem by decomposing the global formula into sub-formulas and the quantifier elimination is performed on these smaller sub-formulas and therefore reduces the cost of the operation.

Lazy Parallel Composition

In contrast to partitioned transition relations method, *lazy parallel composition* presented in [TSL+90] restricts the transition relation of each component before generating the global restricted transition relation. The method is based on the agreement between the restricted transition relation and the global transition relation for "important" states, while other states may behave in a different way. If the original global transition relation N and a set of state S, the computation of the set of successors of S can use any restricted transition relation such that $N'|_S = N|_S$, i.e. N and N' agree on transitions that start from states in S. The advantage of the method is that the restricted transition relation is often smaller than the global transition relation.

Lazy Compositional Verification

A compositional method, *lazy compositional verification*, is presented in [Sha98]. The method allows to prove a global property by showing that it is satisfied by composing a component with an abstract environment and this environment eventually holds of the other components in the system. More concretely, a property C of a component P is satisfied by the composition $P \parallel E$ where E is an abstract environment specification E that captures the expected behavior of the environment. Then when P is composed with another component Q, C might not be property of $P \parallel Q$ but C is property of $P \parallel (Q \wedge E)$. If $P \parallel (Q \wedge E)$ can be simplified to $P \parallel Q$, then E is redundant and can be eliminated. However, in contract to assume-guarantee approach, it is not necessary that Q implies E. While E has eventually to be shown to hold of other components in the system, this proof obligation can be discharged lazily as the system design is being refined. The advantage of lazy compositional verification is that the proof that one component meets the expectations of other components can be delayed until sufficient detail has been added to the design.

Deductive Verification

Deductive methods for proving safety properties, which are also considered as invariance properties, of transition systems are based on a proof rule which can be formulated as follows. To prove that some given predicate Φ is an invariant of a given program S, i.e.

every reachable state of S satisfies Φ, it is necessary and sufficient to find an auxiliary predicate Φ^{aux} with the following properties:

- Φ^{aux} is stronger than Φ,
- Φ^{aux} is preserved by every transition of S, i.e., for every states s and s', if s satisfies Φ^{aux} and s' is reachable from s by a transition, then s' also satisfies Φ^{aux},
- Φ^{aux} is satisfied by every initial state of S.

As shown e.g., in [MP95], this rule is sound and (relatively) complete for proving invariance properties of transition systems. It is very important to understand that the completeness result/proof of this rule does not give a clue of how to find the auxiliary predicate. Indeed, choosing the set of reachable states *Reach(S)* as auxiliary predicate reduces the original problem to the checking of the first premise. Moreover, if S is a finite-state system the predicates can be expressed in propositional logic and checking the premises can be done algorithmically. However, in general, one needs an assertion language which is at least as expressive as integer arithmetic to express predicates, which makes checking the premises of the rule undecidable. Even worse, there are systems for which *Reach(S)* is expressible using closed formula over integers yet computing such a representation cannot be done effectively. The deductive rule provides only a partial answer to the verification of invariance properties. It leaves open (i) how to find the auxiliary predicate Φ^{aux} and (ii) how to prove that Φ^{aux} is preserved by every transition of S and satisfied by the initial states. Problem (ii) is related to the problem of proving tautologies of the underlying assertion language.

Problems in Formal Verification

On exploring the current state of the art in formal verification, it becomes clear that a formal verification method needs to address the following problems:

- **Scalability** that avoids the state space explosion problem and therefore allows increasing the size of systems to be verified.
- **Effectiveness** that permits to detect as early as possible errors of systems' models in the design phase.
- **Incrementality** that integrates verification into the design process.
- **Compositionality** that allows inferring global properties of a system from the known local properties of its sub-systems.

We have mentioned the problems, the needs in software and hardware engineering to construct correct systems. We have also presented the existing approaches addressing to these needs and its general limitations. We have given a brief presentation on some related work that has been done in the formal verification together with its advantages and disadvantages. In the next section, we will present our methods for the compositional and incremental verification.

Our Approach

We present, in this thesis, a compositional approach for verifying safety properties of component-based systems. We exploit both local and global aspects of systems. The former is related to the local behaviors of components. The latter represents global constraints between components which are strongly synchronized. These constraints are important, specially for checking deadlock-freedom since the global deadlocks are due to the strong synchronization. Furthermore, the synchronizations restrict the global behavior and therefore these global constraints allow to eliminate product states which are not feasible by the semantics of parallel composition.

Our rule can be seen as an instance of the proof rule of the deductive approach. We describe techniques for generating automatically auxiliary predicates by using two kinds of invariants of systems: *component invariants* and *interaction invariants*. Component invariants express constraints on local state space of components. Interaction invariants characterize restrictions enforced by synchronizations between components on the global state space of the systems. They are computed automatically from the set of component invariants and the set of interactions between components.

The general rule of our approach is shown in Equation (1).

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma\{B_i\}_i < \Phi >} \quad (1)$$

That is, if $\{\Phi_i\}_i$ are respectively component invariants of a set of atomic components $\{B_i\}_i$, Ψ are interaction invariants computed from a set of interactions γ between $\{B_i\}_i$ and $\{\Phi_i\}_i$, and if the conjunction of these invariants $\bigwedge_i \Phi_i \wedge \Psi$ implies a predicate Φ, then Φ is an invariant of the system resulting from the parallel composition of the set $\{B_i\}_i$ by the set of interactions γ.

The rule allows to prove invariance of a predicate Φ for a system obtained by using a n-ary composition operation parameterized by a set of interactions γ. It uses global invariants which are the conjunction of individual invariants of components Φ_i and an interaction invariant Ψ. Interaction invariants are computed symbolically by solving a set of boolean equations called Boolean Behavioral Constraints obtained from the set of interactions. In the case of system with data, interaction invariants are computed from abstractions of the system to be verified. These are the composition of finite state abstractions B_i^α of the components B_i with respect to their invariants Φ_i. Finally, the invariance of Φ is verified by checking tautology $(\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi$ which can simply be done by using a SAT-Solver tool to check the unsatisfiability of $(\bigwedge_i \Phi_i) \wedge \Psi \wedge (\neg\Phi)$.

Figure 1 illustrates the idea of our method for a system composing of two components strongly synchronized. The basic idea is to approximate as precisely as possible the set of reachable states of the system. In the figure, black area is the set of reachable states of the system that we want to approximate, Φ_1 and Φ_2 are components invariants. Starting from component invariants, the intersection of Φ_1 and Φ_2 is already an over-approximation of the set of reachable states. However this intersection characterizes the maximal combination of the state spaces of two components. It does not take into account the restrictions by strong synchronizations on two components and therefore is a very weak approximation.

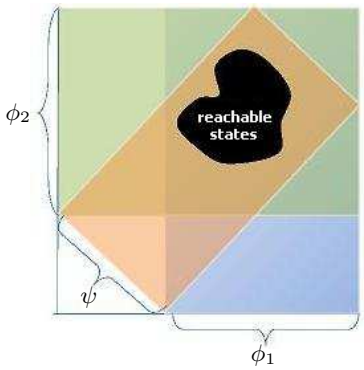


Figure 1: Idea of the method

By adding some interaction invariant Ψ, we can reduce this intersection and get a more precise over-approximation of the set of reachable states.

Our method differs from assume-guarantee methods in that it avoids combinatorial explosion of the decomposition and is directly applicable to systems with n-ary interactions. Furthermore, it only needs guarantees for components. It replaces the search for adequate assumptions for each component by the use of interaction invariants. These can be computed automatically from given component invariants (guarantees). Interaction invariants correspond to a "*cooperation test*" in the terminology of [AFdR80] as they allow to eliminate product states which are not feasible by the semantics of parallel composition.

We also study and propose methods for incremental construction and verification. Incremental construction aims to deal with the complexity of heterogeneous and large-scale systems. It allows building a composite component from smaller parts. Incremental construction provides flexibility in building systems. During the incremental construction process, the verification is necessary to detect early errors in the constructed system. The verification should take advantages of the incremental construction by integrating verification into construction phase. The idea is that, at each stage of the construction, some properties are established and ideally, they should be preserved in the new system obtained in the next steps of the construction. If they are not preserved, at least they can be used in establishing properties for the new system.

We first provide a systematic methodology for incremental construction of components-based systems. We propose rules on invariant preservation from which the already established invariants would not be violated during the incremental construction. However, it is not always possible to apply these rules because many systems do not satisfy them. Moreover, the preserved invariants are often not strong enough to prove safety properties because they do not take into account the new constraints enforced to the system in the incremental construction process. Therefore, we propose methods for incremental computation of invariants from the established invariants.

The idea of the incremental computation of invariants is presented in Figure 2. At some stage of the incremental construction, we have obtained sub-systems with established invariants I_1 and I_2. These sub-systems are then composed to build a new system

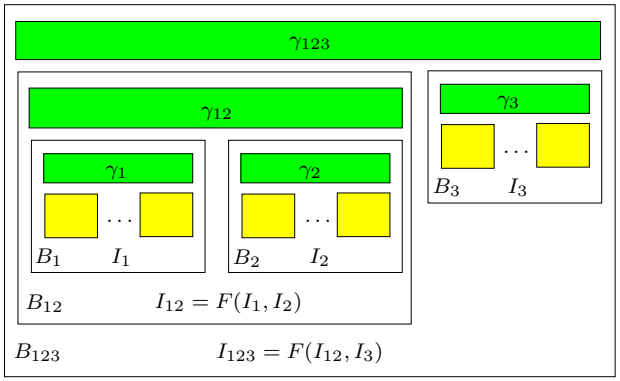


Figure 2: Incremental Verification Idea

and the invariants of the new system are computed from I_1 and I_2 by some function F: $I_{12} = F(I_1, I_2)$. Similarly, the new system is then composed with another sub-system with established invariants I_3. The invariants of the global system are computed in the same way from I_{12} and I_3: $I_{123} = F(I_{12}, I_3)$. The incremental verification by computing incrementally invariants reduces significantly the verification cost in both time and memory usage.

The main contributions of the thesis are:

- We propose a heuristic method for proving safety properties [BBNS08, BBNS10]. For a given property, the method consists in iteratively conjoining the predicate characterizing violations of the property with an over-approximation of the reachable states of the system. The over-approximation is the conjunction of two kinds of invariants: component invariants and interaction invariants. If the conjunction is false, then the property is guaranteed. Otherwise, to eliminate infeasible counterexample, new invariants are computed until either the conjunction becomes false or the method fails to prove invariance of the property. In this case, additional reachability techniques can be used for finer analysis.

- We provide heuristics for computing two types of invariants. Component invariants are over-approximation of reachable states of the components and are generated by simple forward analysis of their behaviors. Interaction invariants characterize global constraints on the global state space induced by strong synchronizations between components. We propose several methods for computing interaction invariants from boolean constraints obtained from interactions.

- We also provide incremental construction and verification methods which are based on a construction process leading to a composite component through a sequence of constituent components [BBL+09, BLN+10]. The sequence starts from a set of atomic components and applies incrementally synchronization constraints. Incremental verification relates the verification process to system construction. It takes advantage of

the system structure for coping with complexity of the global verification. We study rules which allow preserving established invariants during the incremental construction. For the general case where a system may not satisfy these rules, we propose methods for computing incrementally invariants of the entire system from the established invariants of its constituents.

- We propose a method for dealing with data transfer on interactions between components. The method is based on a projection of the changes on interactions into components and on a replacement of data transfer by a type of component called *interaction component*. These projection and replacement enable the construction of an equivalent abstract system without data transfer on which we can apply our verification method.

- The compositional and incremental methods have been fully implemented in the D-Finder tool-set [BBNS09]. We have successfully applied these methods to prove deadlock-freedom of non-trivial case studies, some of them are case studies of our projects, described in the BIP language [BGL+08, BGI+09]. Interesting and significant results show the efficiency of the method as well as the capacities of D-Finder.

Organization of the Thesis

The thesis is split into five parts: the first (Chapter 1) presents an overview of the BIP framework; the second (Chapter 2) describes our compositional verification approach and methods for computing two types of invariants; the third (Chapter 3) presents incremental construction and verification methods; the fourth (Chapter 4) presents a method for dealing with systems with data transfer; the fifth (Chapter 5) describes the D-Finder tool together with the implementation; the sixth (Chapter 6) presents the experimental results on some case studies; and the last part draws the conclusions and future work. The details of all chapters are as follows:

- Chapter 1 presents the basic ideas about component-based methodology, the basic notions about components, their composition using glues, and the necessary properties for component-based construction of systems. It introduces the BIP component framework, describing its architecture, its semantics as well as its properties.

- In Chapter 2 we present our method for verifying safety properties of component-based systems by using invariants. We also present methods for computing two types of invariants: component invariants and interaction invariants. Component invariants are over-approximations of the set of the reachable states generated by forward propagation techniques. Interaction invariants are derived automatically from component invariants and their interactions. When proving invariance of a property fails, it is possible to find stronger invariants by computing stronger component invariants from which stronger interaction invariants are obtained. An application of the method to deadlock detection is also presented in this chapter.

- Incremental construction and verification methods are presented in Chapter 3. First, we formalize an incremental component-based construction process. A composite component is obtained as the composition of a set of atomic components. Then, we formalize the process of incremental construction based on the operation of increments. The construction is hierarchical: increments can be applied either at the same level or at different levels. We associate with the incremental construction the incremental verification. We study rules which allow preserving already established properties. For the general case where a system may not satisfy these rules, we propose methods for incremental computation of global invariants of a composite component from the invariants of its constituent components.

- In Chapter 4, we propose a method for verifying safety properties of component-based systems with data transfer. The method is based on the transformation from a system with data transfer into an equivalent system without data transfer. The transformation is done by projecting the changes of data on interactions into components and by replacing the data transfer by a component called *interaction invariant*.

- In Chapter 5, we present the D-Finder tool and the implementation of the modules in D-Finder: the modules for generating component invariants, for making abstractions, for generating interaction invariants, for checking deadlock-freedom, etc. In the interaction invariant generation module, we implement several methods for computing interaction invariants: an enumerative method using the SMT Sat-Solver Yices and the CUDD package; two symbolic methods based on two symbolic operations: Positive Mapping and Fixed-Point computation. Moreover, two incremental methods are also implemented in D-Finder to support incremental verification.

- In Chapter 6, we provide non trivial case studies showing the capacities of D-Finder as well as the efficiency of the method. First we present results on two systems without data, the Dining Philosopher, a classical problem in detecting deadlocks, and Gas Station where we increase as much as possible the size of the system to show the scalability of the method. Then we present results on an Automatic Teller Machine (ATM) case study which has quite complex structure with a number of variables. And finally, we consider a module in a robotic system which is a case study in one of our projects.

- We conclude the thesis in Chapter 7, with an overview of the work and its future perspectives.

CHAPTER 1

BIP Modeling Framework

Contents

The design technique and formal verification have mutual influence. A clear and simple design of systems can reduce the cost of verification. The major goal of formal verification is to check the correctness of the system and to provide diagnostics when a requirement is not met. These diagnostics on a clear model can help designers understand easily the problems and locate their source. The modification/correction of the model may be expensive if it requires significant changes of the model, specially with the growing size and complexity of systems. The design techniques are therefore important to reduce this cost by dealing with the complexity of systems.

1.1 Component-based Design

Component-based design techniques are used to cope with the complexity of the systems. The idea is that complex systems can be obtained by assembling components (building blocks). This is essential for the development of large-scale, evolvable systems in a timely and affordable manner. It offers flexibility in the construction phase of systems by supporting the addition, removal, or modification of components without any or very little impact on other components. Components are usually characterized by abstractions that ignore implementation details and describe properties relevant to their composition, e.g., transfer functions, interfaces. Composition is used to build complex components from simpler ones. It can be formalized as an operation that takes, as input, a set of components and their integration constraints and provides, as output, the description of a new, more complex component. This approach mitigates the complexity of systems by offering incrementality in the construction phase. However, for being able to deal with complexity in verification, the component frameworks need to allow constructivity along the design process.

Constructivity is the possibility to build complex systems that meet given requirements by assembling components with known properties. The correctness of the systems is inferred and guaranteed by construction with little computation. Component-based design techniques confer numerous advantages, in particular through reuse of existing components. A key issue is the existence of composition frameworks ensuring the correctness of composite components. In particular, we need frameworks allowing not only reuse of components but also reuse of their properties for establishing global properties of composite components from properties of their constituent components. Hence, we need theory allowing constructivity and meeting the following requirements:

Incrementality. This means that composite systems can be considered as the composition of smaller parts. Incrementality provides flexibility in building systems by simply adding or removing components and the result of construction is independent of the order of integration. It is necessary for progressive analysis and the application of compositionality rules. Incrementality allows coping with the complexity of the heterogenous and large-scale systems in both construction and verification phases.

Compositionality. Compositionality rules allow inferring global system properties from the local properties of the sub-systems. (e.g, inferring global deadlock-freedom from the deadlock-freedom of the individual components). Compositionality is necessary for obtaining correctness-by-construction.

Composability. Composability rules guarantee that, under some conditions, essential properties of a component will be preserved after integration. Composability means stability of previously established component properties across integration, e. g, a deadlock-free component will remain deadlock-free after gluing together with other components. Composability is essential for incremental construction as it enables the construction of large systems without disturbing the behavior of their components.

Verimag has developed a Component-based Modeling Framework called BIP (**B**ehavior - **I**nteraction - **P**riority) [BBS06] for modeling heterogeneous real-time components. We will provide in this chapter a short description of the BIP framework. We start by giving notions about component-based framework: the ideas about component, their composition and the necessary properties for component-based construction of systems.

1.2 Basic Ideas

Component-based design consists in building a component satisfying a given property from:

- a set of components $B_1, B_2, \ldots B_n$, described by their behavior, and
- a set of glue operators $\mathcal{GL} = \{gl_1, gl_2, \ldots gl_n\}$ on components characterizing coordination between components.

A component is an entity with well-defined interfaces for interacting with its environment. It denotes an executable description of which the run can be modeled as sequences of actions. Tasks, processes, threads, functions, blocks of code can be considered as components. Two types of components are considered:

- *Atomic* component, a basic element that only represents behavior.
- *Compound* component, a composition of a set of components by using *glue*.

A component is denoted graphically by a box. The box of an atomic component contains behavior inside and the box of a composite component contains other components and glue. Behavior is represented by a labeled transition system (LTS).

Definition 1 (Labeled Transition System) *A labeled transition system is a triple* $B = (L, P, \mathcal{T})$*, where* L *is a set of locations,* P *is a set of actions, and* $\mathcal{T} \subseteq L \times P \times L$ *is a set of transitions, each labeled by an action.*

For any pair of locations $l, l' \in L$ and an action $p \in P$, we write $l \xrightarrow{p} l'$, iff $(l, p, l') \in \mathcal{T}$. If such l' does not exist, we write $l \not\xrightarrow{p}$.

When components are composed together, it might be necessary to restrict the product of behaviors in order to meet some global properties. Glue are used for this purpose. The glue is a separate layer that composes the underlying layer of behaviors. It is a set of operators mapping tuples of behaviors into an equivalent behavior. Given $\{B_1, B_2, \ldots, B_n\}$ a set of atomic components, their composition with the glue $\mathcal{GL}$ is a composite component B (figure 1.1) represented as follows:

$$B = \mathcal{GL}(B_1, B_2, \ldots B_n)$$

The new behavior B is obtained by applying restrictions implied by the meaning of the glue to the product of the behaviors of $B_1, B_2, \ldots B_n$. Since glue restrict the product of the behavior, the behavior of B is smaller than the product of $B_1, B_2, \ldots B_n$. This new component B can be further used for composition with other components.

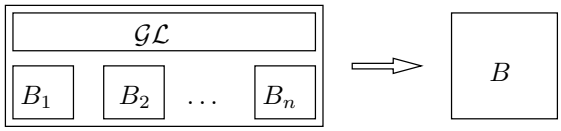


Figure 1.1: Component Composition.

1.3 BIP Modeling Framework

BIP is a component framework where the composition of behaviors are performed by two kinds of glue: *interactions* and *priorities*. Interactions characterize collaborations between components and priorities allow choosing an interaction to be executed amongst possible interactions. The construction of BIP components is based on a 3-layers architecture (figure 1.2): **B**ehavior, **I**nteraction and **P**riority. Compound components are built by composition of simpler components and its layers are obtained by composing separately the layers of the constituents. A component is composed of:

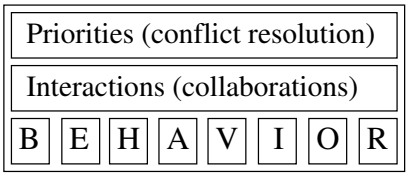


Figure 1.2: Layered component model

- **Behavior**: behavior is a labeled transition system describing elementary transformations of states. Transitions consist of triggers and local computations. Triggers are conditions depending on local state and port expressions which characterize ability of the component to interact with its environment.

- **Interactions**: interactions are architecture constraints on behavior. An interaction is a global transformation of the states of different components. Interactions can be considered as a function allowing computing the interactions of a component from those of its constituents.

- **Priorities**: priorities provide a mechanism for restricting the global behavior of the layers underneath by filtering amongst possible interactions. They are very useful for enforcing state invariant properties such as mutual exclusion and scheduling policies.

A component in BIP can also be viewed as a point in a three-dimensional space represented in Figure 1.3. The dimension *Behavior* characterizes component behavior and the space $Interactions \times Priorities$ characterizes the overall structure of the system.

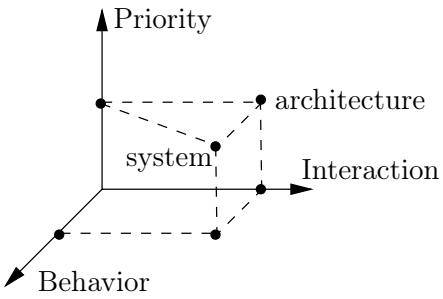


Figure 1.3: Three-dimentional space construction

1.3.1 Atomic Components

A BIP atomic component is a Labeled Transition System extended with data. It consists of a set of ports used for the synchronization with other components, a set of control locations, a set of transitions and a set of local variables. The transitions describe the behavior of the component.

Definition 2 (Atomic Component) *An atomic component is a transition system extended with data* $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ *, where:*

- $(L, P, \mathcal{T})$ *is a transition system, that is*
  - $L = \{l_1, l_2, \ldots, l_k\}$ *is a set of control locations,*
  - $P = \{p_1 \ldots p_n\}$ *is a set of ports,*
  - $\mathcal{T} \in L \times P \times L$ *is a set of transitions,*
- $X = \{x_1, \ldots, x_n\}$ *is a set of variables and for each* $\tau \in \mathcal{T}$ *respectively,* g_τ *is a guard, a predicate on* X*, and* $f_\tau(X, X')$ *is an update relation, a predicate on* X *(current) and* X' *(next) state variables.*

A transition is of the form $\tau = (l, p, g_\tau, f_\tau, l')$ where l (respectively l') is the source (respectively destination) location, p is a port through which an interaction is sought. The transition τ can be executed only if its guard g_τ, a boolean condition on the set of variables X, is true. g_τ is also known as the pre-condition for interaction through the port p. f_τ is a computation step consisting of local state transformations. A transition can be represented in a simple form $\tau = (l, p, l')$ to insist only the state transformation or in case that the guard $g_\tau = true$ and there is no internal computation f_τ.

In BIP there are two kinds of ports:

- *Complete port*: an active port which can initiate an interaction without synchronization with other ports. Complete port is graphically represented by a triangle.
- *Incomplete port*: a passive port hence it needs a synchronization with other ports to execute its transitions. An incomplete port is denoted by a circle.

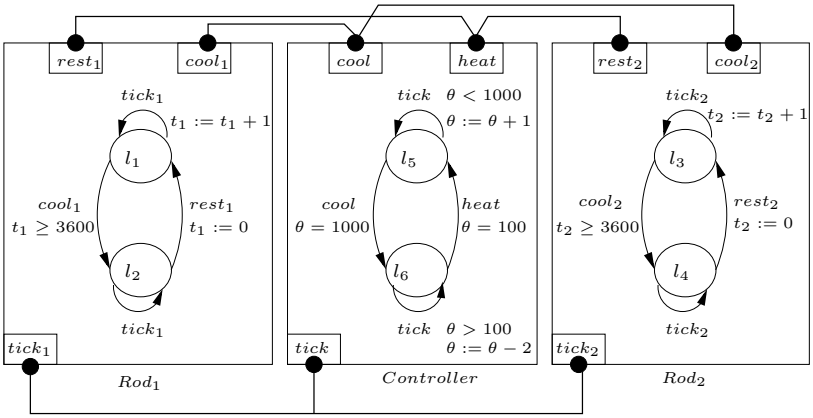


Figure 1.4: Temperature Control System

A port is *enabled* if at least one of its transitions is enabled, or *disabled* if all its transitions are disabled. A transition $\tau = (l, p, g_\tau, f_\tau, l')$ is enabled if its source location l is reached and its guard g_τ is true. Conversely, the transition τ is disabled if the component is not at l or its guard g_τ is false.

The behavior of an atomic component is a labeled transition system with moves of the form $(l_1, x) \xrightarrow{p} (l_2, x')$, where l_1, l_2 are control locations of the automaton and x, x' are respectively valuations of the variables at each control location. The move $(l_1, x) \xrightarrow{p} (l_2, x')$ is possible if there exists a transition $(l_1, p, g_\tau, f_\tau, l_2)$, such that $g_\tau(x) = true$. As a result of the move, the set of variables are modified to $x' = f_\tau(x)$. The semantics of execution of transitions is formally defined as follows:

Definition 3 (Semantics) *The semantics of* $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$*, is a transition system* $(Q, P, \mathcal{T}_0)$ *such that*

- $Q = L \times \mathbf{X}$ *is a set of states, where* $\mathbf{X}$ *denotes the set of valuations of variables* X*,*
- $\mathcal{T}_0$ *is the set including transitions* $((l, \mathbf{x}), p, (l', \mathbf{x}'))$ *such that* $g_\tau(\mathbf{x}) \wedge f_\tau(\mathbf{x}, \mathbf{x}')$ *for some* $\tau = (l, p, l') \in \mathcal{T}$*. As usual, if* $((l, \mathbf{x}), p, (l', \mathbf{x}')) \in \mathcal{T}_0$ *we write* $(l, \mathbf{x}) \xrightarrow{p} (l', \mathbf{x}')$*.*

We define here useful notions for later use: given a transition $\tau = (l, p, l') \in \mathcal{T}$, l and l' are respectively, the *source* and the *target* location denoted respectively by ${}^\bullet\tau$ and $\tau^\bullet$. We extend this notation for ports: ${}^\bullet p = \{{}^\bullet\tau | \tau = (l, p, l')\}$ and $p^\bullet = \{\tau^\bullet | \tau = (l, p, l')\}$ are respectively the set of source and target locations of the transitions labeled by the port p.

We present below the Temperature Control System, a case study in BIP, which is used as an example for illustrating the modeling of components and of system in BIP. This case study will also be used for showing the application of our compositional verification method in the next chapter.

Example 1 (Temperature Control System) *[ACH+95] This system controls the coolant*

temperature in a reactor tank by moving two independent control rods. The goal is to maintain the coolant between the temperatures θ_m and θ_M. When the temperature reaches its maximum value θ_M, the tank must be refrigerated with one of the rods. The temperature rises at a rate v_r and decreases at rate v_d. A rod can be moved again only if T time units have elapsed since the end of its previous movement. If the temperature of the coolant cannot decrease because there is no available rod, a complete shutdown is required.

We provide a discretized model of the Temperature Control System in BIP, decomposed into three atomic components: a Controller and two components Rod1, Rod2 modeling the rods. We take $\theta_m = 100°$, $\theta_M = 1000°$, $T = 3600$ seconds. Furthermore, we assume that $v_r = 1°/s$ and $v_d = 2°/s$. Figure 1.4 presents the BIP model of Temperature Control System. The components Rod1 and Rod2 are identical, up to the renaming of locations and ports. Each one has two control locations and four transitions: two loop transitions labeled by tick and two transitions synchronized with transitions of the Controller. The Controller will be described in detail here as an example for atomic component.

Controller component is composed of a set of locations $L = \{l_5, l_6\}$, a set of ports $P = \{heat, cool, tick\}$, a variable θ representing the temperature of the system and a set of the transitions $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ where:

- $\tau_1 = (l_5, tick, g = (\theta < 1000), f = (\theta := \theta+1), l_5)$*: from l_5 location, tick transition can take place if $\theta < 1000$ and it increases the temperature θ by 1. Since this transition is a loop, the component is still at l_5 location after the transition.*
- $\tau_2 = (l_5, cool, g = (\theta = 1000), l_6)$*: from l_1 location, if the temperature $\theta = 1000$, cool transition must take place to refrigerate the system by triggering one of two Rods.*
- $\tau_3 = (l_6, tick, l_6)$*: at l_6 location, tick loop transition can be executed if $\theta > 100$ and it decreases the temperature θ by 2.*
- $\tau_4 = (l_6, rest, l_5)$*: from l_6 location, heat transition must be executed if the temperature θ reaches 100 ($\theta = 100$). The component returns to location l_5.*

The textual for atomic components in BIP is the following:

```
atomic component::=
  component component_id
    {port complete/incomplete port_id+}+
    [data type_id data_id+]+
    behavior
      initial do statement to state_id
      {state state_id
        {on port_id [provided guard]
        [do statement] to state_id}+}+
    end
  end
```

That is, an atomic component consists of a declaration followed by the definition of its behavior. Declaration consists of ports and data. Ports are identifiers and are specified as

complete or incomplete. For data, basic C types (*int*, *double*, *char*, etc.) can be used. In the behavior, *guard* and *statement* denote respectively C expressions and statements.

The above description of a component defines a type of atomic component from which a set of component instances can be created and used as atomic components to build a system. The creation of instances will be described in the composite component section.

Behavior is defined by a set of transitions. The initial location of the component together with the initial action specifying initial valuations of variables is described following the keyword **initial**. The keyword **state** is followed by a control location and the list of outgoing transitions from this location. Each transition is labeled by a port identifier followed by its guard, its function and a target location.

Example 2 *The BIP text of the Controller component in Example 1 is as follows:*

```
component Controller
  port incomplete tick, cool, heat
  data int th  /* temperature variable theta */
  behavior
    initial do th = 100 to l5
    state l5
      on tick provided th < 1000 do th = th + 1 to l5
      on cool provided th == 1000 to l6
    state l6
      on tick provided th > 100 do th = th - 2 to l6
      on heat provided th == 100 to l5
  end
end
```

1.3.2 Interactions

In BIP, components communicate through a set of interactions. An interaction is a non-empty subset of ports of different components and it synchronizes the executions of the ports.

Definition 4 (Interactions) *Given a set of components* $(B_1, B_2, \ldots, B_n)$*, where* $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i})$*, an interaction* a *is a set of ports, subset of* $\bigcup_{i=1}^{n} P_i$*, such that* $\forall i = 1, \ldots, n \ |a \cap P_i| \leq 1$.

When we write $a = \{p_i\}_{i\in I}$, $I \in 1 \ldots n$, we suppose that for each $i \in I$, $p_i \in P_i$. The interaction model is specified by a set of interactions $\gamma \subseteq 2^P$.

In this thesis, to simplify notation, we write for an interaction $\{p_1, \ldots, p_k\}$ the expression $p_1 \ldots p_k$. Furthermore, for a set of interactions $\{a_1, \ldots, a_n\}$ we write $a_1 + \cdots + a_n$.

We extend interactions with data transfer between the synchronizing components. For an interaction a, we use a guard G_a (boolean condition) and data transfer function F_a to specify data transfer.

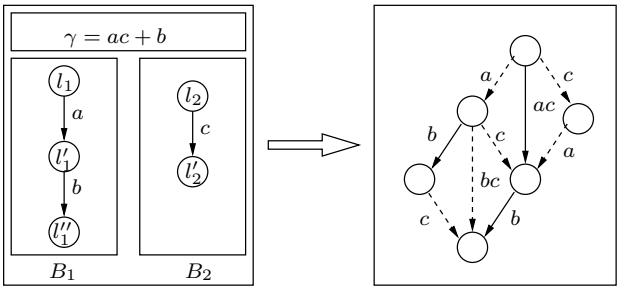


Figure 1.5: Composition by Interactions

Interactions can be *enabled* or *disabled*. An interaction is enabled iff its guard is true and all its ports are enabled. Contrarily, it is disabled iff its guard is false or at least one of its ports is disabled.

Example 3 *For the Temperature Control System presented in Example 1, the set of interactions between the Controller and two Rods is (Figure 1.4):* $\gamma = cool_1cool + cool_2cool + rest_1heat + rest_2heat + tick_1tick_2tick$.

We now provide the operational semantics for the composition of a system of behavior with respect to an interaction model.

Definition 5 (Parallel Composition) *Given n components* B_i=(L_i,P_i,$\mathcal{T}_i$, X_i, $\{g_\tau\}_{\tau\in\mathcal{T}_i}$, $\{f_\tau\}_{\tau\in\mathcal{T}_i}$) *and a set of interactions* γ, *we define* $B = \gamma(B_1, \ldots, B_n)$ *as the component* (L,γ, $\mathcal{T}$, $X, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i}$), *where:*

- $L = L_1 \times L_2 \times \ldots \times L_n$ *is the set of control locations,*
- $X = \bigcup_{i=1}^{n} X_i$ *is the set of variables,*
- $\mathcal{T}$ *is a set of transitions of the form* $\tau = ((l_1, \ldots, l_n), a, g_\tau, f_\tau, (l'_1, \ldots, l'_n))$ *obtained by synchronization of sets of transitions* $\{\tau_i = (l_i, p_i, g_{\tau_i}, f_{\tau_i}, l'_i) \in \mathcal{T}_i\}_{i\in I}$ *such that* $\{p_i\}_{i\in I} = a \in \gamma$ *and* $l'_j = l_j$ *if* $j \notin I$, *for arbitrary* $I \subseteq \{1, ..., n\}$, *the associated guard and function are respectively* $g_\tau = G_a \wedge \bigwedge_{i\in I} g_{\tau_i}$ *and* $f_\tau = F_a \wedge \bigwedge_{i\in I} f_{\tau_i} \wedge \bigwedge_{i\notin I}(X'_i = X_i)$.

The obtained behavior $B = \gamma(B_1, \ldots, B_n)$ can execute a transition $a \in \gamma$, iff for each $i \in I$, the action $a \cap P_i$ is enabled in B_i. The states of the transition system that do not participate in the interaction a remain unchanged. Notice that for $\gamma_\perp = \sum_{i=1}^{n} \sum_{p\in P_i} p$, the component $\gamma_\perp(B_1, \ldots, B_n)$ is the transition system obtained by interleaving the transitions of atomic components.

Example 4 *Figure 1.5 provides an example on the composition of two components* B_1, B_2 *by a set of interactions* $\gamma = ac + b$. *The right side is the composed behavior obtained after the application of the interactions. The overall graph (including solid and dotted arrows)*

shows the product of the two behaviors. The solid arrows are the transitions allowed by the interactions and the graph with these solid arrows presents the composition of two behaviors by the interactions. The dotted transitions are not legal.

In BIP, interactions are structured by connectors. A *connector* is a set of ports characterizing a set of interactions. For example, if p_1, p_2, p_3 are ports of distinct atomic components, then the connector γ consisting of $\{p_1, p_2, p_3\}$ has seven possible interactions $p_1 + p_2 + p_3 + p_1p_2 + p_1p_3 + p_2p_3 + p_1p_2p_3$. To specify the feasible interactions of a connector γ, we use two types of synchronizations:

- Strong synchronization or rendez-vous, when the only feasible interaction of γ is the maximal one, i. e., it contains all the ports of γ.
- Weak synchronization or broadcast, when feasible interactions are all those containing a complete port which initiates the broadcast. That is, if γ consists of $\{p_1, p_2, p_3\}$ and the broadcast is initiated by p_1, then the feasible interactions are $p_1 + p_1p_2 + p_1p_3 + p_1p_2p_3$.

It is possible to represent any arbitrary interaction through a connector by structured combination of the above two basic synchronization protocols.

To characterize these synchronizations, we associate two types with complete and incomplete ports. A feasible interaction of a connector is a set of its ports such that either it contains some complete port, or it is maximal, i.e, consisting of all the incomplete ports.

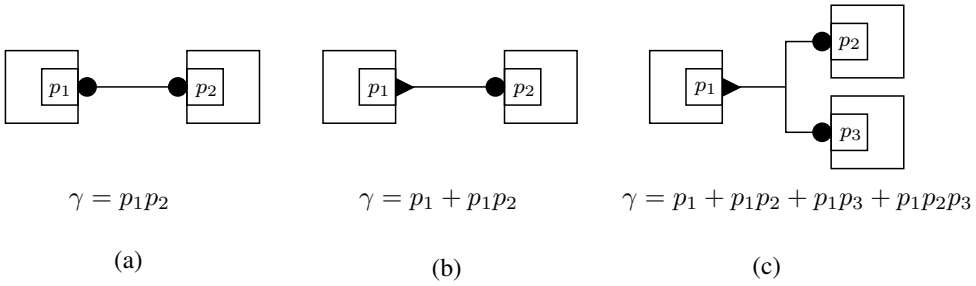


Figure 1.6: Connectors and their interactions.

Example 5 *Example of sets of connectors and their feasible interactions are shown in figure 1.6:*

- *In (a), the connector consists of two incomplete ports p_1 and p_2, hence the only feasible interaction is p_1p_2. It represents a rendez-vous, meaning that both actions are necessary for the synchronization.*
- *In (b), the interaction between p_1 and p_2 is asymmetric. p_1 is a complete port and can occur alone, even if p_2 is not possible. Nevertheless, p_2 is an incomplete port and it needs to synchronize with p_1 to occur. The feasible interactions are $\gamma = p_1 + p_1p_2$.*

- *In (c), the interactions between p_1, p_2 and p_3 are also asymmetric. The port p_1 is complete, it can occur alone or synchronize with either or both p_2 and p_3, hence the feasible interactions are $\gamma = p_1 + p_1p_2 + p_1p_3 + p_1p_2p_3$.*

The textual for connector description in BIP is as follows:

```
interaction::= port_id+
connector::=
  connector connector_id = port_id+
    [complete interaction]
    behavior
      [on interaction [provided guard] [do statement]]+
    end
```

That is, the keyword **connector** is followed by a name and a set of ports. The keyword **complete** define a minimal set of ports to be considered as complete interaction. Then any superset (including) of ports of that minimal set corresponds to a feasible interaction. Since a connector contains a set of interactions, the behavior is defined following the keyword **behavior** for each interaction: the keyword **on** is followed by an interaction together with its guard and its function.

If the definition of complete interaction is omitted, then all interactions containing a complete port (if it exists) are feasible, or only the maximal interaction (if all the ports are incomplete) is feasible.

1.3.3 Priorities

Priorities are a powerful tool for enforcing a given property by restricting nondeterminism. It allows selecting interactions to be executed amongst the feasible ones based on the current global state of the system. The definition of a priority, followed by the composition of behaviors using the priority glue are provided below.

Definition 6 (Priority) *A priority is a relation $\prec \subseteq \gamma \times L \times \gamma$, where γ is the set of interactions, and L is the global set of locations. We write $a \prec_l a'$ for $(a, l, a') \in \prec$. Furthermore, we require that for all $l \in L$, $\prec_l$ is a strict partial order on γ. $a \prec_l a'$ means that interaction a has less priority than a' at location l.*

The textual for the description of priorities is defined by:

```
priority::=
  priority [priority_id [if cond] interaction < interaction]+
```

That is, priorities are a set of rules, each consisting of an ordered pair of interactions associated with a condition (*cond*). The condition is a boolean expression in C on the variables of the components involved in the interactions. When the condition holds and both interactions are enabled, only the higher one is eligible for execution. Conditions can be omitted for static priorities.

1.3.4 Composite Components

In BIP, a composite component allows defining new components which consist of:

- a set of instances of existing sub-components (atomic or composite).
- a set of connectors between the component instances.
- a set of priorities between the interactions.

The BIP textual of a composite component is defined by:

```
composite component::=
  component component_id
    {contains component_id {instance_id[parameters]}+}+
    [connector]+
    priority
  end
```

The instances can have parameters providing initial values to their variables through a named association.

Finally, we consider systems defined as parallel composition of components together with an initial state.

Definition 7 (System) *A system $\mathcal{S}$ is a pair $\langle B, Init\rangle$ where B is a component and $Init$ is the initial state of B.*

We seperate $Init$ from components because we want to reuse components. A component type is used to build different parts of a system or different systems. And depending on the system, the component might have different initial states. Hence the separation of initial states and components provides flexibility in the reuse of components.

We can consider $Init$ in the form of state, that is the set of initial states of components in B, or in the form of a state predicate characterizing the initial state of B.

Example 6 *The components in Temperature Control presented in Example 1 are composed by using the following set of interactions, indicated by connectors in the Figure 1.4:*

$$\gamma = tick\ tick_1\ tick_2 + cool\ cool_1 + cool\ cool_2 + heat\ rest_1 + heat\ rest_2$$

The initial state of the system is $Init = (l_5 \wedge (\theta = 100), l_1 \wedge (t_1 = 3600), l_3 \wedge (t_2 = 3600))$. $Init$ can also be represented in the form of initial condition as $Init = (l_5 \wedge (\theta = 100)) \wedge (l_1 \wedge (t_1 = 3600)) \wedge (l_3 \wedge (t_2 = 3600))$.

The Temperature Control System is represented by $S = \langle B, Init\rangle$ where B is the Temperature Control composite component described in BIP language as follows:

```
component TemperatureController
  contains Rod        rod1, rod2
  contains Controller ctrl

  connector tick = ctrl.tick, rod1.tick1, rod2.tick2
    behavior
      on ctrl.tick, rod1.tick1, rod2.tick2
        provided true do {# #}
    end
  connector cool1 = ctrl.cool, rod1.cool1
    behavior
      on ctrl.cool, rod1.cool1 provided true do {# #}
    end
  connector cool2 = ctrl.cool, rod2.cool2
    behavior
      on ctrl.cool, rod2.cool2 provided true do {# #}
    end
  connector heat1 = ctrl.heat, rod1.rest1
    behavior
      on ctrl.heat, rod1.rest1 provided true do {# #}
    end
  connector cool1 = ctrl.heat, rod2.rest2
    behavior
      on ctrl.heat, rod2.rest2 provided true do {# #}
    end
end
```

1.4 Properties of BIP Components

1.4.1 Invariants

A state predicate I is an invariant of a system S, if every reachable state of the system S satisfies I. In other words, each state that is reached during the computation of S satisfies I. We first define the set of reachable states of a system.

Definition 8 (Reachable States) *Given a system $S = \langle B, Init \rangle$ where B is a component and $Init$ is the initial state of B. A state l is called reachable (accessible) in S if from the initial state there exists an execution sequence $Init \xrightarrow{p_1} l_1 \xrightarrow{p_2} \ldots \xrightarrow{p_n} l_n$ such that $l_n = l$. We denote the set of reachable states of S by $Reach(S)$.*

Formally, we have the following definition of system invariants:

Definition 9 (Invariants of System) *Given a system S and its set of reachable states $Reach(S)$. A state predicate I is an invariant of S, denoted by $inv(S, I)$, if every state of $Reach(S)$ satisfies I.*

In component-based construction, components can be reused to build different systems or different parts of a system. And the initial states of different instances of a component type may be different. Therefore, we define here the notion of invariants of component. Then the invariants of a system are obtained from the invariants of components that are used to build the system depending on the initial state of the system.

Definition 10 (Invariants of Component) *Given a component* $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$*, a state predicate* I *is an invariant of* B*, denoted by* $inv(B, I)$*, if for any state* $l \in L$ *and any port* $p \in P$*,* $I(l)$ *and* $l \xrightarrow{p} l' \in \mathcal{T}$ *imply* $I(l')$*, where* $I(l)$ *means that* l *satisfies* I*.*

That is, a state predicate I is an invariant of a component B if for any state l of B that satisfies I, all the states reached from l also satisfy I.

For a system $S = \langle B, Init\rangle$, all the states reached from the initial state are the set of reachable states of the system. Therefore by Definitions 9 and 10, any invariant of B that is satisfied by the initial state is also invariant of the system S.

Proposition 1 *Given a system* $S = \langle B, Init\rangle$ *where* B *is a component and* $Init$ *is the initial state, then any invariant* I *of* B *is also invariant of* S*, denoted by* $inv(S, I)$*, if the initial state* $Init$ *satisfies* I*.*

Proof The proposition is proven from the facts that $Init$ satisfies I and because I is an invariant of B, according to Definition 10 all the states reached from $Init$ satisfies I.

We extensively use the following well-known results about invariants.

Proposition 2 *Let* I_1, I_2 *be two invariants of a component* B*. Then* $I_1 \wedge I_2$*,* $I_1 \vee I_2$ *are invariants of* B*.*

Proof If l is a state of B such that $(I_1 \wedge I_2)(l)$, we have $I_1(l)$ and $I_2(l)$. For any successor l' of l, we have $I_1(l')$ and $I_2(l')$ because I_1 and I_2 are invariants of B, therefore $(I_1 \wedge I_2)(l')$. Similarly for $I_1 \vee I_2$.

Similarly if I_1 and I_2 are invariants of a system S, then $I_1 \wedge I_2$ and $I_1 \vee I_2$ are invariants of S.

The proof that a given predicate I is an invariant of a given component B can be done by finding a stronger invariant, i.e, a predicate that is an invariant of the component B and implies the predicate I. We have the following proposition:

Proposition 3 *Given a predicate* I *of a component* B*. If there exists an invariant* I' *of* B *such that* $I' \Rightarrow I$ *then* I *is also an invariant of the component* B*.*

1.4.2 Local Deadlock-freedom

In the rest of the thesis, we consider control locations of atomic components as boolean variables.

Definition 11 (Deadlock-free States) *Given a component* $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$*, we present by DFS the state predicate characterizing deadlock-free states:*

$$DFS = \bigvee_{l\in L} \bigvee_{\tau\in l^\bullet} en(\tau) \; where \; en(\tau) = l \wedge g_\tau$$

The predicate $en(\tau)$ of a transition $\tau = (l, p, g_\tau, f_\tau, l')$ characterizes a set of states from which the transition τ is enabled, i.e the component is at its source location l and its guard g_τ is true. The following lemma gives a useful characterization of DFS:

Lemma 1 *Given a component* $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$*,*

$$DFS = \bigwedge_{l\in L} (l \Rightarrow \bigvee_{\tau\in l^\bullet} g_\tau) = \bigvee_{l\in L} (l \wedge \bigvee_{\tau\in l^\bullet} g_\tau)$$

Proof The proof is based on the fact that $\bigvee_{l\in L} l$ and $\neg(l \wedge l')$ for $l \neq l'$.

The predicate DFS characterizes the deadlock-freedom property of a component. A component is deadlock-free if whenever it reaches a location, it always can go out by at least one of its outgoing transitions, i.e the guard of at least one outgoing transition is true.

1.4.3 Global Deadlocks

A system has global deadlocks if at some global state, there is no interaction that can be executed. The global deadlocks therefore depend on the enabledness of the interactions. Hence we first define the enabledness condition of an interaction.

Definition 12 (Enabledness) *Given a component* $B = \gamma(B_1, \ldots, B_n)$ *we define, for each interaction* $a \in \gamma$*, an enabledness predicate under which the interaction* a *is feasible as follows:*

$$en(a) = \bigwedge_{p\in a} en(p) \quad where \quad en(p) = \bigvee_{port(\tau)=p} en(\tau)$$

$port(\tau)$ for a transition τ is the port labeling that transition.

That is, $en(a)$ characterizes all the states from which interaction a can be executed. The interaction a can be executed if all its ports are ready for synchronizing and a port is ready if at least one of its transitions is enabled.

We now define a predicate called DIS that characterizes a set of global deadlocks of a system.

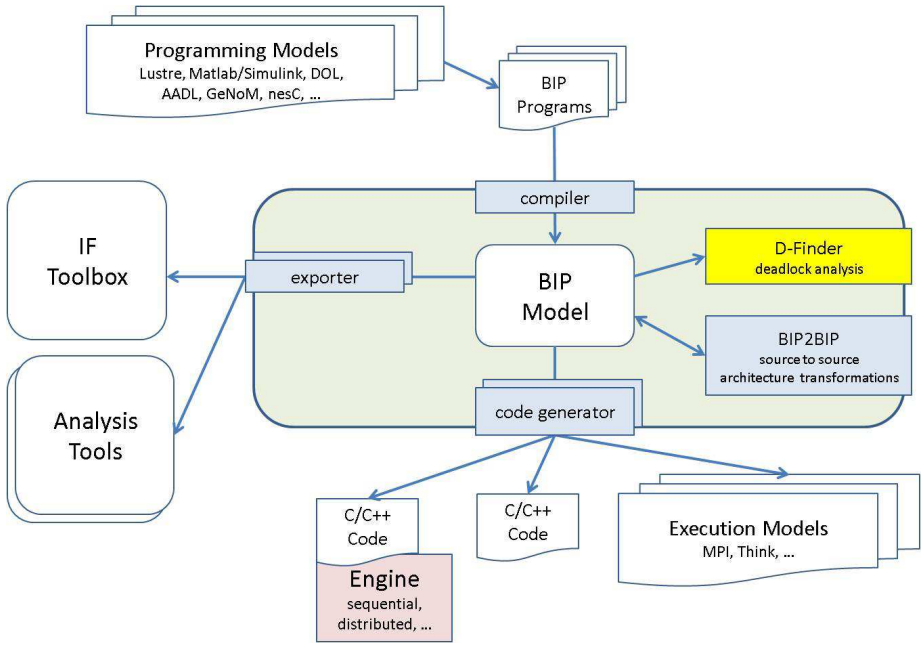


Figure 1.7: The BIP Tool-Chain

Definition 13 (Deadlock States) *We define the predicate DIS characterizing the set of the states of $\gamma(B_1, \ldots, B_n)$ from which all interactions are disabled:*

$$DIS = \bigwedge_{a \in \gamma} \neg en(a)$$

Example 7 *For the Temperature Control System (see Figure 1.4), we have:*

$$\begin{aligned} DIS \;=\; & (\neg(l_5 \wedge \theta < 1000)) \bigwedge (\neg(l_6 \wedge \theta = 100) \vee \neg l_2) \\ \bigwedge\; & (\neg(l_6 \wedge \theta > 100)) \bigwedge (\neg(l_5 \wedge \theta = 1000) \vee \neg(l_3 \wedge t_2 \geq 3600)) \\ \bigwedge\; & (\neg(l_5 \wedge \theta = 1000) \vee \neg(l_1 \wedge t_1 \geq 3600)) \bigwedge (\neg(l_6 \wedge \theta = 100) \vee \neg l_4) \end{aligned}$$

The system $\langle \gamma(B_1, \ldots, B_n), Init \rangle$ is deadlock-free if the predicate $\neg DIS$ is an invariant of the system.

1.5 BIP Tool-Chain

The BIP tool-chain provides a set of tools for the modeling, the execution, the verification and the static transformation of BIP models.

The overview of the BIP tool-chain is shown in Figure 1.7. It includes the following tools:

- An *editor*, for describing textually a system in BIP language.

- A *compiler*, for generating a BIP model from BIP description source.
- A *code generator*, for generating, from a model, C++ code executable on the BIP engine. The code-generator can also produce THINK specification [PPRS06], from which the Think tool-chain can generate code to be executed over a choice of target platforms.
- *D-Finder*, of which the method and the implementation are presented in this thesis, is a compositional verification tool for component-based systems described in BIP language [BBNS09].
- *BIP2BIP transformations*, allow useful transformations which generate an efficient monolithic component from a composite component [BJS09].
- An *exporter* to connect with external tools such as IF toolbox or analysis tools.
- A set of translators from other languages (Lustre, Matlab/Simulink, ect.) to BIP. For example, an *AADL-to-BIP* translation from Architecture Analysis & Design Language (AADL) into BIP [CRBS08], allows simulation of systems specified in AADL and application to these systems of formal verification techniques developed for BIP, e.g. deadlock detection.

The editor, compiler and code generator form the front-end of the tool-chain. The back-end provides a platform for analyzing and executing the C++ application code which is generated by front-end. The back-end includes an engine and the associated software infrastructure. The engine is a controller which selects and executes interactions between the components. First it considers the states of the components and the interaction model to find all the enabled interactions. Then it applies the priority rules to eliminate lower priority interactions, then chooses one amongst the maximal enabled for execution.

1.6 Summary

Component-based approach is aimed to deal with the complexity of systems. It is based on the idea of building a complex system by assembling basic components (blocks). It provides important characteristics for system construction such as reuse, incrementality, compositionality, etc. It allows not only the reuse of components but also the reuse of known properties of constituent components.

We have presented BIP, a component-based framework for modeling heterogeneous systems. The BIP component model is the superposition of three layers: the lower layer describes the behavior of a component as a transition system; the intermediate layer consists of the interactions between transitions of the layer underneath; the upper layer describes the priorities characterizing a set of scheduling policies for interactions. Such a layering offers a clear separation between components' behaviors and the structure of the system (interactions and priorities).

BIP modeling framework allows dealing with complexity of systems by providing incremental composition of heterogeneous components. It also considers correctness-by-construction for a class of essential properties such as deadlock-freedom [GS05].

We have also presented several important properties of BIP components such as invariants, deadlock-freedom. We have defined predicates characterizing deadlock-freedom of atomic components as well as of systems.

The BIP tool-chain has been developed providing automated support for component integration and generation of glue code meeting given requirements. Efficient model transformations, verification methods have also been studied and implemented in the BIP tool-chain.

We are now going to present our compositional verification method for component-based systems. We will also show applications of our method for verifying systems described in the BIP language.

Part II

Verification Method

CHAPTER 2

Compositional Verification

Contents

Formal verification based on Model-Checking nowadays suffers from the state space explosion problem because of the growing size and complexity of systems. Compositional verification approach is used for alleviating this problem. The idea is to apply "divide-and-conquer" techniques to infer global properties of a system from properties of its subsystems. Instead of verifying globally the entire system, compositional approach first decomposes it into small subsystems and verifies each of them individually. The size of a subsystem is often quite smaller compared to the size of the whole system, hence there is less risk of explosion of state space. Then, properties of the global system are inferred from the verified properties of its subsystems.

In this chapter, we present a compositional method for the verification of safety properties for component-based systems described in a subset of the BIP language encompassing multi-party interactions. The BIP framework allows to define rich interaction models by using hierarchical interactions extended with data transfer as presented in [Bas08]. However in this chapter, we restrict to pure synchronizations, i.e. synchronizations without data transfer. The absence of hierarchy is not a real limitation, as long as hierarchical interaction models can be statically transformed into equivalent flat interaction models with

a potential increased number of interactions [BJS09]. For the systems with data transfer between components, we will present a method to deal with them in Chapter 4.

The organization of this chapter is as follows: Section 1 presents the compositional verification rule together with methods for computing invariants. We then present in Section 2 an abstraction technique that we use to deal with systems with data. Section 3 shows the procedure of the verification method for checking safety properties. In Section 4 we provide an application of the method for checking deadlock-freedom of component-based systems described in BIP. And we finish this chapter by giving conclusions in Summary. We use the Temperature Control System presented in the previous chapter as running example through the chapter for illustrating invariant computation, abstraction and deadlock-freedom checking.

2.1 Compositional Verification Method

2.1.1 Compositional Verification Rule

We propose a compositional method for verifying safety properties. The idea of the method is based on the inference rules represented in Equations 2.1 and 2.2. The rule in Equation 2.1 says that if the initial state $Init$ of a system S satisfies a predicate Φ and Φ is preserved by every transition τ of the system, that is every reachable state of S satisfies Φ, then Φ is an invariant of S. Such invariant Φ is called *inductive invariant*. Unfortunately, most of invariants are not inductive, that is they are not preserved by every transition. Therefore, an extension rule is proposed in Equation 2.2 which allows proving the invariance property of Φ by finding an auxiliary predicate Φ^{aux} such that : (1) Φ^{aux} is stronger than Φ; (2) Φ^{aux} is satisfied by the initial state; and (3) Φ^{aux} is preserved by every transition of the system.

$$\frac{Init \models \Phi \quad \{\Phi\}\tau\{\Phi\} \quad \forall \tau \in S}{S \models \Box\Phi} \tag{2.1}$$

$$\frac{Init \models \Phi^{aux} \quad \{\Phi^{aux}\}\tau\{\Phi^{aux}\} \quad \forall \tau \in S \quad \Phi^{aux} \Rightarrow \Phi}{S \models \Box\Phi} \tag{2.2}$$

The best solution as an auxiliary predicate Φ^{aux} is the set of reachable states $Reach(S)$. However, the computation of the reachable state set causes the well-known state space explosion problem. Hence another solution, on which our method is based, is to take an over-approximation $Reach_{App}(S)$ of the reachable state set. If $Reach_{App}(S)$ implies the predicate Φ, then the system S satisfies Φ.

The goal of the method is therefore to compute a strong-enough over-approximation of the set of reachable states to be able to prove invariance properties. Our method for computing such approximation focuses on two aspects of systems: one is the local behavior of atomic components; the other is the global constraints between atomic components based on the interactions between them. Since over-approximation of the reachable state set can be characterized by invariants, we exploit two kinds of invariants:

- *Component invariants* Φ_i of B_i which are over-approximations of components' reachability sets. They are computed by forward propagation techniques.

- *Interaction invariants* Ψ which are global constraints on the states of atomic components involved in interactions. Interactions force the synchronized atomic components to move together from a set of states to another set of states, hence there are explicit constraints on the states of these atomic components. Interaction invariants captures these constraints by statically exploiting the structure of the interaction set. They are computed from Boolean Behavioral Constraints which are a set of implications obtained from interactions and local behavior.

The main rule of our approach is as follows:

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma\{B_i\}_i < \Phi >} \tag{2.3}$$

The rule 2.3 allows to prove invariance of a predicate Φ for a system obtained by using a n-ary composition operation parameterized by a set of interactions γ on a set of components $\{B_i\}_i$. It uses global invariants which are the conjunction of component invariants $\{\Phi_i\}_i$ and interaction invariants Ψ. The verification of invariance-property Φ is then done by checking tautology $(\bigvee_i \Phi_i) \wedge \Psi \Rightarrow \Phi$ or equivalently the unsatisfiability of $(\bigvee_i \Phi_i) \wedge \Psi \wedge (\neg\Phi)$.

Methods for computing component invariants and interaction invariants will be presented in the next sub-sections.

2.1.2 Component Invariants

Component invariants are over-approximation of the set of reachable states of components. If an atomic component B is finite state with the initial state $Init$, then we can take $\Phi = Reach(\langle B, Init\rangle)$, the set of reachable states of B or any upper approximation of $Reach(\langle B, Init\rangle)$. If the components are infinite state, $Reach(\langle B, Init\rangle)$ can be approximated as shown in [LBBO01]. In this section, we present a lightweight method for the computation of sequences of increasingly stronger inductive invariants for atomic components. Component invariants are computed by using the *post* predicate transformer which defines the propagation of a predicate through a transition or a transition system. The *post* predicate transformer allows computing state successors of atomic components. Its formal definition is as follows :

Definition 14 (Post Predicate Transformer w.r.t Transition) *Given a component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ and a predicate φ on the set of variables X. We define a post predicate transformer of φ w.r.t a transition $\tau = (l, p, g_\tau, f_\tau, l') \in \mathcal{T}$, which is a propagation of φ by the transition τ, as follows:*

$$post_\tau(\varphi)(X) = \exists X'.\varphi(X') \wedge g_\tau(X') \wedge f_\tau(X', X)$$

Transition τ can be executed only if its guard $g_\tau(X')$ (X' is the previous valuation of X at the source location l) is true. The run of the transition executes function $f_\tau(X', X)$ which updates the value of the predicate $\varphi(X')$ at the source location l and a new predicate $post_\tau(\varphi)(X)$ is produced at the destination location l'.

We define in a similar way, the *pre* predicate transformer for a transition τ, $pre_\tau(\varphi)(X) = \exists X'.g_\tau(X) \wedge f_\tau(X, X') \wedge \varphi(X')$.

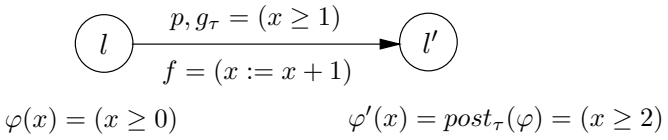


Figure 2.1: An example of post predicate

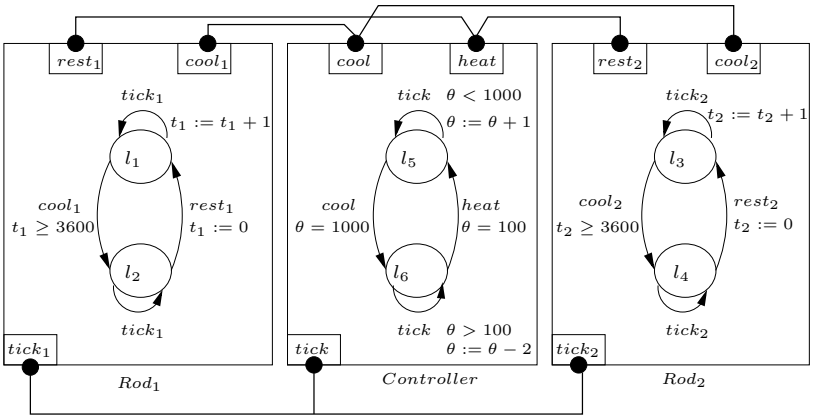


Figure 2.2: Temperature Control System

Example 8 *Figure 2.1 illustrates a propagation of a predicate $\varphi = (x \geq 0)$ by a transition τ from l to l' with its guard $g_\tau = (x \geq 1)$ and its update function $f_\tau = (x := x+1)$. The post condition of φ with respect to the transition τ is $\varphi'(x) = post_\tau(\varphi)(x) = \exists x'.(x' \geq 1) \wedge (x = x' + 1) \wedge (x' \geq 0) = (x \geq 2)$. $\varphi'(x)$ is the propagation of $\varphi(x)$ through the transition τ.*

If τ is a loop transition, i.e it is of the form $\tau = (l, p, g_\tau, f_\tau, l)$, we can use $post^*$ predicate transformer which defines the iterative propagation by the loop transition according to the number of iterations. The condition for using $post^*$ is that we can find a *transitive closure* $\mathcal{F}_\tau(n, X', X)$ [CJ98, BIL09] of the function $f_\tau(X', X)$ where n is the number of iterations. For example, if the function f_τ is of the form $x = x' + a$ where a is a constant, then its transitive closure is $\mathcal{F}_\tau(n, x', x) = (x = x' + a * n)$. It means that, an iteration step increases the value of x by a, hence after n times of iterations, the value of x is increased by $a * n$. The predicate transformer of a predicate $\varphi(x)$ over τ is $post^*_\tau(x) = \exists n \exists x'.(n \geq 1) \wedge \varphi(x') \wedge g_\tau(x' + a * (n-1)) \wedge (x = x' + a * n)$.

Example 9 *Consider the component Controller in Figure 2.2 which has a loop transition $\tau = (l_5, tick, \theta < 1000, \theta = \theta + 1, l_5)$. The predicate to be propagated by the loop transition is $\varphi = (\theta = 100)$ which is obtained from the post predicate transformer of heat incoming transition. The loop transition τ can occur if the guard $\theta < 1000$ is true and it increases the value of the variable θ by 1. The value of θ after n times of iterations is $(100 + 1 * n)$. The $post^*_\tau$ of φ w.r.t the transition τ is $post^*_\tau(\varphi)(\theta) = \exists n \exists \theta'.(n \geq 1) \wedge (\theta' = 100) \wedge (\theta' + 1 * (n-1) < 1000) \wedge (\theta = \theta' + 1 * n) = (101 \leq \theta \leq 1000)$.*

Consider a predicate φ_l at each control location l of a component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, the global predicate is $\bigvee_{l\in L}(l \wedge \varphi_l)$. The transformation of the global predicate is done by the propagation on all the transitions of the component.

Definition 15 (Post Predicate Transformer w.r.t Transition System) *Given a component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ and a predicate $\Phi = \bigvee_{l\in L}(l \wedge \varphi_l)$ where φ_l is a predicate at control location l, we define the post predicate transformer of Φ w.r.t to the transition system of B as follows:*

$$post(\Phi) = \bigvee_{l\in L}\left(\bigvee_{\tau=(l,p,l')}(l' \wedge post_\tau(\varphi_l))\right)$$

.

Equivalently, we have that $post(\Phi) = \bigvee_{l\in L} l \wedge (\bigvee_{\tau=(l',p,l)} post_\tau(\varphi_{l'}))$. This allows computing $post(\Phi)$ by forward propagation of the assertions associated with control locations in Φ.

For a component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, $\varphi_l = true$ is the weakest invariant at each control location of S, and $\Phi = \bigvee_{l\in L} l \wedge true$ is the weakest component invariant. The iterative forward propagation of this predicate by transition system provides a stronger invariant. The iteration terminates when a fix point is reached. The following proposition provides such means to compute increasingly stronger invariants of a component.

Proposition 4 *Given a system $\mathcal{S} = \langle B, Init\rangle$ where $Init$ is the initial condition of the component B, the following iteration defines a sequence of increasingly stronger inductive invariants:*

$$\Phi_0 = true \quad \Phi_{i+1} = Init \vee post(\Phi_i)$$

Proof By induction. Φ_0 is an inductive invariant. If Φ_i is an inductive invariant then $Init \vee post_\tau(\Phi_i) \Rightarrow \Phi_i$. As *post* is monotonic and distributes over disjunction, $post_\tau(\Phi_{i+1}) = post(Init \vee post(\Phi_i)) \Rightarrow post(\Phi_i) \Rightarrow \Phi_{i+1}$. Moreover, $Init \Rightarrow \Phi_{i+1}$. So Φ_{i+1} is an inductive invariant.

We use different strategies for producing such invariants. We usually iterate until we find good enough invariants. The good enough invariants mean that they are able to prove some safety properties. This will be explained in Section 2.3.

Example 10 *For the Temperature Control System of figure 2.2, the predicates $\Phi_1 = (l_1 \wedge t_1 \geq 0) \vee (l_2 \wedge t_1 \geq 3600)$ and $\Phi_2 = (l_3 \wedge t_2 \geq 0) \vee (l_4 \wedge t_2 \geq 3600)$ are respectively inductive invariants of the Rod_1 and Rod_2 components given the initial conditions $Init_1 = l_1 \wedge t_1 = 3600$ and $Init_2 = l_3 \wedge t_2 = 3600$. Also, the predicate $\Phi_3 = (l_5 \wedge 100 \leq \theta \leq 1000) \vee (l_6 \wedge 100 \leq \theta \leq 1000)$ is a non-inductive invariant of the Controller component, given the initial condition $Init_3 = l_5 \wedge \theta = 100$. An auxiliary inductive invariant that implies Φ_3 is $\Phi_3^{aux} = (l_5 \wedge 100 \leq \theta \leq 1000) \vee (l_6 \wedge 100 \leq \theta \leq 1000) \wedge (\theta$ is even$)$.*

A key issue is efficient computation of such invariants as the precise computation of *post* requires quantifier elimination. An alternative to quantifier elimination is to compute over-approximations of *post* based on syntactic analysis of the predicates. In this case, the obtained invariants may not be inductive. We present below lightweight techniques for computing component invariants by avoiding quantifier elimination process.

Lightweight Computation of Component Invariants

We provide a brief description of a syntactic technique used for approximating *post* predicate transformer for a transition τ. The idea is that the *post* can be approximated by predicates which are not affected by the update function of the transition, therefore the methods focuses on finding such predicates. Theses techniques and also some other techniques for generating *post* predicate transformer and component invariants are proposed and well presented in [BL99].

Given a component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, consider a transition $\tau = (l, p, g_\tau, f_\tau, l') \in \mathcal{T}$ where its guard g_τ is a predicate of the form $g_\tau(Y)$, f_τ is of the form $Z' = e_\tau(U)$; Y, Z, U are subsets of the set of variables X. If $Z \cap U = \emptyset$ which means that variables in the set U are not affected by the associate update function e_τ, then the predicate $Z = e_\tau(U)$ is preserved by the transition τ. Similarly, if $Z \cap Y = \emptyset$, the predicate $g_\tau(Y)$ is also preserved by the transition. Moreover, for an arbitrary predicate φ at l', we find a decomposition $\varphi = \varphi_1(Y_1) \wedge \varphi_2(Y_2)$ such that $Y_2 \cap Z = \emptyset$ i.e. $\varphi_2(Y_2)$ is not affected by the update function f_τ, then the predicate $\varphi_2(Y_2)$ still holds after the execution of the transition.

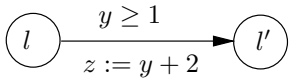
To formulate the general case, given a transition τ above, we denote by $func(\tau)$ the predicate $Z = e_\tau(U)$ and by $guard(\tau)$, the guard $g_\tau(Y)$. For a transition $\tau = (l', p, g(Y), Z' = e(U), l)$ we have $Z \cap (Y \cup U) = \emptyset$ and for an arbitrary predicate φ at l', we find a decomposition $\varphi = \varphi_1(Y_1) \wedge \varphi_2(Y_2)$ such that $Y_2 \cap Z = \emptyset$. Then, the *post* predicate transformer can approximated by $post^a_\tau = \varphi_2(Y_2) \wedge guard(\tau) \wedge func(\tau)$.

The required condition for Φ_l to be an invariant is $Z \cap (Y \cup U) = \emptyset$. If this condition is not satisfied, by considering separately the condition for $Z \cap Y$ and $Z \cap U$, we can get $post^a_\tau$ as follows:

$$post^a_\tau(\varphi) = \varphi_2(Y_2) \wedge \begin{cases} guard(\tau) \wedge func(\tau) & \text{if } Z \cap (Y \cup U) = \emptyset \\ func(\tau) & \text{if } Z \cap U = \emptyset \text{ and } Z \cap Y \neq \emptyset \\ guard(\tau) & \text{if } Z \cap Y = \emptyset \text{ and } Z \cap U \neq \emptyset \\ true & \text{otherwise} \end{cases} \tag{2.4}$$

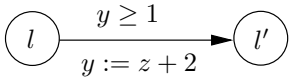
Example 11 *Figure 2.3 illustrates examples for the cases defined in Equation 2.4. In all the transitions, the predicate at the source location is $\varphi(y, z) = y \geq 0 \wedge x \geq 0$ which can be decomposed into $\varphi(y) = y \geq 0$ and $\varphi(z) = z \geq 0$.*

- *In example (a), the variable y of the guard is not affected by f_τ, hence $post^a_\tau(\varphi) = \varphi(y) \wedge func(\tau) \wedge guard(\tau) = (z = y + 2) \wedge (y \geq 1)$.*



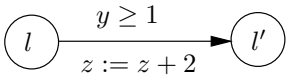
(a)

$\varphi = (y \geq 0) \wedge (z \geq 0)$ $\quad post^a_\tau(\varphi) = (y \geq 1) \wedge (z = y + 2)$



(b)

$\varphi = (y \geq 0) \wedge (z \geq 0)$ $\quad post^a_\tau(\varphi) = (z \geq 0) \wedge (y = z + 2)$



(c)

$\varphi = (y \geq 0) \wedge (z \geq 0)$ $\quad post^a_\tau(\varphi) = (y \geq 1)$

Figure 2.3: Examples of $post^a_\tau$

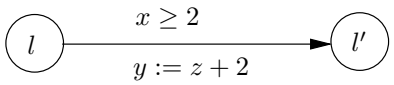
- *In example (b), the variable y of the guard is changed through the transition but the variable z on the right side of f is unchanged, hence $post^a_\tau(\varphi) = \varphi(z) \wedge func(\tau) = (z \geq 0) \wedge (y = z + 2)$.*
- *In the third example (c), y of the guard stays unchanged through the transition but the variable z appears on both sides of the function, therefore $post^a_\tau(\varphi) = \varphi(y) \wedge guard(\tau) = (y \geq 1)$.*

In the case $Z \cap Y \neq \emptyset$ or $Z \cap U \neq \emptyset$, we still can apply the above rules by decomposing Y and U into two parts: one part is disjoint with Z and the other part is not disjoint with Z.

Consider a transition $\tau = (l, p, g_\tau, f_\tau, l')$ of a component B. Assume that its guard is of the form $g_\tau(Y)$ and the associated update function f_τ is of the form $Z'_1 = e_\tau(U) \wedge Z'_2 = Z_2$ where $Y, Z_1, Z_2, U \subseteq X$. Z'_1 and Z'_2 are respectively next valuations of Z_1 and Z_2. For an arbitrary predicate φ, we find a decomposition $\varphi = \varphi_1(Y_1) \wedge \varphi_2(Y_2)$ such that $Y_2 \cap Z_1 = \emptyset$, the *post* predicate transformer for the transition τ can be approximated as follows:

$$post^a_\tau(\varphi) = \varphi_2(Y_2) \wedge \left\{ \begin{array}{cc} g_\tau(Y) & \text{if } Z_1 \cap Y = \emptyset \\ true & \text{otherwise} \end{array} \right\} \wedge \left\{ \begin{array}{cc} Z_1 = e_\tau(U) & \text{if } Z_1 \cap U = \emptyset \\ true & \text{otherwise} \end{array} \right\} \quad (2.5)$$

Example 12 *Figure 2.4 is an example where we can apply Equation 2.5. The transition τ has a guard $g_\tau = (x \geq 2)$, a function $f_\tau = (y := z + 2); (t := t + 2)$. Consider a predicate $\varphi = (x \geq 0) \wedge (y \geq 2)$ which can be split into two parts: $\varphi_1 = (x \geq 0)$ where x is unchanged through the transition and $\varphi_2 = (y \geq 2)$ where y is affected by function f_τ. Similarly, the update function can be decomposed into two parts: $f_1 = (y := z + 2)$ where variable on the right side is different from the left and $f_2 = (t := t + 2)$ where the variable t is on both the right and the left sides. Finally we have $post^a_\tau(\varphi) = \varphi_1 \wedge guard(\tau) \wedge func_1(\tau) = (x \geq 2) \wedge (y = z + 2)$.*



$\varphi(x) = (x \geq 0) \wedge (y \geq 2)$ $\qquad$ $post^a_\tau(\varphi) = (x \geq 2) \wedge (y = z + 2)$

Figure 2.4: Examples of $post^a_\tau$

The following proposition says that, for a transition τ and a predicate φ, $post^a_\tau(\varphi)$ is an over approximation of the post predicate transformer $post_\tau(\varphi)$.

Proposition 5 *If τ and φ are respectively a transition and a state predicate as above, then $post_\tau(\varphi) \Rightarrow post^a_\tau(\varphi)$.*

Proof We can over-approximate successively $post_\tau(\varphi)$ as follows:

$$
\begin{array}{rcl}
post_\tau(\varphi)(X') & = & \exists X.\, \big(\varphi(X) \wedge g_\tau(X) \wedge f_\tau(X, X')\big) \\
& = & \exists Z_1, Z_2.\, \big(\varphi_1(Y_1) \wedge \varphi_2(Y_2) \wedge g_\tau(Y) \wedge Z'_1 = e_\tau(U) \wedge Z'_2 = Z_2\big) \\
& \Rightarrow & \exists Z_2.\, \big(\varphi_2(Y_2) \wedge Z'_2 = Z_2\big) \bigwedge \exists Z_1, Z_2.\, \big(g_\tau(Y) \wedge Z'_1 = e_\tau(U) \wedge Z'_2 = Z_2\big) \\
& = & \varphi_2(Y'_2) \bigwedge \exists Z_1, Z_2.\, \big(g_\tau(Y) \wedge Z'_1 = e_\tau(U) \wedge Z'_2 = Z_2\big) \\
& \Rightarrow & \varphi_2(Y'_2) \bigwedge \left\{ \begin{array}{ll} g_\tau(Y') & \text{if } Z_1 \cap Y = \emptyset \\ true & \text{otherwise} \end{array} \right\} \bigwedge \left\{ \begin{array}{ll} Z'_1 = e_\tau(U') & \text{if } Z_1 \cap U = \emptyset \\ true & \text{otherwise} \end{array} \right\} \\
& = & post^a_\tau(\varphi)(X').
\end{array}
$$

2.1.3 Interaction Invariants

For the sake of clarity, we present methods for computing interaction invariants of systems without data. For systems with data, the methods can be applied by using abstraction techniques which will be presented in the next section.

The idea of our compositional verification method, as explained in the method rule, is that we try to compute as precisely as possible over approximations of the set of reachable states. Given a system consisting of n atomic components $B_1, \ldots, B_n$ synchronized by a set of interactions γ, an over-approximation of the global reachable states can be obtained by the intersection of the component invariants of these atomic components. For example, figure 2.5(a) illustrates two components B_1 and B_2 strongly synchronized by a set of two interactions $\gamma = a_1a_2 + b_1b_2$. By taking $Init = l_1 \wedge l_3$ as initial condition, we have the set of reachable states $Reach(\langle \gamma(B_1, B_2), Init\rangle) = (l_1 \wedge l_3) \vee (l_2 \wedge l_4)$ (Figure 2.5(b)). The component invariants of the components B_1 and B_2 are respectively $\Phi_1 = (l_1 \vee l_2)$ and $\Phi_2 = (l_3 \vee l_4)$. The intersection $\Phi_1 \wedge \Phi_2 = (l_1 \vee l_2) \wedge (l_3 \vee l_4) = (l_1 \wedge l_3) \vee (l_1 \wedge l_4) \vee (l_2 \wedge l_3) \vee (l_2 \wedge l_4)$ is an over-approximation of the global reachable states. Unfortunately this over-approximation is often quite large and not strong enough to prove invariance properties because it does not take into account global constraints due to strong synchronizations between atomic

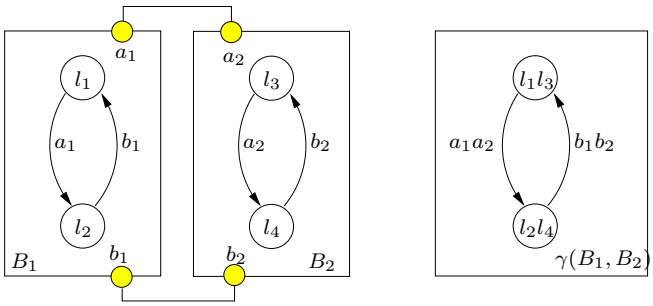


Figure 2.5: Two components with interactions (a) and its composition (b)

components. In this section we present a type of invariants called *interaction invariant* which captures these constraints and therefore represents a more precise over-approximation of the reachable states.

Interactions are used to restrict the global behavior of systems in order to meet given requirements. An interaction consists of a set of ports of different components. The actions of the ports of an interaction must occur simultaneously. There are therefore strong constraints on the moves of the synchronized components. Consider the example in Figure 2.5(a), at control locations l_1 and l_3, interaction a_1a_2 can take place and enforces two components moving together to control locations l_2 and l_4. From l_2 and l_4, interaction b_1b_2 can occur and similarly enforces two components back to l_1 and l_3. Here we have strong constraints between two components: if B_1 is at control location l_2 (respectively l_1), B_2 must be at control location l_4 (respectively l_3) and vice-versa. Interaction invariants characterize such constraints on the global state space induced by strong synchronizations between atomic components.

Consider a set of atomic components $B = (B_1, \ldots, B_n)$, where $B_i = (L_i, P_i, \mathcal{T}_i)$, synchronized by a set of interactions γ. Intuitively, an interaction invariant of $\gamma(B)$ is a predicate in the disjunctive form $\Psi = \bigvee_{l \in L_\Psi} l$ where $L_\Psi \subseteq \bigcup_1^n L_i$ such that if any control location of L_Ψ is reached, then there is always at least a control location of L_Ψ which is reached by the execution of any interaction of γ. In other words, L_Ψ is a set of locations of atomic components such that for any location $l \in L_\Psi$, at least one of its successors by the execution of interactions in γ must belong to L_Ψ. For a system $S = \langle \gamma(B), Init \rangle$, $\Psi = \bigvee_{l \in L_\Psi}$ is an invariant of S if it is an invariant of $\gamma(B)$ and it is initially true, that is L_Ψ has at least an initial location of an atomic component in B. Interaction invariants are computed by solving Boolean Behavioral Constraints which characterize a set of successors of every location of atomic components or by Fixed-point computation.

Example 13 *Consider the component $\gamma(B_1, B_2)$ in Figure 2.5(a) where $\gamma = a_1a_2 + b_1b_2$, a set of locations $\{l_1, l_4\}$ corresponds to an interaction invariant $l_1 \vee l_4$ because l_4 is reached from l_1 by the interaction a_1a_2 and l_1 is reached from l_4 by the interaction b_1b_2.*

There is a similarity between the notion of interaction invariants and the notion of traps

in a type of Petri-net called 1-safe Petri-net, i.e. each place of an 1-safe Petri-net can not have more than one token. An important property of trap is that if a trap initially has a token, it will always have at least a token. The set of places in a Petri-net corresponds to a set of locations in atomic components, initial tokens correspond to initial locations and a place having a token means that its corresponding location is reached.

In this sub-section, we present methods for computing interaction invariants. We first define Forward Interaction Set of a location according to a set of interactions which is used in the methods.

We recall that we use ${}^\bullet\tau$ (respectively $\tau^\bullet$) to denote the source and destination locations of τ. Similarly, for a port p we have ${}^\bullet p = \{{}^\bullet\tau \mid \tau = (l, p, l')\}$ and $p^\bullet = \{\tau^\bullet \mid \tau = (l, p, l')\}$, for an interaction a we have ${}^\bullet a = \{{}^\bullet p \mid p \in a\}$ and $a^\bullet = \{p^\bullet \mid p \in a\}$.

Definition 16 (Forward Interaction Sets) *Given a component $\gamma(B_1, \ldots, B_n)$ where $B_i = (L_i, P_i, \mathcal{T}_i)$ are transition systems, we define for every location $l \in \bigcup_{i=1}^{n} L_i$ its forward interaction set as follows:*

$$\overrightarrow{l^\gamma} = \big\{\{\tau_i\}_{i\in I} \mid \forall i.[(\tau_i \in \mathcal{T}_i) \ \wedge \ (\exists i.{}^\bullet\tau_i = l) \ \wedge \ (\{port(\tau_i)\}_{i\in I} \in \gamma)]\big\}$$

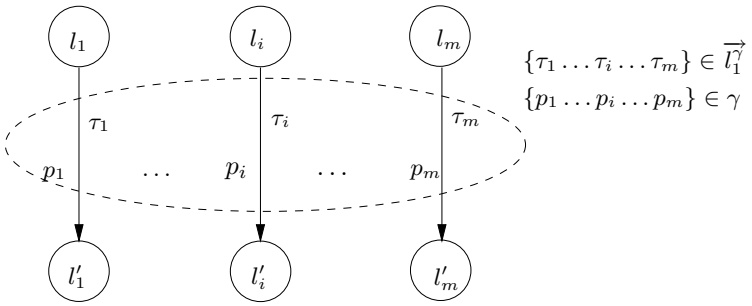


Figure 2.6: Forward interaction sets.

That is, $\overrightarrow{l^\gamma}$ consists of sets of transitions involved in some interaction of γ in which a transition τ_i issued from l can participate. A transition is involved in an interaction if the port labeling the transition participates in the interaction. For example, in Figure 2.6, the set $\{\tau_1 \ldots \tau_i \ldots \tau_m\}$ belongs to the Forward Interaction Sets $\overrightarrow{l_1^\gamma}$ (and also belongs to $\overrightarrow{l_2^\gamma}, \ldots \overrightarrow{l_i^\gamma}, \ldots \overrightarrow{l_m^\gamma}$).

Method based on Positive Mapping

We propose a method for computing interaction invariants from Boolean Behavioral Constraints (BBCs). We first give definition of BBCs and show that every solution of BBCs corresponds to an invariant. Then we provide a method for obtaining all the interaction invariants from BBCs by using an operation called Positive Mapping.

Boolean Behavioral Constraint (BBC) of a location l of an atomic component can be considered as a constraint enforced by a set of interactions γ from that location to a global location. It describes a set of successors of l according to γ, i.e. a set of locations of atomic components which are reached from l by involved interactions in γ. The Boolean Behavioral Constraints (BBCs) of a system is the conjunction of the BBC of all its locations.

We use $Bool[L]$ to denote the free algebra generated by the set of locations L. We provide the formal definition of Boolean Behavioral Constraints (BBCs) for a connector γ on a set of components as follows:

Definition 17 (Boolean Behavioral Constraints (BBCs)) *Let γ be a connector over a tuple of components $B = (B_1, \ldots, B_n)$ where $B_i = (L_i, P_i, \mathcal{T}_i)$ are transition systems. The Boolean Behavioral Constrains for component $\gamma(B)$ with a set of locations $L = \bigcup_{i=1}^{n} L_i$, are defined by a function $|.| : \gamma(B) \rightarrow Bool(L)$ such that:*

$$|\gamma(B)| = \bigwedge_{l \in L} \left(l \Rightarrow \bigwedge_{\{\tau_i\}_{i \in I} \in \overrightarrow{l^\gamma}} \left(\bigvee_{l' \in \{\tau_i^\bullet\}_{i \in I}} l' \right) \right)$$

If $\gamma = \emptyset$, then $|\gamma(B)| = true$, which means that no interactions between the components of B will be considered. $|\gamma(B)|$ can be written as the disjunction of monomials, i.e. $|\gamma(B)| = \bigvee_{i \in I} m_i$, which we call BBC-solutions.

Example 14 *In Figure 2.6, consider the interaction $a = p_1 \ldots p_i \ldots p_m \in \gamma$ between transitions $l_i \xrightarrow{p_i} l'_i$ for $i = 1, \ldots, m$, the corresponding BBCs is*

$$|a(B)| = \bigwedge_{i=1}^{m} (l_i \Rightarrow \bigvee_{j=1}^{m} l'_j)$$

In Figure 2.5(a) the BBCs for the the set of interactions $\gamma = a_1a_2 + b_1b_2$ is $|\gamma(B)| = (l_1 \Rightarrow l_2 \vee l_4) \wedge (l_3 \Rightarrow l_2 \vee l_4) \wedge (l_2 \Rightarrow l_1 \vee l_3) \wedge (l_4 \Rightarrow l_1 \vee l_3)$.

The following theorem provides means for computing interaction invariants from BBCs.

Theorem 1 *Let $B = (B_1, \cdots, B_n)$ be a set of components with $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^{n} L_i$, γ be a connector over B, and $v : L \rightarrow \{true, false\}$ be a boolean valuation different from false. If v is a solution of $|\gamma(B)|$, i.e. $|\gamma(B)|(v) = true$, then $\bigvee_{v(l)=true} l$ is an invariant of $\gamma(B)$.*

Proof According to Definition 17, the constraints are the conjunction of all the implications for interactions of γ. Consider a valuation v such that $|\gamma(B)|(v) = true$. In order to prove that $\bigvee_{v(l)=true} l$ is an invariant, assume that for some global state $l = (l_1, \cdots, l_n)$, there exists l_i such that $v(l_i) = true$. If from l_i there is an interaction a such that $l_i \in {}^\bullet a$, then there exists $l'_j \in a^\bullet$, such that $v(l'_j) = true$ by Definition 17. So any successor state of l by an interaction a satisfies the invariant.

That is, the disjunction of positive valuations in any solution of BBCs is an interaction invariant of $\gamma(B)$. Concretely, if $v = \bigwedge_{i\in I} l_i \wedge \bigwedge_{j\in K} \bar{l}_j$ is a solution of BBCs, then the disjunction of the positive valuations $\bigvee_{i\in I} l_i$ is an interaction invariant of $\gamma(B)$.

A naive way to compute interaction invariants is to get all the solutions of BBCs, but here there is a risk of explosion if the number of solutions is huge. We therefore provide below a method for computing symbolically interaction invariants based on an operation called Positive Mapping which allows removing all the negative valuations of a set of variables in a boolean formula expressed as the disjunction of monomials.

Definition 18 (Positive Mapping) *Given two sets of variables X, Y such that $X \subseteq Y$, and a boolean formula f on Y expressed as the disjunction of monomials. We define an operation, called Positive Mapping, that deletes all the negative variables that do not belong to X, denoted by $f^{p(X)}$, as follows:*

$$(\textstyle\bigwedge_{l_i\in Y} l_i \wedge \bigwedge_{l_j\in X} \bar{l}_j \bigwedge_{l_k\in Y-X} \bar{l}_k)^{p(X)} = \bigwedge_{l_i\in Y} l_i \wedge \bigwedge_{l_j\in X} \bar{l}_j,$$
$$(f_1 \vee f_2)^{p(X)} = f_1^{p(X)} \vee f_2^{p(X)}$$

Here we use $\bar{l}$ for $\neg l$. When X is empty, the positive mapping will remove all the negative variables in f, which is denoted by f^p. If all the variables are negative in f, we have $f^p = false$.

Example 15 *Given a boolean function $f = (x\wedge y\wedge \bar{z})\vee(x\wedge \bar{y}\wedge z)$ and a subset of variables $X = \{x, y\}$, we have Positive Mapping $f^{p(X)} = (x\wedge y)\vee(x\wedge \bar{y}\wedge z)$ and $f^p = (x\wedge y)\vee(x\wedge z)$.*

The global interaction invariant is obtained by conjunction of all interaction invariants. The following theorem allows computing symbolically the global interaction invariant of $\gamma(B)$ from its Boolean Behavioral Constraints $|\gamma(B)|$. But first let us define the *dual operation* which is used in the computation of interaction invariants.

Definition 19 (Dual Operation) *Given a boolean formula $f(X)$ on a set of variables $X = \{x_1, \ldots, x_n\}$. We define the dual operation on $f(X)$, denoted by $\widetilde{f(X)}$, as follows: $\widetilde{f(X)} = \overline{f(\bar{X})}$ where $f(\bar{X})$ is a boolean formula obtained from $f(X)$ by replacing, for each variable $x_i \in X$, its positive form x_i (respectively its negative form $\bar{x}_i$) by its negative form $\bar{x}_i$ (respectively its positive form x_i).*

Example 16 *For the boolean formula $f = (x \wedge \bar{y}) \vee (y \wedge \bar{z}) \vee (x \wedge z)$, we have its dual $\widetilde{f} = (x \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (x \vee z)$.*

Theorem 2 *For any connector γ applied to a tuple of components $B = (B_1, \cdots, B_n)$, the global interaction invariant of $\gamma(B)$ can be obtained as the dual of the positive mapping of $|\gamma(B)|$, denoted by $\widetilde{|\gamma(B)|^p}$.*

Proof (Sketch). $|\gamma(B)|$ can be written as the disjunction of monomials, that is $|\gamma(B)| = \bigvee_{i\in I} m_i$, where m_i is of the form $m_i = \bigwedge_{j\in I} l_{i_j} \wedge \bigwedge_{k\in K} \overline{l_{i_k}}$. We have $|\gamma(B)|^p = \bigvee_{i\in I} m_i^p = \bigvee_{i\in I}(\bigwedge_{j\in I} l_{i_j})$, hence $\widetilde{|\gamma(B)|^p} = \bigwedge_{i\in I}(\bigvee_{j\in I} l_{i_j})$ is the global interaction invariant of $\gamma(B)$ according to Theorem 1.

Here we obtain by $\widetilde{|\gamma(B)|^p}$ all the possible interaction invariants of $\gamma(B)$. For a system $S = \langle \gamma(B), Init \rangle$ where $Init$ is the initial state (the set of initial locations of components in B), the interaction invariants of S are obtained from interaction invariants of $\gamma(B)$ by selecting all the invariants that have at least an initial location. The global interaction invariants of S can be obtained as $(|\gamma(B)| \widetilde{\wedge \bigvee_{l \in Init} l})^p$.

Example 17 *We use the example in Figure 2.5 to illustrate the computation of invariants, where $B = (B_1, B_2)$ and $\gamma = a_1a_2 + b_1b_2$. The BBCs for $\gamma(B)$, according to Example 14 is:*

$$\begin{aligned}|\gamma(B)| &= (l_1 \Rightarrow l_2 \vee l_4) \wedge (l_2 \Rightarrow l_1 \vee l_3) \wedge (l_3 \Rightarrow l_2 \vee l_4) \wedge (l_4 \Rightarrow l_1 \vee l_3) \\ &= (\bar{l_1} \wedge \bar{l_2} \wedge \bar{l_3} \wedge \bar{l_4}) \vee (l_1 \wedge l_2) \vee (l_2 \wedge l_3) \vee (l_1 \wedge l_4) \vee (l_3 \wedge l_4)\end{aligned}$$

By applying the Positive Mapping operation, we have:

$$|\gamma(B)|^p = (l_1 \wedge l_2) \vee (l_2 \wedge l_3) \vee (l_1 \wedge l_4) \vee (l_3 \wedge l_4)$$

Thus the global interaction invariant is:

$$\widetilde{|\gamma(B)|^p} = (l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3)$$

From the global interaction invariant obtained in Example 17, if we take into account the initial state $Init = \{l_1, l_3\}$, we have the same global interaction invariant $\Psi = \widetilde{|\gamma(B)|^p}$ for the system $S = \langle \gamma(B), Init \rangle$ (because all the interaction invariants of $\gamma(B)$ have at least an initial location). The global reachable states of S are approximated by the global interaction invariant $\Psi = (l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3) = (l_1 \wedge l_3) \vee (l_2 \wedge l_4)$. In this case, the obtained global interaction invariant represents exactly the set of reachable states.

Method based on Fixed-point Computation

Fixed-point-based methods are widely used in Model-Checking for computing reachable states. Starting from the global initial states, the global successor states are iteratively computed until no more new global state is generated, i.e, we have reached fixed-points in computing reachable states.

The interaction invariants can be iteratively computed by using fixed-point computation technique. In our fixed-point method, we start from a control location of an atomic component, and then iteratively compute a set of successors of that location by global image vector obtained from the set of interactions. The iteration stops when no more successor is generated, that is we have reached fixed-points. A fixed-point corresponds to a set of locations such that if a location of the set is reached, then at any time, at least one location of the set is reached.

The main difference of our method from the fixed-point method used in Model-Checking is that we do not compute iteratively the global successors of a global state which can be exponential in the size of system. Our fixed-point method computes iteratively successors of every location of atomic components, therefore it does not suffer from the state space explosion problem.

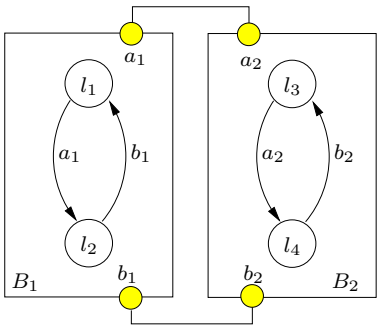


Figure 2.7: An example of two components strongly synchronized

Definition 20 (Image Vector) *Let γ be a connector over a set of components $B = (B_1, \ldots, B_n)$ where $B_i = (L_i, P_i, \mathcal{T}_i)$. The image vector for a set of locations $L = \bigcup_{i=1}^{n} L_i$ according to γ is defined as follows:*

$$\mathbb{V}_\gamma = \{\mathbb{V}_\gamma(l) = l \wedge f_l^\gamma(L) \mid l \in L\} \quad \textit{where} \quad f_l^\gamma(L) = \bigwedge_{\{\tau_i\}_{i\in I} \in \overrightarrow{l^\gamma}} \left(\bigvee_{l' \in \{\tau_i^\bullet\}_{i\in I}} l' \right)$$

The image $\mathbb{V}_\gamma(l)$ of a location l according to a connector γ defines the set of locations (including l) that can be reached from l by involved interactions in γ.

The formula $f_l^\gamma(L)$ is actually the right side of the implication in the BBC of the location l according to definition 17. Let $l \Rightarrow f_l^\gamma(L)$ be an implication for a location $l \in L$, we have $l = l \wedge f_l^\gamma(L)$.

Example 18 *For the example illustrated in Figure 2.7 with $\gamma = a_1a_2 + b_1b_2$, the Boolean Behavioral Constraints of locations l_1, l_2, l_3, l_4 are respectively:*

$$l_1 \Rightarrow l_2 \vee l_4 \qquad l_2 \Rightarrow l_1 \vee l_3 \qquad l_3 \Rightarrow l_2 \vee l_4 \qquad l_4 \Rightarrow l_1 \vee l_3$$

The corresponding image vector is $\mathbb{V}_\gamma = \{\mathbb{V}_\gamma(l_1), \mathbb{V}_\gamma(l_2), \mathbb{V}_\gamma(l_3), \mathbb{V}_\gamma(l_4)\}$ where:

$$\mathbb{V}_\gamma(l_1) = l_1 \wedge (l_2 \vee l_4) \qquad \mathbb{V}_\gamma(l_3) = l_3 \wedge (l_2 \vee l_4)$$
$$\mathbb{V}_\gamma(l_2) = l_2 \wedge (l_1 \vee l_3) \qquad \mathbb{V}_\gamma(l_4) = l_4 \wedge (l_1 \vee l_3)$$

Definition 21 (Image Function) *Let $\mathbb{V}$ be an image vector on a set of variables L, ϕ be a predicate on L in the disjunctive form of monomials $\phi = \bigvee_i \varphi_i$, the image function of ϕ with the image vector $\mathbb{V}$ is defined by:*

$$Image(\mathbb{V}, \phi) = \bigvee_i \Big(\bigwedge_{l \in L_{\varphi_i}} \mathbb{V}(l) \Big)$$

where L_{φ_i} is the set of variables in φ_i.

That is, the *Image* function replaces every variable l of each monomial φ_i by the corresponding image $\mathbb{V}(l)$.

Definition 22 (Fixed-points) *Let $\mathbb{V}$ be an image vector on a set of variables L, ϕ be a predicate on L, we define an iteration process that allows computing fixed-points starting from ϕ according to the image vector $\mathbb{V}$ as follows:*

$$\phi^0 = \phi$$

$$\phi^{k+1} = Image(\mathbb{V}, \phi^k)$$

When $\phi^{k+1} = \phi^k$, the iteration terminates and ϕ^k is the fixed-points of the computation, denoted by $\mathbb{F}(\mathbb{V}, \phi)$.

Example 19 *For the example presented in Figure 2.7 with the image vector obtained in Example 18, and for a predicate $\phi = l_1$, the iterations are as follows:*

$$\begin{array}{l}\phi^0 = \phi = l_1\\ \phi^1 = Image(\mathbb{V}_\gamma, \phi^0) = (l_1 \wedge l_2) \vee (l_1 \wedge l_4)\\ \phi^2 = Image(\mathbb{V}_\gamma, \phi^1) = (l_1 \wedge l_2) \vee (l_1 \wedge l_4)\end{array}$$

The iteration stops because $\phi^2 = \phi^1$ and we obtain the fixed-points $\mathbb{F}(\mathbb{V}_\gamma, l_1) = (l_1 \wedge l_2) \vee (l_1 \wedge l_4)$. Similarly, from locations l_2, l_3, l_4 we obtain respectively the following fixed-points:

$$\begin{array}{l}\mathbb{F}(\mathbb{V}_\gamma, l_2) = (l_1 \wedge l_2) \vee (l_2 \wedge l_3)\\ \mathbb{F}(\mathbb{V}_\gamma, l_3) = (l_2 \wedge l_3) \vee (l_3 \wedge l_4)\\ \mathbb{F}(\mathbb{V}_\gamma, l_4) = (l_1 \wedge l_4) \vee (l_3 \wedge l_4)\end{array}$$

The following theorem provides means for computing interaction invariants from fixed-points.

Theorem 3 *Let $B = (B_1, \ldots, B_n)$ be a set of components with $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^{n} L_i$, γ be a connector over B with the image vector $\mathbb{V}_\gamma$. For any location $l_i \in L$, if $m = \bigwedge l_j$ is a solution (a fixed-point) of $\mathbb{F}(\mathbb{V}_\gamma, l_i)$, then the dual of m, that is $\widetilde{m} = \bigvee l_j$, is an invariant of $\gamma(B)$.*

Proof Let L_m be the set of location variables in the solution m. We assume that for some global state $l = (l_1, \ldots, l_n)$, these exists l_k such that $l_k \in L_m$. If from l_k there is an interaction a such that $l_k \in {}^\bullet a$, then there exists $l'_k \in a^\bullet$, such that $l'_k \in L_m$ by Definition 20, 21 and 22. So any successor state of l by an interaction a satisfies $\widetilde{m} = \bigvee_{l_j \in L_m} l_j$.

Example 20 *According to the fixed-points obtained in Example 19, we have the following interaction invariants:*

$$\begin{array}{ll}\Psi_{l_1} = (l_1 \vee l_2) \wedge (l_1 \vee l_4) & \Psi_{l_2} = (l_1 \vee l_2) \wedge (l_2 \vee l_3)\\ \Psi_{l_3} = (l_2 \vee l_3) \wedge (l_3 \vee l_4) & \Psi_{l_4} = (l_1 \vee l_4) \wedge (l_3 \vee l_4)\end{array}$$

The global interaction invariant is obtained by conjunction of all interaction invariants. The following theorem allows computing symbolically the global interaction invariant of $\gamma(B)$ by fixed-points starting from the disjunction of locations of atomic components in B.

Theorem 4 *Let $B = (B_1, \ldots, B_n)$ be a set of components with $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^{n} L_i$, γ be a connector over B with the image vector $\mathbb{V}_\gamma$, the global interaction invariant of $\gamma(B)$ can be obtained as the dual of fixed-points $\mathbb{F}(\mathbb{V}_\gamma, \bigvee_{l\in L} l)$, denoted by $\widetilde{\mathbb{F}}(\mathbb{V}_\gamma, \bigvee_{l\in L} l)$.*

Proof We have $\mathbb{F}(\mathbb{V}_\gamma, \bigvee_{l\in L} l) = \bigvee_{l\in L} \mathbb{F}(\mathbb{V}_\gamma, l)$, hence $\widetilde{\mathbb{F}}(\mathbb{V}_\gamma, \bigvee_{l\in L} l) = \bigwedge_{l\in L} \widetilde{\mathbb{F}}(\mathbb{V}_\gamma, l)$ is the global interaction invariant according to Theorem 3.

Example 21 *For the example presented in Figure 2.7 with the image vector obtained in Example 18, and for a predicate $\phi = l_1 \vee l_2 \vee l_3 \vee l_4$, the iterations are as follows:*

$$\begin{aligned}
&\phi^0 = \phi = l_1 \vee l_2 \vee l_3 \vee l_4\\
&\phi^1 = Image(\mathbb{V}_\gamma, \phi^0) = (l_1 \wedge l_2) \vee (l_1 \wedge l_4) \vee (l_2 \wedge l_3) \vee (l_3 \wedge l_4)\\
&\phi^2 = Image(\mathbb{V}_\gamma, \phi^1) = (l_1 \wedge l_2) \vee (l_1 \wedge l_4) \vee (l_2 \wedge l_3) \vee (l_3 \wedge l_4)
\end{aligned}$$

The iteration stops because $\phi^2 = \phi^1$ and we obtain the fixed-points $\mathbb{F}(\mathbb{V}_\gamma, l_1) = \phi^1$ from which we obtain the global interaction invariant:

$$\Psi = \widetilde{\mathbb{F}}(\mathbb{V}_\gamma, l_1 \vee l_2 \vee l_3 \vee l_4) = (l_1 \vee l_2) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3) \wedge (l_3 \vee l_4)$$

We call $\mathbb{F}(\mathbb{V}_\gamma, \bigvee_{l\in L} l)$ fixed-points of $\gamma(B)$. For a system $S = \langle \gamma(B), Init \rangle$ where γ is a connector on a set of components B and $Init$ is the initial state, since every interaction invariant must contain at least an initial location of a component in B, we start the iteration for fixed-point computation from the initial locations of atomic components. That is the global interaction invariant of S is obtained as $\widetilde{\mathbb{F}}(\mathbb{V}_\gamma, \bigvee_{l\in Init} l)$. This guarantees the existence of the initial locations in all the obtained invariants.

2.2 Abstraction

Abstraction techniques have been widely developed and used in verification in order to alleviate the state space explosion problem, specially for the verification of infinite systems. For finite systems, abstraction is also necessary and important to the success of the verification. The goal of abstraction is to build, for each concrete system, an abstract system which preserves properties to verify and is less expensive to analyze or to check by existing verification tools such as Model-Checkers.

We have provided two methods for computing interaction invariants of systems without data. For systems with data, an abstraction technique is needed to abstract away the data before applying the methods. The process for computing interaction invariants of systems with data is presented in figure 2.8. It consists of three steps:

- First we need to make an abstraction of the system:

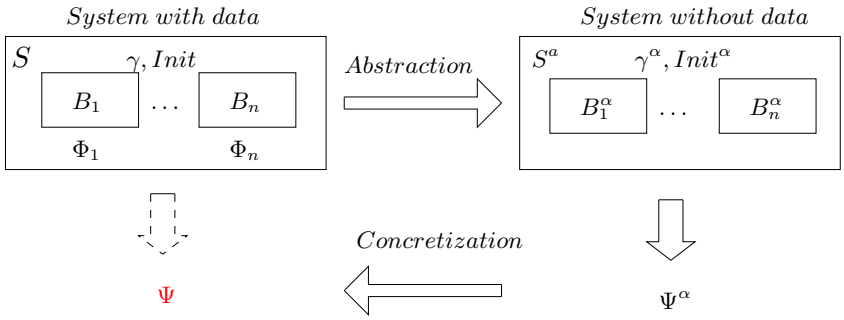


Figure 2.8: Interaction invariants computation for systems with data

  - for each atomic component B_i with data of the system S, an abstraction is made to obtain a corresponding abstract atomic component B_i^α without data.
  - abstract connector γ^α is obtained from γ by generating for each interaction in γ a corresponding abstract interaction.
  - abstract initial condition $Init^\alpha$ is made from $Init$.

- Then the methods for computing interaction invariants of systems without data is applied for the abstract systems S^α. We obtain a set of abstract interaction invariants Ψ^α of S^α.

- Finally, interaction invariants Ψ of the concrete system S are obtained by concretizing the set of the abstract interaction invariants Ψ^α.

The abstraction technique we use is based on the method proposed by Bensalem et al. in [BLO98a]. The basic idea of the method is to use a splitting algorithm to refine an abstract structure in order to preserve properties in the abstract-concrete direction, that is any property satisfied by abstract system will be satisfied by the concrete system. The advantage of this method is that it produces an abstract system which has the same structure as the concrete one. This allows for further application of abstraction and gives a clear correspondence between abstract and concrete transitions which is useful for debugging the concrete system. This method has been implemented in the InVeSt tool [BLO98b].

Given a concrete system S, the abstraction method allows to compute an abstract system S^α that S simulates S^α, that is every computation of S can be mapped to a computation of S^α. Consider a system $\mathcal{S} = \langle \gamma(B_1, \ldots, B_n), Init \rangle$ and a set of component invariants $\Phi_1 \ldots \Phi_n$ associated with the atomic components. We show below, for each component B_i and its associated invariant Φ_i, how to define a finite state abstraction α_i and to compute an abstract transition system $B_i^{\alpha_i}$.

Definition 23 (Abstraction Function) *Let Φ be an invariant of a system $\langle B, Init \rangle$ written in disjunctive form $\Phi = \bigvee_{l \in L} l \wedge (\bigvee_{m \in M_l} \varphi_{lm})$ such that atomic predicates of the form $l \wedge \varphi_{lm}$ are disjoint. Given Φ, an abstraction function α is an injective function*

associating with each atomic predicate $l \wedge \varphi_{lm}$ a symbol $\phi = \alpha(l \wedge \varphi_{lm})$ called abstract state. We denote by Φ^α the set of the abstract states.

Example 22 *Consider the component Controller in the Temperature Control System (Figure 2.2). The component invariant predicate at each location can be written in the disjunctive form according to the post predicate transformers of its incoming transitions as follows:*

- $\Phi_{l_5} = l_5 \wedge (\theta = 100 \vee 101 \leq \theta \leq 1000)$
- $\Phi_{l_6} = l_6 \wedge (\theta = 1000 \vee 100 \leq \theta \leq 998)$

Therefore, we have four abstract states:

$$\Phi_{51} = l_5 \wedge \theta = 100 \qquad \Phi_{61} = l_6 \wedge \theta = 1000$$
$$\Phi_{52} = l_5 \wedge 101 \leq \theta \leq 1000 \qquad \Phi_{62} = l_6 \wedge 100 \leq \theta \leq 998$$

The abstraction function maps concrete states to abstract states taken from a finite set, hence we obtain a finite state system which can be analyzed algorithmically. This allows us to compute an over approximation of the set of reachable states which is sufficient for the verification of invariants. Using the abstraction function, an abstract system is defined as follows:

Definition 24 (Abstract System) *Given a system $\mathcal{S} = \langle B, Init \rangle$, an invariant Φ and an associated abstraction function α, we define the abstract system $\mathcal{S}^\alpha = \langle B^\alpha, Init^\alpha \rangle$ where*

- *$B^\alpha = (\Phi^\alpha, P, \rightsquigarrow)$ is a transition system with $\rightsquigarrow$ such that for any pair of abstract states $\phi = \alpha(l \wedge \varphi)$ and $\phi' = \alpha(l' \wedge \varphi')$ we have $\phi \overset{p}{\rightsquigarrow} \phi'$ iff $\exists \tau = (l, p, l') \in \mathcal{T}$ and $post_\tau(\varphi) \wedge \varphi' \neq false$ (or equivalently $\varphi \wedge pre_\tau(\varphi') \neq false$),*
- *$Init^\alpha = \bigvee_{\phi \in \Phi_0^\alpha} \phi$ where $\Phi_0^\alpha = \{\phi \in \Phi^\alpha \mid \alpha^{-1}(\phi) \wedge Init \neq false\}$ is the set of the initial abstract states.*

The method proceeds by elimination, starting from the universal relation on abstract states. We eliminate pairs of abstract states in a conservative way. To check whether $\phi \overset{p}{\rightsquigarrow} \phi'$, where $\phi = \alpha(l \wedge \varphi)$ and $\phi' = \alpha(l' \wedge \varphi')$, can be eliminated, we check that for all concrete transitions $\tau = (l, p, l')$ we have $post_\tau(\varphi) \wedge \varphi' = false$ or equivalently $\varphi \wedge pre_\tau(\varphi') = false$.

Example 23 *The table below provides the abstract states constructed from the component invariants Φ_1, Φ_2, Φ_3 of respectively Rod1, Rod2, Controller given in example 10.*

$\phi_{11} = l_1 \wedge t_1 = 0$	$\phi_{51} = l_5 \wedge \theta = 100$	$\phi_{31} = l_3 \wedge t_2 = 0$
$\phi_{12} = l_1 \wedge t_1 \geq 1$	$\phi_{52} = l_5 \wedge 101 \leq \theta \leq 1000$	$\phi_{32} = l_3 \wedge t_2 \geq 1$
$\phi_{21} = l_2 \wedge t_1 \geq 3600$	$\phi_{61} = l_6 \wedge \theta = 1000$	$\phi_{41} = l_4 \wedge t_2 \geq 3600$
	$\phi_{62} = l_6 \wedge 100 \leq \theta \leq 998$	

Figure 2.9 presents the computed abstraction of the Temperature Control System with respect to the considered invariants.

We take several transitions of Controller component to illustrate the construction of the abstract transitions, for example:

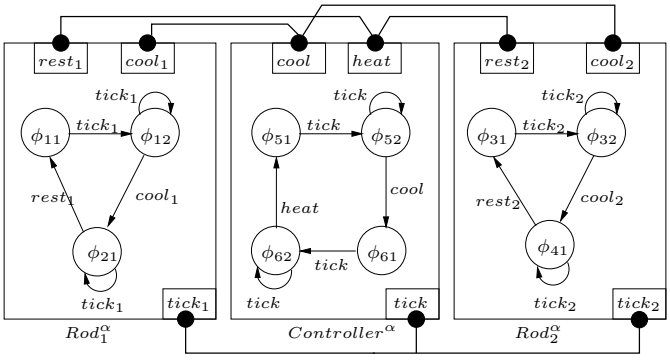


Figure 2.9: Abstraction of the Temperature Control System

- *transition* $\tau^\alpha = (\phi_{52}, cool, \phi_{61})$ *is established because there exists a concrete transition* $\tau = (l_5, cool, l_6)$ *and* $post_\tau(\varphi_{52}) \wedge \varphi_{61} \neq false$ *where* $post_\tau(\varphi_{52}) = (\theta = 1000)$ *and* $\varphi_{61} = (\theta = 1000)$.
- *transition* $\tau^\alpha = (\phi_{52}, cool, \phi_{62})$ *is eliminated since* $post_\tau(\varphi_{52}) \wedge \varphi_{62} = false$ *where* $post_\tau(\varphi_{52}) = (\theta = 1000)$ *and* $\varphi_{62} = (100 \leq \theta \leq 998)$.

Consider the concrete initial condition $Init = l_5 \wedge (\theta = 100) \wedge l_1 \wedge (t_1 = 3600) \wedge l_3 \wedge (t_2 = 3600)$*, we have the abstract initial condition* $Init^\alpha = \phi_{51} \wedge \phi_{12} \wedge \phi_{32}$.

By combining well-known results about abstractions, we can compute interaction invariants of $\langle \gamma(B_1, ..., B_n), Init \rangle$ from interaction invariants of $\langle \gamma(B_1^\alpha, \ldots, B_n^\alpha), Init^\alpha \rangle$.

The following proposition says that $\gamma(B_1^{\alpha_1}, \ldots, B_n^{\alpha_n})$ is an abstraction of $B = \gamma(B_1, ..., B_n)$

Proposition 6 *If* $B_i^{\alpha_i}$ *is an abstraction of* B_i *with respect to an invariant* Φ_i *and its abstraction function* α_i *for* $i = 1, ..., n$ *, then* $B^\alpha = \gamma(B_1^{\alpha_1}, \ldots, B_n^{\alpha_n})$ *is an abstraction of* $B = \gamma(B_1, ..., B_n)$ *with respect to* $\bigwedge_{i=1}^n \Phi_i$ *and an abstraction function* α *obtained as the composition of the* α_i.

The following proposition says that invariants of the abstract system are also invariants of the concrete system.

Proposition 7 *If* B^α *is an abstraction of* B *with respect to* Φ *and its abstraction function* α*, then* B^α *simulates* B*. Moreover, if* Φ^α *is an invariant of* $\langle B^\alpha, Init^\alpha \rangle$ *then* $\alpha^{-1}(\Phi^\alpha)$ *is an invariant of* $\langle B, Init \rangle$.

Proof We show that the relation $(l, \mathbf{x})R\phi$ is a simulation if $\alpha^{-1}(\phi) = l \wedge \varphi$ and $\varphi(\mathbf{x})$ for the valuation $\mathbf{x}$. If $(l, \mathbf{x}) \xrightarrow{p} (l', \mathbf{x}')$ is a transition of B and $(l, \mathbf{x})R\phi$ for some abstract state ϕ, then we show that there exists $\phi' = \alpha(l' \wedge \varphi')$ such that $\phi \overset{p}{\rightsquigarrow} \phi'$. As Φ is an invariant of B, if $(l', \mathbf{x}')$ is reachable then $\exists \varphi'\ l' \wedge \varphi' \Rightarrow \Phi$ such that $\varphi'(\mathbf{x}')$ and $\phi' = \alpha(l' \wedge \varphi')$. Moreover, as $\varphi(\mathbf{x}) \wedge \varphi'(\mathbf{x}')$, we have $\varphi(\mathbf{x}) \wedge pre_\tau(\varphi)(\mathbf{x}) \neq false$ for $\tau = (l, p, l')$ and therefore $\phi \overset{p}{\rightsquigarrow} \phi'$.

Thus, it is possible to compute from interaction invariants of the abstract system, interaction invariants for the concrete system $\langle \gamma(B_1, ..., B_n), Init \rangle$.

We can show by application of the following proposition that the iteration process gives progressively stronger invariants, in particular that for stronger component invariants we get stronger interaction invariants.

Proposition 8 *Let $\langle B, Init \rangle$ be a system and Φ, Φ' two non empty invariants such that $\Phi \Rightarrow \Phi'$. If α and α' are the abstraction functions corresponding to Φ and Φ' respectively, then B^{α} simulates $B^{\alpha'}$.*

Proof For two successive component invariants Φ_i and Φ'_i for B_i, we have $\Phi_i \Rightarrow \Phi'_i$. From proposition 8 we deduce that $B_i^{\alpha_i}$ simulates $B_i^{\alpha'_i}$ where α_i and α'_i are the abstraction functions corresponding to Φ_i and Φ'_i. As the simulation relation is preserved by parallel composition, we have $\gamma(B_1^{\alpha_1}, ..., B_n^{\alpha_n})$ simulates $\gamma(B_1^{\alpha'_1}, ..., B_n^{\alpha'_n})$. We can show that for each positive valuation set L' of a solution of $\gamma(B_1^{\alpha'_1}, ..., B_n^{\alpha'_n})$ there exists a positive valuation set L of a solution of $\gamma(B_1^{\alpha_1}, ..., B_n^{\alpha_n})$ such that $L \subseteq L'$. From this we infer that for each interaction invariant of $\gamma(B'_1, ..., B'_n)$ there exists a stronger interaction invariant of $\gamma(B_1, ..., B_n)$.

Below we provide an example on the Temperature Control System for showing the computation of abstract interaction invariants and then the concretization to obtain interaction invariants of the concrete system.

Example 24 *For the abstraction of the Temperature Control System given in figure 2.9, we have the following Boolean Behavioral Constraints:*

$$
\begin{array}{lll}
\phi_{11} \Rightarrow (\phi_{12} \vee \phi_{32} \vee \phi_{52}) & \wedge\ (\phi_{21} \vee \phi_{32} \vee \phi_{62}) & \phi_{61} \Rightarrow (\phi_{12} \vee \phi_{32} \vee \phi_{62}) \\
\quad \wedge\ (\phi_{12} \vee \phi_{32} \vee \phi_{62}) & \phi_{32} \Rightarrow (\phi_{61} \vee \phi_{41}) & \quad \wedge\ (\phi_{21} \vee \phi_{41} \vee \phi_{62}) \\
\quad \wedge\ (\phi_{12} \vee \phi_{41} \vee \phi_{52}) & \phi_{41} \Rightarrow (\phi_{51} \vee \phi_{31}) & \quad \wedge\ (\phi_{21} \vee \phi_{32} \vee \phi_{62}) \\
\quad \wedge\ (\phi_{12} \vee \phi_{41} \vee \phi_{62}) & \phi_{51} \Rightarrow (\phi_{12} \vee \phi_{32} \vee \phi_{52}) & \quad \wedge\ (\phi_{12} \vee \phi_{41} \vee \phi_{62}) \\
\phi_{12} \Rightarrow (\phi_{61} \vee \phi_{21}) & \quad \wedge\ (\phi_{21} \vee \phi_{41} \vee \phi_{52}) & \phi_{62} \Rightarrow (\phi_{51} \vee \phi_{11}) \\
\phi_{21} \Rightarrow (\phi_{51} \vee \phi_{11}) & \quad \wedge\ (\phi_{21} \vee \phi_{32} \vee \phi_{52}) & \quad \wedge\ (\phi_{51} \vee \phi_{31}) \\
\phi_{31} \Rightarrow (\phi_{12} \vee \phi_{32} \vee \phi_{52}) & \quad \wedge\ (\phi_{12} \vee \phi_{41} \vee \phi_{52}) & \\
\quad \wedge\ (\phi_{12} \vee \phi_{32} \vee \phi_{62}) & \phi_{52} \Rightarrow (\phi_{61} \vee \phi_{21}) & \\
\quad \wedge\ (\phi_{21} \vee \phi_{32} \vee \phi_{52}) & \quad \wedge\ (\phi_{61} \vee \phi_{41}) &
\end{array}
$$

According to theorem 2, by applying the positive mapping and the dual operation, we obtain the following global abstract interaction invariant of the abstract system:

$$
\begin{array}{lll}
\Psi^a = (\phi_{11} \vee \phi_{31} \vee \phi_{32} \vee \phi_{52} \vee \phi_{61} \vee \phi_{62}) & \wedge & (\phi_{21} \vee \phi_{41} \vee \phi_{51} \vee \phi_{52}) \\
\quad \wedge\ (\phi_{11} \vee \phi_{12} \vee \phi_{31} \vee \phi_{52} \vee \phi_{61} \vee \phi_{62}) & \wedge & (\phi_{12} \vee \phi_{21} \vee \phi_{51}) \\
\quad \wedge\ (\phi_{11} \vee \phi_{12} \vee \phi_{31} \vee \phi_{32} \vee \phi_{61} \vee \phi_{62}) & \wedge & (\phi_{32} \vee \phi_{41} \vee \phi_{51})
\end{array}
$$

The concretization of the global abstract interaction invariant provides the global interaction invariant of the concrete system:

$$\begin{aligned}\Psi = &((l_2 \wedge t_1 \geq 3600) \vee (l_4 \wedge t_2 \geq 3600) \vee (l_5 \wedge 100 \leq \theta \leq 1000))\\ &\bigwedge ((l_1 \wedge t_1 \geq 0) \vee (l_2 \wedge t_1 \geq 3600) \vee (l_3 \wedge t_2 \geq 0) \vee (l_4 \wedge t_2 \geq 3600))\\ &\bigwedge ((l_3 \wedge t_2 \geq 1) \vee (l_4) \vee (l_5 \wedge \theta = 100))\\ &\bigwedge ((l_1 \wedge t_1 \geq 0) \vee (l_3 \wedge t_2 \geq 0) \vee (l_6 \wedge \theta = 1000) \vee (l_6 \vee 100 \leq \theta \leq 998))\\ &\bigwedge ((l_1 \wedge t_1 \geq 1) \vee (l_2) \vee (l_5 \wedge \theta = 100))\end{aligned}$$

2.3 Checking Safety Properties

We have presented the methods for computing component invariants and interaction invariants. We have also presented an abstraction technique to compute interaction invariants of systems with data. In this section, we will show the procedure for the verification of safety properties by using these invariants.

We give a sketch of a semi-algorithm in Algorithm 1 allowing to prove invariance of a safety property Φ by iterative application of the verification rule (2.3). The semi-algorithm takes a system $\langle \gamma(B_1, \ldots, B_n), Init \rangle$ and a predicate Φ. It iteratively computes invariants of the form $\mathcal{X} = \Psi \wedge (\bigwedge_{i=1}^{n} \Phi_i)$ where Ψ is an interaction invariant and Φ_i an invariant of component B_i. It consists of the following steps:

```
1  function verify(S = ⟨γ(B_1, ..., B_n), Init⟩, Φ)
2  begin
3      for each component B_i do
4          Φ_i = true;
5      end
6      while true do
7          for each component B_i do
8              compute component invariants Φ'_i;
9              Φ_i = Φ_i ∧ Φ'_i;
10             compute abstraction B_i^a = α(B_i, Φ_i);
11         end
12         compute abstract system ⟨γ(B_1^a, ..., B_n^a), Init^a⟩;
13         compute interaction invariants Ψ^a of ⟨γ(B_1^a, ..., B_n^a), Init^a⟩;
14         concretize abstract invariants Ψ = α^{-1}(Ψ^a);
15         X = Ψ ∧ (⋀_{i=1}^{n} Φ_i);
16         if ¬Φ ∧ X is unsatisfiable then
17             return Φ is an invariant;
18         end
19         else if receive stop or timeout then
20             return inconclusive;
21         end
22     end
23 end
```

Algorithm 1: Checking Invariance-Property Φ

- Step 0: every component invariant Φ_i is initially true (line 4).
- Step 1: for each component B_i, a stronger invariant Φ'_i is computed (line 8) and Φ_i is updated by conjoining with Φ'_i (line 9).

- Step 2: Φ_i is then used, together with B_i, to compute an abstraction of B_i by the abstract function α (line 10).
- Step 3: the abstract system $\langle \gamma(B_1^a, \ldots, B_n^a), Init^\alpha \rangle$ is computed from the set of abstract components, the set of interactions γ and the initial condition $Init$ (line 12).
- Step 4: interaction invariants are computed from the abstract system (line 13) and their concretization provides concrete interaction invariants of the concrete system (line 14).
- Step 5: the global invariant $\mathcal{X} = \Psi \wedge (\bigwedge_{i=1}^{n} \Phi_i)$ is used to verify the invariance of Φ by checking whether $\neg\Phi \wedge \mathcal{X}$ is unsatisfiable. If it is, the verification terminates and results the invariance of Φ (line 17). If $\mathcal{X}$ is not strong enough for proving that Φ is an invariant then either a new iteration with stronger Φ_i is started by returning to Step 1 or we can stop. In this case, we cannot conclude about invariance of Φ and *inconclusive* result is returned (line 20).

2.4 Application for Checking Deadlock-Freedom

We present an application of the method for checking deadlock-freedom. To guarantee that global deadlocks are exclusively due to synchronizations, we use the local deadlock-freedom property of atomic components, that is if an atomic component reaches a state, it is always able to go out by at least one of the out-going transitions from that state. This property is checked for all the atomic components of the system to be verified before checking the global deadlock-freedom property of the system.

According to Definition 13, the predicate DIS characterizes a set of deadlock states, i.e a set of states from which no interaction can take place. A system is deadlock-free if the predicate $\neg DIS$ is an invariant because in that case, all the reachable states of the system satisfy $\neg DIS$ which means that no state in DIS is reachable.

To check that $\neg DIS$ is an invariant, we need a stronger invariant Φ such that $\Phi \Rightarrow \neg DIS$ or equivalently $\Phi \wedge DIS = false$. We apply the algorithm 1 and here the invariance property Φ to be proved is $\neg DIS$.

Example 25 *This example illustrates the verification of deadlock-freedom of the Temperature Control System. The DIS predicate of the system is as follows (Example 7):*

$$DIS = (\neg(l_5 \wedge \theta < 1000)) \bigwedge (\neg(l_6 \wedge \theta = 100) \vee \neg l_2) \bigwedge (\neg(l_6 \wedge \theta > 100)) \bigwedge (\neg(l_5 \wedge \theta = 1000) \vee \neg(l_3 \wedge t_2 \geq 3600)) \bigwedge (\neg(l_5 \wedge \theta = 1000) \vee \neg(l_1 \wedge t_1 \geq 3600)) \bigwedge (\neg(l_6 \wedge \theta = 100) \vee \neg l_4)$$

$\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$ *is the conjunction of the component invariants given in example 10. The predicate* $\Phi \wedge DIS$ *is satisfiable and it is the disjunction of the following terms:*

1. $(l_1 \wedge 0 \leq t_1 < 3600) \wedge (l_3 \wedge 0 \leq t_2 < 3600) \wedge (l_6 \wedge \theta = 100)$
2. $(l_1 \wedge 0 \leq t_1 < 3600) \wedge (l_4 \wedge t_2 \geq 3600) \wedge (l_5 \wedge \theta = 1000)$
3. $(l_1 \wedge 0 \leq t_1 < 3600) \wedge (l_3 \wedge 0 \leq t_2 < 3600) \wedge (l_5 \wedge \theta = 1000)$

4. $(l_2 \wedge t_1 \geq 3600) \wedge (l_3 \wedge 0 \leq t_2 < 3600) \wedge (l_5 \wedge \theta = 1000)$

5. $(l_2 \wedge t_1 \geq 3600) \wedge (l_4 \wedge t_2 \geq 3600) \wedge (l_5 \wedge \theta = 1000)$

Each one of the above terms represents a family of possible deadlocks. To decrease the number of potential deadlocks, we find a new invariant Φ' stronger than Φ, such that $\Phi' = \Phi \wedge \Psi$, where Ψ is the global interaction invariant obtained in Example 24.

The predicate $\Phi' \wedge DIS$ is reduced to:

6. $(l_1 \wedge 1 \leq t_1 < 3600) \wedge (l_3 \wedge 1 \leq t_2 < 3600) \wedge (l_5 \wedge \theta = 1000)$

7. $(l_1 \wedge 1 \leq t_1 < 3600) \wedge (l_4 \wedge t_2 \geq 3600) \wedge (l_5 \wedge \theta = 1000)$

8. $(l_2 \wedge t_1 \geq 3600) \wedge (l_3 \wedge 1 \leq t_2 < 3600) \wedge (l_5 \wedge \theta = 1000)$

Finally, it can be checked by using finite state reachability analysis on an abstraction of the system without variables, that only the first term represents feasible deadlocks, the two other being spurious. This term characterizes deadlock configurations leading to complete shutdown.

2.5 Summary

We have introduced our compositional method for verifying safety properties of component-based systems. The method is based on two kinds of invariants characterizing both local and global constraints of systems: component invariants and interaction invariants. Component invariants are over approximation of reachable states sets of components and are computed by using forward propagation. Interaction invariants characterize global constraints related to strong synchronizations between components. We have proposed two methods based on Positive Mapping and Fixed-point for computing symbolically the set of interaction invariants.

The methods for computing interaction invariants are applied for systems without data. For systems with data, we need to abstract away data before applying the methods. The concrete invariants are then obtained by concretizing abstract ones. An abstraction technique based on the component invariants is therefore introduced.

We have also presented an algorithm to verify safety properties by using component invariants and interaction invariants.

Finally, we shown an application of the compositional verification method on checking deadlock-freedom. We illustrated the method on an example, the Temperature Control System. We proved that the system is not deadlock-free and provided potential deadlocks of the system.

The innovation of our compositional verification method is that we use interaction invariants to characterize contexts of individual components. By using component invariants and interaction invariants, we have successfully combined constraints on both local and global aspects of systems. Moreover, the techniques we use to analyze systems are lightweight, hence it is possible to increase the size and the complexity of systems that can be handled.

The next chapter presents incremental construction and verification methods.

CHAPTER 3

Incremental Construction and Verification

Contents

Compositional verification approach avoids the state space explosion problem and therefore allows increasing significantly the size and complexity of the systems that can be handled. In the previous chapter, we presented our compositional verification method which is based on the use of invariants. Though we use lightweight techniques for computing invariants, the method may still suffer from the fast increasing size and complexity of systems. The computation of invariants from scratch for a system having thousands of components might be very expensive.

Moreover, nowadays the incremental construction deals with the complexity of the heterogeneous and large-scale systems in the construction phase. The idea is that composite systems can be considered as the composition of smaller parts. The verification should take advantage of the incremental construction process by integrating verification into construction phase in order to detect as soon as possible errors in the model. The verification should also be able to reuse the established properties of sub-systems in the verification of the global system.

The motivation of the work in this chapter is to provide a systematic methodology for the incremental construction and verification of component-based systems. We first

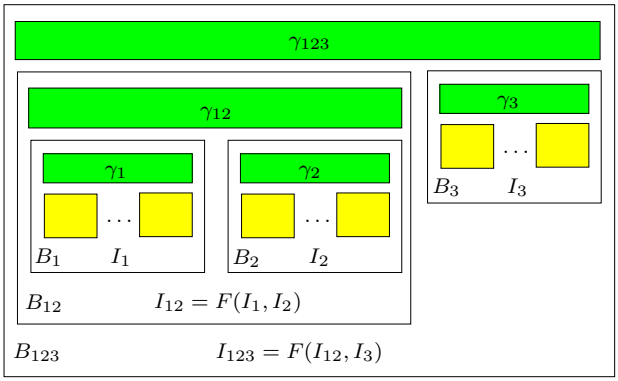


Figure 3.1: Incremental Verification Idea

formalize the incremental construction of component-based construction, then we propose rules on invariant preservation from which the already established invariants would not be violated during the incremental construction.

However, when the incremental construction is beyond of the invariant preservation rules, the verification process is still required to ensure the system correctness and the generation of new invariants is needed. Therefore, we propose a method for the incremental computation of invariants. It takes advantage of the system structure for coping with complexity of monolithic verification. The incremental method allows reusing the computed invariants from sub-systems which can be considered as the decomposed parts according to concepts of the incremental construction. The reuse of established invariants reduces significantly both time and memory usage in the verification of the system.

Figure 3.1 illustrates the idea of the method for the incremental computation of invariants. First we consider the composite component $B_{12} = \gamma_{12}(B_1, B_2)$ which is built from two constituents B_1, B_2 with its established invariants I_1, I_2. The incremental method allows computing invariants of the composite component B_{12} from invariants of its constituents: $I_{12} = F(I_1, I_2)$. Similarly, if B_{12} is composed with another constituent B_3 to build another composite component B_{123}, then the invariants of B_{123} are also computed from the established invariants of B_{12} and B_3: $I_{123} = F(I_{12}, I_3)$. It means that invariants of a composite components are always computed from the invariants of its constituents.

The chapter is organized as follows: first we give a formal definition of incremental construction based on the operation of increment of a connector. At some stage of the construction, a component can be transformed only by an increment operation which enforces synchronization between interactions of its connectors. The construction is hierarchical: increments can be applied ether at the same level or at different levels. We also provide rules to preserve the invariants during the incremental construction. Since the invariant preservation rules are not always satisfied by the systems, we present, in section 2, a method for incremental computation of invariants of a composite component from invariants of its constituent components. We finish the chapter by some conclusions.

In this chapter, the incremental verification method is considered for systems without

data. For systems with data, we need to use the abstraction technique presented in the previous chapter and then apply the method on abstract systems. We also recall that, to simplify notation, for a connector $\gamma = \{a_1, \ldots, a_n\}$, we write $\gamma = a_1 + \cdots + a_n$.

3.1 Incremental Construction and Invariant Preservation

In component-based systems, the construction of composite component is hierarchical and step-wise. We assume that a system is obtained from a set of atomic components represented by their behavior by adding progressively interactions. It is important to ensure the system correctness by verification during the construction in order to detect early errors. At some stage of the construction we have a component $\gamma(B)$ and a set of established invariants. We want to preserve the already established invariants after adding new interactions - an incremental modification of the behavior. In this section, we present the incremental construction framework and the rules for that the established invariants are preserved.

3.1.1 Incremental Construction

In the incremental construction of component-based systems, layers of connectors are applied to build the system bottom-up. $\gamma_\perp(B)$ can be viewed as the initial composite component obtained as the interleaving of individual components, where $B = (B_1, \ldots, B_n)$. If at some stage of the construction, we have obtained a component $\gamma(B)$, the construction process continues by enforcing new synchronizations on interactions of γ. We call these interactions generated by enforcing new synchronizations *increments*. When building a composite system in a bottom-up manner, it is essential that some already enforced synchronizations are not relaxed when increments are added. To guarantee this property, we propose the notion of *forbidden interactions*.

Definition 25 (Closure and Forbidden Interactions) *Let γ be a connector.*

- *The closure γ^c of γ, is the set of the non-empty interactions contained in some interaction of γ. That is $\gamma^c = \{a \neq \emptyset \mid \exists b \in \gamma.\ a \subseteq b\}$.*
- *The forbidden interactions γ^f of γ is the set of the interactions strictly contained in all the interactions of γ. That is $\gamma^f = \gamma^c - \gamma$.*

It is easy to see that for two connectors γ_1 and γ_2, we have $(\gamma_1 + \gamma_2)^c = \gamma_1^c + \gamma_2^c$ and $(\gamma_1 + \gamma_2)^f = (\gamma_1 + \gamma_2)^c - \gamma_1 - \gamma_2$.

Example 26 *Consider a connector $\gamma = p_1p_2 + p_3 + p_4$, we have $\gamma^c = p_1 + p_2 + p_3 + p_4$ and $\gamma^f = \gamma^c - \gamma = p_1 + p_2$.*

In our theory, a connector describes a set of interactions and, by default, also those interactions in where only one component can make progress. This assumption allows us to define new increments in terms of existing interactions.

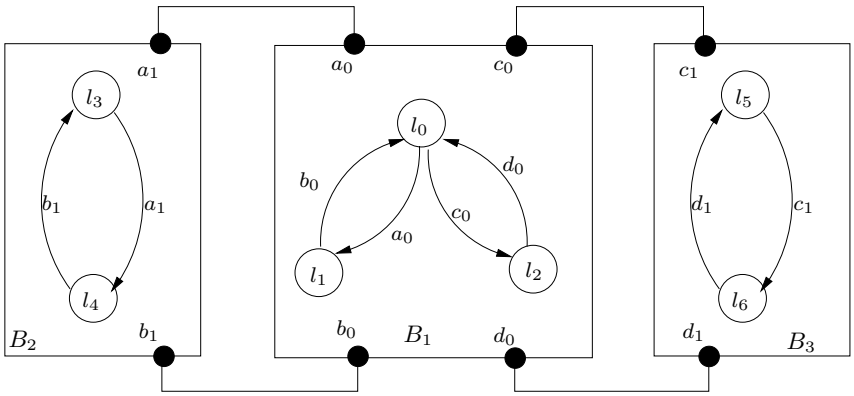


Figure 3.2: Incremental construction example

Definition 26 (Increments) *Consider a connector γ over B and let $\delta \subseteq 2^\gamma$ be a set of interactions. We say δ is an increment over γ if for any interaction $a \in \delta$ we have interactions $b_1, \ldots, b_n \in \gamma$ such that $\bigcup_{i=1}^{n} b_i = a$.*

In practice, one has to make sure that existing interactions defined by γ will not break the synchronizations that are enforced by the increment δ. For doing so, we remove from the original connector γ all the interactions that are forbidden by δ. This is done with the operation of *Layering*, which describes how an increment can be added to an existing set of interactions without breaking synchronization enforced by the increment. Formally, we have the following definition.

Definition 27 (Layering) *Given a connector γ and an increment δ over γ, the new set of interactions obtained by combining δ and γ, also called layering, is given by the following set $\delta\gamma = (\gamma - \delta^f) + \delta$ the incremental construction by layering, that is, the incremental modification of γ by δ.*

The above definition describes one-layer incremental construction. By the successive application of increments, we can construct the system with multiple layers.

Example 27 *Consider the example presented in Figure 3.2, let $\gamma = a_0 + b_0 + c_0 + d_0 + a_1 + b_1 + c_1 + d_1$ and $\delta_1 = a_0a_1 + b_0b_1$, we have $\delta_1\gamma = a_0a_1 + b_0b_1 + c_0 + d_0 + c_1 + d_1$.*

Besides the fusion of interactions, incremental construction can also be obtained by first combining the increments and then apply the result to the existing system. This process is called *Superposition*. Formally, we have the following definition.

Definition 28 (Superposition) *Given two increments δ_1, δ_2 over a connector γ, the operation of superposition between δ_1 and δ_2 is defined by $\delta_1 + \delta_2$.*

Superposition can be seen as a composition between increments, with looser coupled relation. If we combine the superposition of increments with the layering proposed in Definition

27, then we obtain an incremental construction from a set of increments. Formally, we have the following proposition.

Proposition 9 *Let γ be a connector over B, the incremental construction by the superposition of n increments $\{\delta_i\}_{1\leq i\leq n}$ is given by*

$$(\sum_{i=1}^{n}\delta_i)\gamma = (\gamma - (\sum_{i=1}^{n}\delta_i)^f) + \sum_{i=1}^{n}\delta_i \tag{3.1}$$

The above proposition provides a way to transform incremental construction by a set of increments into the separate constituents, where $\gamma - (\Sigma_{i=1}^{n}\delta_i)^f$ is the set of interactions that are not tightened during the incremental construction process.

We conclude the subsection with the following example.

Example 28 *In the example of Figure 3.2, let $\gamma = a_0+b_0+c_0+d_0+a_1+b_1+c_1+d_1$. Two increments are $\delta_1 = a_0a_1 + b_0b_1$ and $\delta_2 = c_0c_1 + d_0d_1$. When we consider two increments together, we have $(\delta_1 + \delta_2)\gamma = a_0a_1 + b_0b_1 + c_0c_1 + d_0d_1$.*

Notice that $(\delta_1+\delta_2)\gamma \neq \delta_1\gamma+\delta_2\gamma$. For example, $\delta_1\gamma+\delta_2\gamma = a_0a_1+b_0b_1+c_0c_1+d_0d_1+a_0+b_0+c_0+d_0+a_1+b_1+c_1+d_1$ in Example 28. The reason is that $\delta_1\gamma+\delta_2\gamma$ means the composition of two connectors.

3.1.2 Invariant Preservation in Incremental Construction

In Sub-section 3.1.1, we have presented a methodology for the incremental design of composite systems. In this section, we study the concept of *invariant preservation*. More precisely, we propose sufficient conditions that guarantee that already satisfied invariants are not violated when new interactions are added to the design.

We start by introducing the *looser synchronization preorder* on connectors, which we will use to characterize invariant preservation. As we have seen, interactions characterize the behavior of a composite component. We observe that if two interactions do not contain the same port, the execution of one interaction will not block the execution of the other interaction. Formally, we have the following definition of conflict-free interactions.

Definition 29 (Conflict-free Interactions) *Given a connector γ, let a_1, $a_2 \in \gamma$, if $a_1 \cap a_2 = \emptyset$, we say that there is no conflict between a_1 and a_2. If there is no conflict between any interactions of γ, we say that γ is conflict-free.*

The conflict-free connector ensures that the execution of one interaction will not disable other interactions. For example, connector $\gamma = p_1p_2 + p_2p_3$ is not conflict-free. The execution of p_1p_2 makes a transition labeled by p_2 move to its target location from the source location and p_2p_3 may not be enabled.

We now propose a preorder relation that allows to guarantee the absence of conflicts when new interactions are added. Formally, we have the following definition.

Definition 30 (Looser synchronization Pre-order) *We define the looser synchronization pre-order* $\preccurlyeq \subseteq 2^{2^P} \times 2^{2^P}$. *For two connectors* γ_1, γ_2, $\gamma_1 \preccurlyeq \gamma_2$ *if for any interaction* $a \in \gamma_2$, *there exist interactions* $b_1, \ldots, b_n \in \gamma_1$, *such that* $a = \bigcup_{i=1}^{n} b_i$ *and there is no conflict between any* b_i *and* b_j, *where* $1 \leq i, j \leq n$ *and* $i \neq j$. *We simply say that* γ_1 *is looser than* γ_2.

The above definition requires that the stronger synchronization should be obtained by the fusion of conflict-free interactions. The reason is that the execution of interactions may be disturbed by two conflict interactions, i.e., the execution of one interaction could block the transitions issued from the other interaction. However, if we fuse them together, it means that the transitions of both interactions can be executed, which violates the constraints of the previous behavior.

Example 29 *For two connectors* $\gamma_1 = \{p_1p_2, p_3p_4, p_5p_6, p_7p_8\}$ *which is conflict-free and* $\gamma_2 = \{p_1p_2p_3p_4, p_5p_6p_7p_8\}$, *we have* $\gamma_1 \preccurlyeq \gamma_2$.

It is easy to see that if γ_1, γ_2, γ_3, γ_4 are connectors such that $\gamma_1 \preccurlyeq \gamma_2$, and $\gamma_3 \preccurlyeq \gamma_4$, then we have $\gamma_1 + \gamma_3 \preccurlyeq \gamma_2 + \gamma_4$.

Definition 31 (Reachable States) *Given a connector* γ *over a component* B, L *is the set of states of* B, *we define* $Reach(l, \gamma(B)) = \{l_i \mid \exists a_i \in \gamma \wedge l \xrightarrow{a_i}{}^{*} l_i\} \cup \{l\}$ *the set of reachable states from* $l \in L$ *by any interaction of* γ.

The above definition provides a notation to record the set of reachable states from a state l through all possible interactions in $\gamma(B)$. If there is no executable interaction from l, we have that $reach(l, \gamma(B)) = \{l\}$.

Lemma 2 *Given two connectors* γ_1, γ_2 *over* B, *if* $\gamma_1 \preccurlyeq \gamma_2$, *we have* $Reach(l, \gamma_2(B)) \subseteq Reach(l, \gamma_1(B))$ *for any* $l \in L$,

Proof If there exists a path from $l \in L$ in $\gamma_2(B)$, we have $l \xrightarrow{a_1} l_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} l_m$, where $a_i \in \gamma_2$. Because $\gamma_1 \preccurlyeq \gamma_2$, for any a_i, we have a set of interactions $b_j \in \gamma_1$ such that $a_i = \bigcup_{j=1}^{k} b_j$. From any state l_i, there exists a set of interactions $\bigcup_{j=1}^{k} b_j$ such that $l_i \xrightarrow{b_1} \cdots \xrightarrow{b_k} l_{i+1}$. Therefore, we conclude that $Reach(l, \gamma_2(B)) \subseteq Reach(l, \gamma_1(B))$ for any $l \in L$.

This lemma shows that from the same state the set of reachable states under a tighter connector is always a subset of reachable states under a looser connector.

We now propose the following proposition which establishes a link between the Looser Synchronization Preorder and invariant preservation.

Proposition 10 *Let* γ_1, γ_2 *be two connectors over* B. *If* $\gamma_1 \preccurlyeq \gamma_2$, *we have* $inv(\gamma_1(B), I) \Rightarrow inv(\gamma_2(B), I)$.

Proof Let $Reach(l, \gamma_2(B))$ be the set of reachable states from the path started from $l \in L$ in $\gamma_2(B)$. Because $Reach(l, \gamma_2(B)) \subseteq Reach(l, \gamma_1(B))$, for any $l' \in Reach(l, \gamma_2(B))$, l' is reachable in $\gamma_1(B)$. As $inv(\gamma_1(B), I)$, we have $I(l')$. Then we can conclude that $inv(\gamma_2(B), I)$.

The above proposition, which will be used in the incremental design, simply says that if an invariant is satisfied, then it will remain when combinations of conflict-free interactions are added (following our incremental methodology) to the connector. This is not surprising as the tighter connector can only restrict the behaviors of the composite system.

We now switch to the more interesting problem of providing sufficient conditions to guarantee that invariants are preserved by the incremental construction.

Proposition 11 *Let γ be a connector over B and δ be an increment of γ such that $\gamma \preccurlyeq \delta$, then we have $\gamma \preccurlyeq \delta\gamma$.*

Proof Because $\gamma \preccurlyeq \gamma - \delta^f$, we have $\gamma \preccurlyeq (\gamma - \delta^f) + \delta = \delta\gamma$.

The above proposition, together with Proposition 10, says that the addition of an increment preserves the invariant if the initial connector is looser than the increment.

We continue our study and discuss the invariant preservation between the components obtained from superposition of increments and separately applying increments over the same set of components. We use the following definition.

Definition 32 (Interference-free Connectors) *Given two connectors γ_1, γ_2, for any $a_1 \in \gamma_1$, $a_2 \in \gamma_2$, if either a_1 and a_2 are conflict-free or $a_1 = a_2$, we say that γ_1 and γ_2 are interference-free.*

This definition considers a relation between two connectors. We observe that two interference-free connectors will not break or block the synchronizations specified by each other. Though we require that the interactions between γ_1 and γ_2 are conflict-free, γ_1 or γ_2 respectively can contain conflict interactions. For example, consider two connectors $\gamma_1 = p_1\ p_2 + p_2\ p_3$, $\gamma_2 = p_4\ p_5$. γ_1 is not conflict-free, but γ_1 and γ_2 are interference-free.

Lemma 3 *Given two interference-free connectors γ_1, γ_2, we have $\gamma_1 \cap \gamma_2^f = \emptyset$ and $\gamma_2 \cap \gamma_1^f = \emptyset$, and $(\gamma_1 + \gamma_2)^f = \gamma_1^f + \gamma_2^f$.*

Proof Since γ_1 and γ_2 are interference-free, if $\gamma_1 \cap \gamma_2 = \emptyset$, we have $\gamma_1 \cap \gamma_2^f = \emptyset$ and $\gamma_2 \cap \gamma_1^f = \emptyset$. If $\gamma_1 \cap \gamma_2 \neq \emptyset$, for any $a \in \gamma_1 \cap \gamma_2$, we know that $a \notin \gamma_1^f$ and $a \notin \gamma_2^f$. $\gamma_1 \cap \gamma_2^f = \emptyset$ and $\gamma_2 \cap \gamma_1^f = \emptyset$ are still correct.

According to Definition 25, we have $(\gamma_1 + \gamma_2)^f = \gamma_1^c + \gamma_2^c - (\gamma_1 + \gamma_2) = (\gamma_1^c - (\gamma_1 + \gamma_2)) + (\gamma_2^c - (\gamma_1 + \gamma_2))$. Because γ_1 and γ_2 are interference-free, $\gamma_1^c - (\gamma_1 + \gamma_2) = \gamma_1^c - \gamma_1 = \gamma_1^f$ and $\gamma_2^c - (\gamma_1 + \gamma_2) = \gamma_2^f$. So we have $(\gamma_1 + \gamma_2)^f = \gamma_1^f + \gamma_2^f$.

We now present the main result of the section.

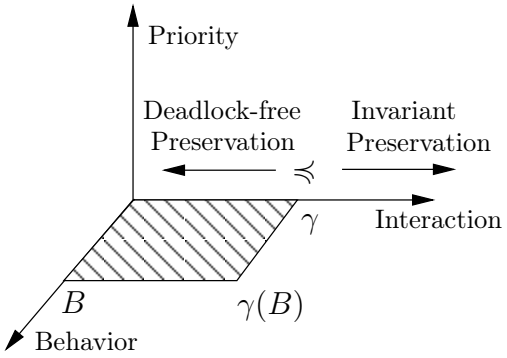


Figure 3.3: Invariant preservation for looser synchronization relation

Proposition 12 *Consider two increments δ_1, δ_2 over γ such that $\gamma \preccurlyeq \delta_1$ and $\gamma \preccurlyeq \delta_2$, if δ_1 and δ_2 are interference-free, and $inv(\delta_1\gamma(B), I_1)$, $inv(\delta_2\gamma(B), I_2)$, we have $inv((\delta_1 + \delta_2)\gamma(B), I_1 \wedge I_2)$.*

Proof We will show that $\delta_1\gamma \preccurlyeq (\delta_1 + \delta_2)\gamma$ and $\delta_2\gamma \preccurlyeq (\delta_1 + \delta_2)\gamma$, then the conclusion can be obtained Proposition 10.

Because δ_1 and δ_2 are interference-free, we have $(\delta_1+\delta_2)^f = \delta_1^f+\delta_2^f$, then $\gamma-(\delta_1+\delta_2)^f = \gamma - (\delta_1^f + \delta_2^f)$. As $\gamma - (\delta_1^f + \delta_2^f) \subseteq \gamma - \delta_1^f$, we obtain that $\gamma - \delta_1^f \preccurlyeq \gamma - (\delta_1^f + \delta_2^f)$ and $\gamma - \delta_1^f + \delta_1 \preccurlyeq \gamma - (\delta_1^f + \delta_2^f) + \delta_1$. Because δ_1 and δ_2 are interference-free, $\delta_2 \cap \delta_1^f = \emptyset$ and $\gamma \preccurlyeq \delta_2$, we have $\gamma - \delta_1^f \preccurlyeq \delta_2$. So $\gamma - \delta_1^f + \delta_1 \preccurlyeq \gamma - (\delta_1^f + \delta_2^f) + \delta_1 + \delta_2$. The same rule can be applied to $\delta_2\gamma$. Therefore, we have $\delta_1\gamma \preccurlyeq (\delta_1 + \delta_2)\gamma$ and $\delta_2\gamma \preccurlyeq (\delta_1 + \delta_2)\gamma$, thus $inv((\delta_1 + \delta_2)\gamma(B), I_1 \wedge I_2)$.

The above proposition considers a set of increments $\{\delta_i\}_{1\leq i\leq n}$ over γ that are interference-free. The proposition says that if for any δ_i the separate application of increments over component $\delta_i\gamma(B)$ preserves the original invariants of $\gamma(B)$, then the system obtained from considering the superposition of increments over γ preserves the conjunction of the invariants of individual increments.

We now briefly study the relation between the looser synchronization preorder and *property preservation*. Figure 3.3 shows the three ingredients of the BIP toolset, that are (1) priorities, which we will not use here, (2) interactions, and (3) behaviors of components. We shall see that the looser synchronization preorder preserves invariants (Proposition 12). This means that the preorder preserves the so-called reachability properties. On the other hand, the preorder does not preserve deadlocks. Indeed, adding new interactions may lead to the addition of new deadlock conditions. Given two connectors γ_1 and γ_2 over component B such that γ_2 is tighter than γ_1, i.e. $\gamma_1 \preccurlyeq \gamma_2$, we can conclude that if $\gamma_2(B)$ is deadlock-free, then $\gamma_1(B)$ is deadlock-free. However, we can still reuse the invariant of $\gamma_1(B)$ as an over-approximation of the one of $\gamma_2(B)$.

Discussion. Though we can reuse invariants to save computation time, the invariants of the system with a looser connector may be too weak with respect to a new system obtained with a tighter connector. Consider the example given in Figure 3.4 and let $\gamma =$

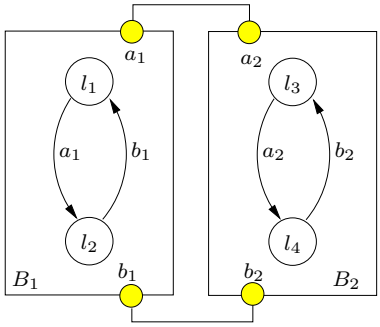


Figure 3.4: Example for the invariant preservation

$p_1 + p_2 + q_1 + q_2$, $\delta_1 = p_1\ p_2$, and $\delta_2 = q_1\ q_2$. By using the technique presented in the next section, we shall see that the invariant for $\delta_1\gamma(B)$ and $\delta_2\gamma(B)$ is $(l_1 \vee l_2) \wedge (l_3 \vee l_4)$. By applying Proposition 12, we obtain that this invariant is preserved for $(\delta_1 + \delta_2)\gamma(B)$. This invariant is weaker than the invariant $(l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3)$ that is directly computed on $(\delta_1 + \delta_2)\gamma(B)$. To overcome the above problem, we will now propose an approach that can be used to compute invariants in an incremental manner.

3.2 Incremental Computation of Invariants

In the previous section, we have shown the rules for invariant preservation during incremental construction. However, in general we can not always apply these rules. In this section, we put forward the method to compute incrementally invariants in more general case of incremental component-based construction which can also be applied for incremental verification. The method is lightweight because we can reuse the computed invariants from constituents that can be considered as the decomposed parts according to concepts of the incremental construction.

3.2.1 Incremental Computation of BBCs

From BBC definition and Theorem 1 and 2 in the previous chapter we know that the invariants can be computed from connectors. And Section 3.1.1 shows that the increments will not be modified during the incremental construction. Therefore, we could start from increments to compute incrementally their BBCs and invariants. In this subsection, we provide a method for incremental computation of global BBCs.

Lemma 4 *Given two connectors γ_1, γ_2 over B, we have*

$$|(\gamma_1 + \gamma_2)(B)| = |\gamma_1(B)| \wedge |\gamma_2(B)|$$

Proof By Definition 17, we have $|(\gamma_1 + \gamma_2)(B)| = \bigwedge_{a \in (\gamma_1+\gamma_2)} |a(B)| = \bigwedge_{a \in \gamma_1} |a(B)| \wedge \bigwedge_{a \in \gamma_2} |a(B)| = |\gamma_1(B)| \wedge |\gamma_2(B)|$.

The following proposition provides a method for obtaining the Boolean Behavioral Constraints taking into account component structure.

Proposition 13 *Let γ be a connector over B, the Boolean Behavioral Constraint for the system obtained by superposition of n increments $\{\delta_i\}_{1\leq i\leq n}$ can be written as*

$$|(\sum_{i=1}^{n}\delta_i)\gamma(B)| = |(\gamma - (\sum_{i=1}^{n}\delta_i)^f)(B)| \wedge \bigwedge_{i=1}^{n}|\delta_i(B)| \tag{3.2}$$

Proof By Equation 3.1, the union of $\gamma - (\sum_{i=1}^{n}\delta_i)^f$ and $\sum_{i=1}^{n}\delta_i$ is the set of interactions from the superposition of increments $\{\delta_i\}_{1\leq i\leq n}$ over γ. The proof can be concluded by the application of Lemma 4.

Proposition 13 provides a way to decompose the computation of BBCs with respect to increments. The decomposition is based on the fact that different increments describe the interactions between different components. To simplify the notation, $\gamma - (\Sigma_{i=1}^{n}\delta_i)^f$ is represented by δ_0. We have the following example.

Example 30 *For Example 28, consider the two increments $\delta_1 = a_0a_1 + b_0b_1$ and $\delta_2 = c_0c_1 + d_0d_1$. The composite component is $(\delta_1 + \delta_2)\gamma = a_0a_1 + b_0b_1 + c_0c_1 + d_0d_1$. The BBCs for interactions of δ_1 and δ_2 respectively are*

$$|\delta_1(B)| = (l_0 \Rightarrow l_1 \vee l_4) \wedge (l_1 \Rightarrow l_0 \vee l_3) \wedge (l_3 \Rightarrow l_1 \vee l_4) \wedge (l_4 \Rightarrow l_0 \vee l_3)$$
$$|\delta_2(B)| = (l_0 \Rightarrow l_2 \vee l_6) \wedge (l_2 \Rightarrow l_0 \vee l_5) \wedge (l_5 \Rightarrow l_2 \vee l_6) \wedge (l_6 \Rightarrow l_0 \vee l_5)$$

Because $\gamma - (\delta_1 + \delta_2)^f = \emptyset$, we have $|(\delta_1 + \delta_2)\gamma(B)| = |\delta_1(B)| \wedge |\delta_2(B)|$ where $|\delta_1(B)|$ and $|\delta_2(B)|$ are the BBCs above.

3.2.2 Incremental Computation of Invariants based on Positive Mapping

In the previous sub-section we have shown how to compute incrementally BBCs. The BBCs of a composite component can be obtained as the conjunction of BBCs of constituent components. In this sub-section, we propose a method which allows computing incrementally invariants from invariants obtained from increments. It does not consider the relations between increments which is more flexible.

To distinguish the shared variables between different BBCs, we define the common location variables shared by multiple connectors. We recall that for an interaction a, we denote by ${}^\bullet a$ (respectively $a^\bullet$) the set of source locations (respectively destination locations) of the transitions involved in a. We also extend this notation for connectors: ${}^\bullet\gamma = \bigcup_{a\in\gamma} {}^\bullet a$ (respectively $\gamma^\bullet = \bigcup_{a\in\gamma} a^\bullet$).

We propose the following definition that will help in the process of reusing existing invariants.

Definition 33 (Common Location Variables L_c) *Given a set of connectors $\{\gamma_1, \ldots, \gamma_n\}$, we define the set of common location variables by:*

$$L_c \;=\; \bigcup_{i,j\in[1,n]\wedge i\neq j}(sup(\gamma_i)\cap sup(\gamma_j))$$

where $sup(\gamma) = {}^\bullet\gamma \cup \gamma^\bullet$ *is the set of locations involved in some interaction* a *of* γ.

When the set of common state variables is empty, the connectors are really disjoint. And there is no common variables between their BBCs.

Example 31 *For the example presented in Figure 3.2, consider* $\delta_1 = a_0a_1 + b_0b_1$ *and* $\delta_2 = c_0c_1 + d_0d_1$, *we have* $sup(\delta_1) = {}^\bullet(a_0a_1) \cup (a_0a_1)^\bullet \cup {}^\bullet(b_0b_1) \cup (b_0b_1)^\bullet = \{l_0, l_1, l_3, l_4\}$ *and similarly* $sup(\delta_2) = \{l_0, l_2, l_5, l_6\}$, *hence* $L_c = sup(\delta_1) \cap sup(\delta_2) = \{l_0\}$.

Given a set of increments $\{\delta_1, \ldots, \delta_n\}$ over γ, according to Proposition 9 and Proposition 13, during the incremental construction, the set of interactions $(\sum_{i=1}^n \delta_i)\gamma$ (respectively BBCs $|(\sum_{i=1}^n \delta_i)\gamma(B)|$) can be obtained from the sets of interactions $\gamma - \sum_{i=1}^n \delta_i$ and $\{\delta_i\}_{i=1}^n$ (respectively from the set of BBCs $|\gamma - \sum_{i=1}^n \delta_i(B)|$ and $\{|\delta_i(B)|\}_{i=1}^n$). From that we propose a method for computing interaction invariants of $(\sum_{i=1}^n \delta_i)\gamma(B)$ by using interaction invariants obtained from $|\gamma - \sum_{i=1}^n \delta_i(B)|$ and $\{|\delta_i(B)|\}_{i=1}^n$.

Proposition 14 *Consider a composite component* B. *Let* γ *be a connector for* B *and assume a set of increments* $\{\delta_i\}_{1\leq i\leq n}$ *over* $\gamma(B)$. *Let* $\delta_0 = \gamma - (\sum_{i=1}^n \delta_i)^f$, $I_{\delta_i} = \{\phi_k\}_{k\in\mathcal{I}_i}$, *for* $i = 0, \ldots, n$, *be the interaction invariants for each* $|\delta_i(B)|$, $S_{\delta_i} = \{m_k\}_{k\in\mathcal{I}_i}$, *for* $i = 0, \ldots, n$, *be the corresponding BBC-solutions, and let*

- L_ϕ *be the set of location variables in invariant* ϕ,
- L_c *be the common location variables between* $\{\delta_0, \delta_1, \ldots, \delta_n\}$.

Then the interaction invariant of $(\Sigma_{i=1}^n\delta_i)\gamma(B)$ *is obtained as follows:*

$$I = \left(\bigwedge_{i=0}^{n} \bigwedge_{\substack{k\in\mathcal{I}_i\wedge \\ L_{\phi_k}\cap L_c=\emptyset}} \phi_k\right) \wedge \left(\bigwedge_{(k_{i1},\ldots,k_{ir})\in\mathbb{D}} \bigvee_{j=1}^{r} \phi_{k_{ij}}\right)$$

where
$\mathbb{D} = \{(k_{i1}, \ldots, k_{ir}) | (k_{ij} \in \mathcal{I}_{ij}\ \forall j = 1 \ldots r) \wedge (L_{\phi_{k_{ij}}} \cap L_c \neq \emptyset) \wedge (\bigwedge_{j=1}^r m_{k_{ij}} \neq false) \wedge ((k_{i1}, \ldots, k_{ir})\ is\ maximal)\}$.

Proof In every S_{δ_i}, there exists a solution m_{0i} without any variables in the positive form, which has no invariant corresponding to. For any $\phi_k, k \in \mathcal{I}_i$, there exists m_k such that $\phi_k = \widetilde{m_k^p}$. According to Proposition 13, the BBC-solution of $|(\Sigma_{i=1}^n\delta_i)\gamma(B)|$ is $\bigwedge_{i=0}^n S_{\delta_i} = \bigwedge_{i=0}^n \bigvee_{k\in\mathcal{I}_i} m_k = \bigvee_{k0\in\mathcal{I}_0,\ldots,kn\in\mathcal{I}_n} \bigwedge_{i=0}^n m_{ki}$.

- If an m_{ki} does not contain any common location variables, there exists solution m_{0j} containing only negations in S_{δ_j} such that $i \neq j$ and $(\bigwedge_{j=0\wedge j\neq i}^n m_{ki} \wedge m_{0j})^p = m_{ki}^p$, so ϕ_{ki} is one of the BBC-invariants of $(\Sigma_{i=1}^n\delta_i)\gamma(B)$.

- If there is a maximal set $\{m_{k_{i1}}, \ldots, m_{k_{ir}}\}, k_{ij} \in \mathcal{I}_{ij} \forall j = 1 \ldots r$ such that all of them contain common location variables, and $\bigwedge_{j=1}^{r} m_{k_{ij}} = false$, it is not a solution of $|(\Sigma_{i=1}^{n} \delta_i)\gamma(B)|$. If $\bigwedge_{j=1}^{r} m_{k_{ij}} \neq false$, we have $(\widetilde{\bigwedge_{j=1}^{r} m_{k_{ij}}})^p = \widetilde{\bigwedge_{j=1}^{r} \phi_{k_{ij}}} = \bigvee_{j=1}^{r} \phi_{k_{ij}}$.

That is, if an interaction invariant obtained from $BBCs$ of δ_i contains only local location variables of δ_i, then it is also invariant of the global system after the superposition because it is not affected by others increments. If a set of invariants $\{\phi_{0j_0}, \ldots, \phi_{nj_n}\}$ obtained from $BBCs$ of $\{\delta_0, \ldots, \delta_n\}$ which all contain common location variables, they have influence on each other, hence we need to check the conjunction of the corresponding BBC-solutions. If the conjunction is not false, it is a global BBC-solution and then an interaction invariant of the global system is established by the disjunctions of all these invariants. Since each non common variable occurs only in one of the BBCs, and the conjunction of BBC-solutions is false or not depends only on the common location variables, we can delete the non-common negative variables separately by the positive mapping in every BBC-solutions, which drastically reduces complexity of computation.

Example 32 *In Example 30, we illustrate the BBCs for the two increments in the example presented in Figure 3.2 where* $\delta_1 = a_0a_1 + b_0b_1$ *and* $\delta_2 = c_0c_1 + d_0d_1$. *Here we show how to compute the interaction invariants of* $(\delta_1 + \delta_2)\gamma(B)$ *from interaction invariants obtained from the increments (for this example* $\gamma - (\delta_1 + \delta_2) = \emptyset$*).*

According to Example 31, we have $L_c = \{l_0\}$. *Let* S_{δ_1}, S_{δ_2} *be the BBC-solutions for* $|\delta_1(B)|$ *and* $|\delta_2(B)|$ *respectively:*

$$S_{\delta_1} = (\bar{l_0} \wedge \bar{l_1} \wedge \bar{l_3} \wedge \bar{l_4}) \vee (l_0 \wedge l_1) \vee (l_1 \wedge l_3) \vee (l_0 \wedge l_4) \vee (l_3 \wedge l_4)$$
$$S_{\delta_2} = (\bar{l_0} \wedge \bar{l_2} \wedge \bar{l_5} \wedge \bar{l_6}) \vee (l_0 \wedge l_2) \vee (l_2 \wedge l_5) \vee (l_0 \wedge l_6) \vee (l_5 \wedge l_6)$$

and $I_{\delta_1}, I_{\delta_2}$ *be their interaction invariants:*

$$I_{\delta_1} = (l_0 \vee l_1) \wedge (l_0 \vee l_4) \wedge (l_1 \vee l_3) \wedge (l_3 \vee l_4) \quad I_{\delta_2} = (l_0 \vee l_2) \wedge (l_0 \vee l_6) \wedge (l_2 \vee l_5) \wedge (l_5 \vee l_6)$$

By applying $I_{(\delta_1+\delta_2)\gamma(B)} = F(I_{\gamma-(\delta_1+\delta_2)^f}, I_{\delta_1}, I_{\delta_2})$, *we have:*

- *The invariants* $(l_1 \vee l_3), (l_3 \vee l_4), (l_2 \vee l_5), (l_5 \vee l_6)$ *do not contain any common location variables, so they are also interaction invariants of* $(\delta_1 + \delta_2)\gamma(B)$.

- *The invariants* $(l_0 \vee l_1), (l_0 \vee l_4)$ *(with the corresponding BBC-solutions* $(l_0 \wedge l_1), (l_0 \wedge l_4)$*) and* $(l_0 \vee l_2), (l_0 \vee l_6)$ *(with the corresponding BBC-solutions* $(l_0 \wedge l_2), (l_0 \wedge l_6)$*) contain the common location variable* l_0*, and the conjunction between any two monomials from two groups of BBC-solutions are not false, hence the disjunction of any two invariants from two groups of invariants is an invariant of* $(\delta_1 + \delta_2)\gamma(B)$*:* $(l_0 \vee l_1 \vee l_2), (l_0 \vee l_1 \vee l_6), (l_0 \vee l_2 \vee l_4), (l_0 \vee l_4 \vee l_6)$.

Finally, the global interaction invariant of $(\delta_1 + \delta_2)\gamma(B)$ *is:*

$$I_{(\delta_1+\delta_2)\gamma(B)} = (l_0 \vee l_1 \vee l_2) \wedge (l_0 \vee l_1 \vee l_6) \wedge (l_0 \vee l_2 \vee l_4) \wedge (l_0 \vee l_4 \vee l_6) \wedge (l_1 \vee l_3) \wedge (l_3 \vee l_4) \wedge (l_2 \vee l_5) \wedge (l_5 \vee l_6)$$

3.2.3 Incremental Computation of Invariants based on Fixed-point

In the previous chapter, we presented the method for computing interaction invariants based on fixed-point computation. According to Theorem 4, for a component $\gamma(B)$, the interaction invariants of $\gamma(B)$ can be obtained as the dual of the fixed-points $\widetilde{\mathbb{F}}(\mathbb{V}_\gamma, \bigvee_{l\in L} l)$. We call $\mathbb{F}(\mathbb{V}_\gamma, \bigvee_{l\in L} l)$ fixed-points of $\gamma(B)$. In this section, we provide a method which allows computing fixed-points of a composite component from the fixed-points of its constituents.

According to Proposition 9, the set of interactions $(\sum_{i=1}^n \delta_i)\gamma$ can be obtained from the sets of interactions $\gamma - (\sum_{i=1}^n \delta_i)^f$ and $\{\delta_i\}_{i=1}^n$. From that, we propose a method which allows computing fixed-points of $(\sum_{i=1}^n \delta_i)\gamma(B)$ from the fixed-points obtained from $\gamma - (\sum_{i=1}^n \delta_i)^f$ and $\{\delta_i\}_{i=1}^n$ over B.

First, for a set of connectors $\{\gamma_i\}_{i=1}^n$ over a component B, the following proposition allows getting the global image vector of $(\sum_{i=1}^n \gamma_i)(B)$ from the image vectors of $\gamma_i(B)$.

Proposition 15 *Given a set of connectors $\{\gamma_i\}_{i=1}^n$ over a set of components $B = \{B_i\}_{i=1}^n$ where $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^n L_i$. Let $\mathbb{V}_{\gamma_i}$ be image vector for all locations in L according to γ_i, then the image vector $\mathbb{V}_\gamma$ for all the locations of L according to $\sum_{i=1}^n \gamma_i$ can be obtained as follows:*

$$\mathbb{V}_\gamma(l) = \begin{cases} \bigwedge_{l\in {}^\bullet\gamma_i} \mathbb{V}_{\gamma_i}(l) & \text{if } l \in \bigcup_{i=1}^n {}^\bullet\gamma_i \\ l & \text{otherwise} \end{cases} \tag{3.3}$$

Proof We have:

$$\mathbb{V}_\gamma(l) = \bigwedge_{a\in\gamma\wedge l\in {}^\bullet a} \mathbb{V}_a(l) = \bigwedge_{l\in {}^\bullet\gamma_i} (\bigwedge_{a\in\gamma_i\wedge l\in {}^\bullet a} \mathbb{V}_a(l)) = \bigwedge_{l\in {}^\bullet\gamma_i} \mathbb{V}_{\gamma_i}(l)$$

The Proposition 15 allows to compute the image vector of $(\sum_{i=1}^n \delta_i)\gamma(B)$ from the image vector of $\delta_0 = \gamma - (\sum_{i=1}^n \delta_i)^f$ and $\{\delta_i\}_{i=1}^n$ as follows:

$$\mathbb{V}(l) = \begin{cases} \bigwedge_{l\in {}^\bullet\delta_i} \mathbb{V}_{\delta_i}(l) & \text{if } l \in \bigcup_{i=0}^n {}^\bullet\delta_i \\ l & \text{otherwise} \end{cases} \tag{3.4}$$

Example 33 *In the example of Figure 3.5, let $\gamma = a_1{+}b_1{+}c_1{+}a_2{+}b_2{+}c_2$, $\delta_1 = a_1a_2{+}c_1c_2$, $\delta_2 = b_1b_2$, we have:*

$$(\delta_1 + \delta_2)\gamma = (\gamma - (\delta_1 + \delta_2)^f) + \delta_1 + \delta_2 = a_1a_2 + c_1c_2 + b_1b_2$$

The image for every location according to δ_1 are as follows:

$$\begin{array}{lll} \mathbb{V}_{\delta_1}(l_1) = l_1 \wedge l_2 \vee l_1 \wedge l_5, & \mathbb{V}_{\delta_1}(l_2) = l_2, & \mathbb{V}_{\delta_1}(l_3) = l_1 \wedge l_3 \vee l_3 \wedge l_4 \\ \mathbb{V}_{\delta_1}(l_4) = l_2 \wedge l_4 \vee l_4 \wedge l_5, & \mathbb{V}_{\delta_1}(l_5) = l_5, & \mathbb{V}_{\delta_1}(l_6) = l_1 \wedge l_6 \vee l_4 \wedge l_6 \end{array}$$

The image of every location according to δ_2 are as follows:

$$\begin{array}{lll} \mathbb{V}_{\delta_2}(l_1) = l_1, & \mathbb{V}_{\delta_2}(l_2) = l_2 \wedge l_3 \vee l_2 \wedge l_6, & \mathbb{V}_{\delta_2}(l_3) = l_3 \\ \mathbb{V}_{\delta_2}(l_4) = l_4, & \mathbb{V}_{\delta_2}(l_5) = l_5 \wedge l_6 \vee l_3 \wedge l_5, & \mathbb{V}_{\delta_2}(l_6) = l_6 \end{array}$$

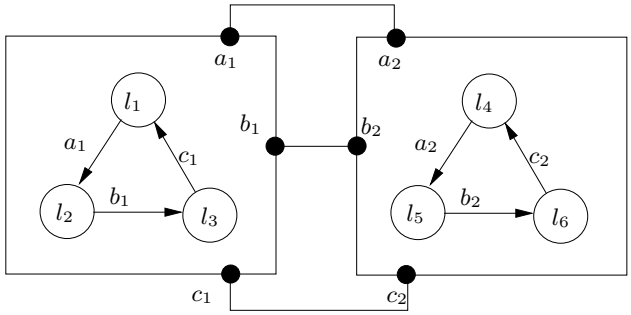


Figure 3.5: Example for incremental invariant computation

For $(\delta_1 + \delta_2)\gamma(B)$, the image vector $\mathbb{V}$ is obtained from those of its increments as follows:

$$\mathbb{V}(l_1) = l_1 \wedge l_2 \vee l_1 \wedge l_5, \quad \mathbb{V}(l_2) = l_2 \wedge l_3 \vee l_2 \wedge l_6$$
$$\mathbb{V}(l_3) = l_1 \wedge l_3 \vee l_3 \wedge l_4, \quad \mathbb{V}(l_4) = l_2 \wedge l_4 \vee l_4 \wedge l_5$$
$$\mathbb{V}(l_5) = l_5 \wedge l_6 \vee l_3 \wedge l_5, \quad \mathbb{V}(l_6) = l_1 \wedge l_6 \vee l_4 \wedge l_6$$

The following proposition allows computing fixed-points for $(\sum_{i=1}^{n} \delta_i)\gamma(B)$ from the fixed-points obtained from $\gamma - (\sum_{i=1}^{n} \delta_i)^f$ and $\{\delta_i\}_{i=1}^{n}$ over B.

Proposition 16 *Given a connector γ over a set of components B, a set of increments $\{\delta_1, \ldots, \delta_n\}$ over γ, and sets of fixed-points $\{S_i\}_{i=0}^{n}$ where:*

- $S_0 = \mathbb{F}(\mathbb{V}_{\delta_0}, \bigvee_{l \in {}^\bullet\delta_0} l)$ *with* $\delta_0 = \gamma - (\sum_{i=1}^{n} \delta_i)^f$,
- $S_i = \mathbb{F}(\mathbb{V}_{\delta_i}, \bigvee_{l \in {}^\bullet\delta_i} l)$ *with* $1 \leq i \leq n$.

Let $\mathbb{V}$ be the image vector according to $(\sum_{i=1}^{n} \delta_i)\gamma$, the fixed-points $\mathbb{F}(\mathbb{V}, \bigvee_{i=0}^{n} S_i)$ are the fixed-points of $(\sum_{i=1}^{n} \delta_i)\gamma(B)$.

Proof Given two sets of monomials S_1, S_2, we denote $S_1 \sqsubseteq S_2$ if if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ such that s_2 implies s_1.

From Proposition 15, for any set of interactions $\gamma' \in \{\gamma - (\sum_{i=1}^{n} \delta_i)^f, \delta_1, \ldots, \delta_n\}$ and for any location $l \in {}^\bullet\gamma'$ we have $\mathbb{V}_{\gamma'}(l) \sqsubseteq \mathbb{V}_{(\sum \delta_i)\gamma}(l)$. Let S_i and S be respectively the fixed-points obtained from l by $\mathbb{V}_{\gamma'}$ and $\mathbb{V}_{(\sum \delta_i)\gamma}$, we have $l \sqsubseteq S_i \sqsubseteq S$. Therefore by starting from $\bigvee S_i$, the fixed-points $\mathbb{F}(\mathbb{V}, \bigvee_{i=0}^{n} S_i)$ are the fixed-points of $(\sum_{i=1}^{n} \delta_i)\gamma(B)$.

The number of iterations to reach fixed-points $\mathbb{F}(\mathbb{V}, \bigvee_i S_i)$ starting from S_i can be significantly smaller compared to the number of iterations to reach fixed-points $\mathbb{F}(\mathbb{V}, \bigvee_i l_i)$ where we start from locations. Therefore the computation cost can be significantly reduced in both time and memory usage.

Example 34 *In the example of Figure 3.5 with $\gamma = a_1 + b_1 + c_1 + a_2 + b_2 + c_2$, $\delta_1 = a_1a_2 + c_1c_2$, $\delta_2 = b_1b_2$, since $\gamma - (\delta_1 + \delta_2)^f = \emptyset$, we have two sets of fixed-points corresponding to two increments:*

$$S_1 = \mathbb{F}(\mathbb{V}_{\delta_1}, \bigvee_{l \in {}^\bullet\delta_1} l) = (l_1 \wedge l_2) \vee (l_1 \wedge l_5) \vee (l_4 \wedge l_5) \vee (l_2 \wedge l_4)$$
$$S_2 = \mathbb{F}(\mathbb{V}_{\delta_2}, \bigvee_{l \in {}^\bullet\delta_2} l) = (l_2 \wedge l_3) \vee (l_2 \wedge l_6) \vee (l_5 \wedge l_6) \vee (l_3 \wedge l_5)$$

Let $\phi^0 = S_1 \vee S_2$*, the iteration with the image vector* $\mathbb{V}$ *obtained in Example 33 provides*

$$\begin{aligned}\phi^1 \ &= (l_1 \wedge l_2 \wedge l_3) \vee (l_1 \wedge l_2 \wedge l_6) \vee (l_1 \wedge l_5 \wedge l_6) \vee (l_1 \wedge l_3 \wedge l_5) \\ &\vee \ (l_2 \wedge l_3 \wedge l_4) \vee (l_2 \wedge l_4 \wedge l_6) \vee (l_4 \wedge l_5 \wedge l_6) \vee (l_3 \wedge l_4 \wedge l_5)\end{aligned}$$

Then $\phi^2 = \phi^1$*, so* $\mathbb{F}(\mathbb{V}, S_1 \vee S_2) = \phi^1$ *are the fixed-points of* $(\delta_1 + \delta_2)\gamma(B)$.

3.3 Summary

We have presented methods allowing incremental construction and verification of component-based systems. Consider a system built from its constituents, we provide conditions in which the established invariants are preserved after the construction. However these conditions are quite limited because many systems do not satisfy these conditions. We therefore proposed two methods for incrementally computing invariants of general systems from the established invariants of their constituent: one is based on positive mapping operation and the other is based on fixed-point computation. The reuse of established invariants (and established fixed-points) reduces significantly the computation cost compared to the operation on the global system from scratch.

We have presented compositional and incremental methods for verification of component-based systems. However, all these methods are limited to systems without data transfer. Hence we are going to present, in the next chapter, a method for dealing with systems with data transfer.

CHAPTER 4

Dealing with Data Transfer

Contents

4.1 Idea and Methodology

The BIP framework allows to define rich interaction models by using interactions extended with data as presented in [Bas08]. However, in the previous chapters, we considered the interaction models without data transfer. Although it allows easier decomposition and compositional reasoning on the system, the absence of data transfer between components is a severe limitation in practice. The difficulty of handling the data transfer is that we can not compute the component invariants of an atomic component in isolation from the others. For example, if a variable involved in a local transition is updated during the execution of an interaction, we should propagate the changes of valuations from the interaction into the transition.

Our current compositional verification method ignores the data transfer in interactions and therefore get a coarser abstraction. Nevertheless, we still expect to approximate the state space by a tighter abstraction. And that motivates us to study the abstraction method toward the interaction with data transfer. The idea is, since we already have a method for dealing with systems containing only pure interactions, that is interactions without data transfer, we want to transform the interaction models with data transfer into the models without data transfer on which we can apply our previous method. In this chapter, we

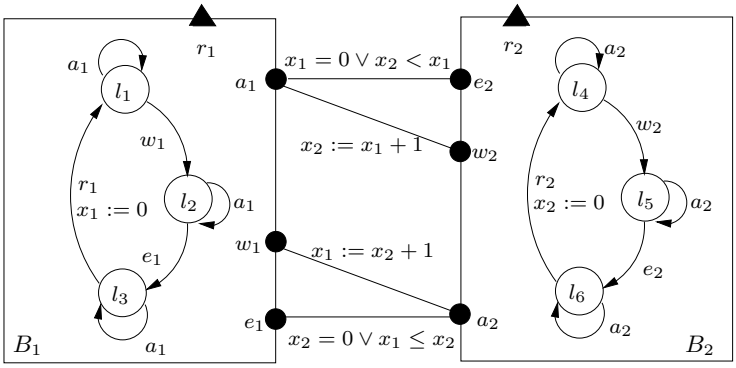


Figure 4.1: Modeling of Bakery in BIP

propose methods for computing invariants taking into account the data transfer. For component invariants, we project the changes on interactions to transitions for the computation of post predicate transformers over transitions. For interaction invariants, we propose to generate new components to "replace" the data transfer on interactions and then apply our previous methods to compute interaction invariants. Our method for dealing with data transfer in BIP framework can be applied to other synchronous modeling mechanisms that do not support shared variables.

The organization of the chapter is as follows: first we present a method for computing component invariants in systems with data transfer. Then we present a method for generating interaction invariants based on the replacement of data transfer by a component called *interaction component*. Finally, we apply the methods on an example, the Bakery Protocol.

Example 35 (Bakery) *We consider 2-process Bakery Protocol [Lam74] as an example to illustrate the method presented in this Chapter. Figures 4.1 presents the model of Bakery in BIP. Two components B_1 and B_2 model two processes and they are identical up to the renaming of ports and locations. Hence we present the behavior of the component B_1, the behavior of B_2 is similar. B_1 has three locations: l_1 if B_1 is in idle section; l_2 if B_1 is waiting to enter its critical section; and l_3 if B_1 it is in its critical section. It has a variable x_1 ranging over the natural numbers and representing the "tokens" of the process, 3 loop transitions labeled by a allowing accessing the value of "token", three transitions w, e, r for moving between the locations. Initially in location l_1, B_1 can take w_1 to move to the waiting location l_2. From l_2, B_1 can enter the critical location l_3 by e_1 transition. The return to l_1 by transition r_1 resets the "token" x_1 to 0.*

Two processes B_1 and B_2 communicate by a set of interactions $\gamma = a_1w_2 + a_1e_2 + a_2w_1 + a_2e_1 + r_1 + r_2$ with the corresponding guards and functions $g_{a_1e_2} = (x_1 = 0 \vee x_2 \leq x_1)$, $g_{a_2e_1} = (x_2 = 0 \vee x_2 < x_1)$, $f_{a_1w_2} = (x_2 := x_1 + 1)$ and $f_{a_2w_1} = (x_1 := x_2 + 1)$. The function $f_{a_2w_1} = (x_1 := x_2 + 1)$ allows B_1 to get the "token" according to the "token" of B_1 while moving to waiting location l_2 by transition w_1. The guard $g_{a_2e_1} = (x_2 = 0 \vee x_2 < x_1)$ allows B_1 to move to the critical location l_3 only if B_2 is in idle location l_1 ($x_2 = 0$)

or it has higher priority "token" ($x_1 \geq x_2$). Similarly for $f_{a_1w_2} = (x_2 := x_1 + 1)$ and $g_{a_1e_2} = (x_1 = 0 \vee x_2 \leq x_1)$ for the process B_2.

An important property for Bakery system is mutual exclusion: both components can not be at the critical locations l_3 and l_6 at the same time, i.e $P = \neg(l_3 \wedge l_6)$.

4.2 Component Invariant Generation

In a system without data transfer, variables are always modified only inside their atomic components, and component invariants are computed by the *post* predicate transformer w.r.t internal transitions of these components. However, in a system with data transfer, variables can be modified by interactions between components, thus their values depend on variables of other components, and the computation of component invariants should take into account the constraints on the variables of other components. In other words, we need to project the updates on interactions into components for the computation of component invariants. We define below such projection.

Definition 34 (Interaction-to-Component Projection) *Given a connector $\gamma(B)$ where $B = (B_1, \ldots, B_n)$ with $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau \in \mathcal{T}_i}, \{f_\tau\}_{\tau \in \mathcal{T}_i})$, let $a \in \gamma$ be an interaction with guard $G_a(X)$ and function $F_a(X, X')$, let $\varphi(X)$ be a random predicate and let $\Phi(X) = \bigwedge_{X \cap X_i \neq \emptyset} \Phi_i(X_i)$ be a predicate where Φ_i is an invariant of component B_i. The projection $\mathcal{P}$ of φ over interaction a to component B_i is defined as follows:*

$$\mathcal{P}_a(\varphi)(X_i) = \exists X', \exists (X \setminus X_i).\varphi(X') \wedge \Phi(X') \wedge G_a(X') \wedge F_a(X', X)$$

where $X \setminus X_i$ means to remove X_i from X.

Starting from the weakest invariant $\Phi_i(X_i) = true$ of component B_i, we can increasingly compute tighter predicates.

Example 36 *Consider the interaction w_1a_2 of the Bakery example in Figure 4.1 with an update function $x_1 = x_2' + 1$ where variable x_1 of component B_1 is modified according to variable x_2 of component B_2. Since x_1, x_2 are natural numbers, the predicate $\Phi_2(x_2) = (x_2 \geq 0)$ (resp. $\Phi_1(x_1) = (x_1 \geq 0)$) is always true and therefore is an invariant of B_2 (resp. B_1). The predicate projection of $\varphi =$ **true** to B_1 over interaction w_1a_2 is:*

$$\mathcal{P}_{w_1a_2}(\varphi)(x_1) = \exists x_1' \exists x_2' \exists x_2 . true \wedge \Phi_1(x_1') \wedge \Phi_2(x_2') \wedge (x_1 = x_2' + 1) = (x_1 \geq 1)$$

Similarly, we have $\mathcal{P}_{w_2a_1}(\varphi)(x_2) = (x_2 \geq 1)$.

For an interaction a with a guard G_a and a function F_a, interaction a first executes the interaction function F_a, then it executes the functions of the transitions of its ports. Since the update function of a transition takes place after the update function of the interaction that its port participates, the post predicate transformer over the transition have to take into account the changes due to the interaction. Here, we extend the definition of *post* predicate transformer over a transition defined in Chapter 2 by taking into account interaction-to-component projection.

Definition 35 (Transition-based Post Predicate Transformer) *Given a component $\gamma(B)$ with $B = (B_1, \ldots, B_n)$ where $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i})$, let $\tau \in \mathcal{T}_i$ be a transition with guard $g_\tau(X_i)$ and function $f_\tau(X_i, X_i')$, and let $\gamma_i = \{a \mid a \in \gamma \wedge X_a^u \cap X_i \neq \emptyset\}$ be a set of interactions whose update variables X_a^u involve some variables in X_i. The post predicate transformer of $\varphi(X_i)$ over τ is defined as follows:*

$$post_\tau(\varphi)(X_i) = \exists X_i'.(\bigvee_{a\in\gamma_i} \mathcal{P}_a(\varphi)(X_i')) \wedge g_\tau(X_i') \wedge f_\tau(X_i', X_i)$$

Roughly speaking, we consider all the possible predicate updates by the interactions, in which the port of transition τ participates, as the predicate before executing transition τ.

Example 37 *Let us consider the post predicate computation for all non loop transitions of component B_1 in Bakery example (Figure 4.1). Initially $\varphi = true$, we have:*

- $post_{w_1}(true)(x_1) = \exists x_1'.\mathcal{P}_{w_1a_2}(true)(x_1') \wedge (x_1 = x_1') = (x_1 \geq 1)$. *The invariant at l_2 is therefore $\varphi_{l_2} = (x_1 \geq 1)$.*
- $post_{e_1}(\varphi_{l_2})(x_1) = \exists x_1'.\mathcal{P}_{e_1a_2}(\varphi_{l_2})(x_1') \wedge (x_1 = x_1')$ *where* $\mathcal{P}_{e_1a_2}(\varphi_2)(x_1) = \exists x_2.\varphi_{l_2} \wedge (x_2 \geq 0) \wedge ((x_2 = 0) \vee (x_1 \leq x_2)) = (x_1 \geq 1)$, *hence $\varphi_{l_3} = post_{e_1}(\varphi_{l_2})(x_1) = (x_1 \geq 1)$ is invariant at l_3.*
- $post_{r_1}(\varphi_{l_3})(x_1) = (x_1 = 0)$ *is invariant at l_1.*

The component invariant of B_1 is $\Phi_1 = (l_1 \wedge x_1 = 0) \vee (l_2 \wedge x_1 \geq 1) \vee (l_3 \wedge x_1 \geq 1)$. Similarly, the component invariant of B_2 is $\Phi_2 = (l_4 \wedge x_2 = 0) \vee (l_5 \wedge x_2 \geq 1) \vee (l_6 \wedge x_2 \geq 1)$.

After computing the effects to the local components caused by the updates in the interactions, we can deal with the interactions with data transfer and generate interaction components to replace the data transfer on these interactions. The following subsection will focus on the interaction component generation.

4.3 Interaction Invariant Generation

The computation of interaction invariants of a system with data transfer consists of two main steps:

- First, we need to transform the system with data transfer to a corresponding system without data transfer. In this section, we will focus on this transformation.
- Then we use the method presented in Chapter 2 to compute interaction invariants for the system without data transfer.

Based on static analysis between a set of components and a set of interactions, the method proposed in this section "replaces" the effect of data transfer on these interactions by a component called *interaction component*. Then that interaction without data is connected to the new component which allows to mimic the original behavior of the interaction. The

new components encode the guards in the interactions and the updates between the involved components into locations and transition relations. The locations in the new components enumerate all the possible combinations between the guards on the interactions. And the transition relations record the variable updates, which will be synchronized with the updates of the original components.

The choice of the set of interactions from which an interaction component is generated is based on the set of variables involved. Consider a set of variables X, we choose a set of interactions γ such that its data transfer involves in X or the update functions of its transitions involve in X. Moreover, γ should include all interactions involving in X, that is X is not affected by any interaction outside the set γ.

Given a set of interactions with data transfer, we need to compute the transition system for the interaction component and the new interactions. Therefore, it is necessary to consider the predicates updated by executing one interaction.

Definition 36 (Interaction-based Post Predicate Transformer) *Given a connector $\gamma(B)$ where $B = (B_1, \ldots, B_n)$ with $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i})$ and $X = \bigcup_{i=1}^{n} X_i$, let a be an interaction of γ, the post predicate transformer of a predicate $\varphi(X)$ over the interaction a is defined as follows:*

$$post_a(\varphi)(X) = \bigwedge_{p\in a} \left(\bigvee_{port(\tau)=p} post_\tau(\varphi)(X) \right)$$

where $port(\tau)$ is the port labeling the transition τ.

That is, the *post* predicate transformer of an interaction a represents the effect of its execution taking into account the effect by the executions of all involved transitions.

The transformation from a system with data transfer to a system without data transfer is based on the replacement of the data transfer by a component called *interaction component*. If we remove the data transfer from an interaction, we need to know when the interaction can be executed and the effect after the execution of the interaction. The execution condition of the interaction is represented by the guard which is encoded in the location of interaction component. The execution effect of the interaction is presented by the post predicate transformer *post* defined above and is encoded into transitions of the interaction component.

Definition 37 (Interaction Component) *Given $\gamma = \{a_i\}_i$ a set of interactions where $G_{a_i}(X)$ and $F_{a_i}(X, X')$ are respectively guard and function of interaction a_i, we define the interaction component $B^d = (L^d, P^d, \mathcal{T}^d)$ over γ, where:*

- *$L^d = \{\varphi^d \mid \exists\gamma' \subseteq \gamma . \varphi^d = (\bigwedge_{a\in\gamma'} G_a \wedge \bigwedge_{a\in\gamma-\gamma' \wedge G_a \neq true} \overline{G}_a) \neq false\}$ is a set of the conjunctive combinations of the sets $\{\{G_a, \overline{G}_a\}\}_{a\in\gamma}$. We define the abstraction function α^d which associates each atomic predicate $\bigwedge_{a\in\gamma'} G_a \wedge \bigwedge_{b\in\gamma-\gamma'} \overline{G}_a$ a symbol φ^d.*
- *P^d is a set of ports $\{p_a\}$ where each p_a corresponds to an interaction $a \in \gamma$.*

- $\mathcal{T}^d \subseteq L^d \times P^d \times L^d$ *where for any* $\varphi_1^d, \varphi_2^d \in L^d$, *if there exists* $a \in \gamma$ *such that* $\varphi_1^d \wedge G_a \neq false$ *and* $\varphi_2^d \wedge post_a(\varphi_1^d)(X) \neq false$, $(\varphi_1^d, p_a, \varphi_2^d) \in \mathcal{T}^d$.

During the interaction component generation, we can update the interactions by connecting with the ports of the interaction component, and generate an abstract system without data transfer.

Definition 38 (Abstract System of Data Transfer) *Given a connector* $\gamma(B)$ *and a set of interaction components* $\{B_1^d, \ldots, B_k^d\}$ *with* $B_i^d = (L^d, P^d, \mathcal{T}^d)$ *generated from the corresponding set of interactions* $\gamma_i \subseteq \gamma$, *we define the abstract system* $\gamma^d(B, B^d)$, *where* γ^d *is obtained from the following process:*

- *For any* $\tau = \{\varphi_i^d, p_a, \varphi_j^d\} \in \mathcal{T}^d$, *we generate a new interaction by adding the port* p_a *to the corresponding interaction* a: $a^d = a.p_a$, *and add* a *to the replaced interaction list:* $\gamma_r = \gamma_r + a$.

- *After no more interaction is generated, add new interactions* $\{a^d\}$ *to* γ *and remove replaced interactions in* γ^r *from* γ: $\gamma^d = \gamma + \{a^d\} - \gamma^r$.

Two definitions above provide the method to remove the data transfer in the interactions and to connect the interaction components with other components.

The process for generating an interaction component for a set of interactions is presented in Algorithm 2. It takes as input a set of interactions γ on a set of atomic components $(B_1, \ldots, B_n)$. These interactions have guards, functions or transitions involving in the same set of variables X. It basically works as follows:

- First step is to generate a set of locations according to the interaction guards. For each non-empty subset $\gamma' \subseteq \gamma$ such that the predicate $\varphi^d = \bigwedge_{a \in \gamma'} G_a \wedge \bigwedge_{a \in \gamma - \gamma' \wedge G_a \neq \mathbf{true}} \overline{G}_a$ is not **false**, we generate a location φ^d for the interaction component and add it to the location sets L^d (lines 4, 5).

- Second step is to generate the set of ports and the set of transitions. We considered any two locations φ_i^d, φ_j^d of L^d and any interaction $a \in \gamma$. If $\varphi_i^d \wedge G_a \wedge \bigwedge_{port(\tau) \in a} g_\tau \neq \mathbf{false}$ and $post_a(\varphi_i^d) \wedge \varphi_j^d \neq \mathbf{false}$ (line 10), that is there is a transition going out from φ_i^d and coming into φ_j^d and its port participates to the interaction a. If p_a does not exist in P^d, we create a port p_a (line 12), a new interaction by adding p^a to a (line 13) and a is added to the replaced interaction list which will be remove at the end (line 14). Then we generate a transition $\tau = (\varphi_i^d, p_a, \varphi_j^d)$ (line 16).

Finally, after the complete construction of the interaction component, the set of interactions is obtained by removing the replaced interaction list γ^r from γ (line 20).

The following proposition shows that the abstract system $\gamma'(B, B')$ by removing data transfer simulates $\gamma(B)$, and invariants of $\gamma'(B, B')$ are also invariants of $\gamma(B)$.

Proposition 17 *Given* $\gamma(B)$ *with data transfer on some of the interactions of* γ, *let* $B = \{B_i\}_{1 \leq i \leq n}$ *be a set of components with* $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau \in \mathcal{T}_i}, \{f_\tau\}_{\tau \in \mathcal{T}_i})$, $B^d =$

```
1  function generateInteractionComponent(γ(B_1, ..., B_n))
2  begin
3      L^d = ∅; P^d = ∅; T^d = ∅;
4      forall γ' ⊆ γ such that φ^d = ⋀_{a∈γ'} G_a ∧ ⋀_{a∈γ−γ'∧G_a≠true} Ḡ_a ≠ false do
5          L^d = L^d ∪ φ^d;
6      end
7      γ^r = ∅;        /* set of interactions to be removed */
8      forall φ_i^d, φ_j^d ∈ L^d do
9          forall a ∈ γ do
10             if (φ_i^d ∧ G_a ∧ ⋀_{port(τ)∈a} g_τ ≠ false)&(post_a(φ_i^d) ∧ φ_j^d ≠ false) then
11                 if p_a does not exist in P^d then
12                     P^d = P^d ∪ p_a;
13                     γ = γ ∪ {a.p_a};
14                     γ^r = γ^r ∪ a;
15                 end
16                 T^d = T^d ∪ {(φ_i^d, p_a, φ_j^d)};
17             end
18         end
19     end
20     γ^d = γ \ γ^r;
21 end
```

Algorithm 2: Interaction Component Generation

$\{B_j^d\}_{1\leq j\leq k}$ be a set of interaction components obtained from sets of interactions in γ with $B_j^d = (L_j^d, P_j^d, T_j^d)$, and γ^d be the corresponding set of interactions without data transfer, then $\gamma^d(B, B^d)$ simulates $\gamma(B)$. Moreover, if Φ^{α_d} is an invariant of S^d then $\alpha_d^{-1}(\Phi^{\alpha_d})$ is an invariant of S.

Proof Let (l, x) be a global state of $\gamma(B)$, where $l \in L_1 \times \cdots \times L_n$, and $(l, \varphi_1^d, \ldots, \varphi_n^d, y)$ be a global state of $\gamma'(B, B')$ where $\varphi_j^d \in L_i^d$ and $x, y \in \bigcup_{i=1}^n X_i$. We show that $(l, x)R(l, \varphi_1^d, \ldots, \varphi_n^d, y)$ is a simulation, if $v(y) \wedge \bigwedge_{j=1}^n \varphi_j^d \models v(x)$, where $v(x)$ is a valuation of x. If $(l, x) \xrightarrow{a} (l', x')$ is a transition of $\gamma(B)$, then we show that there exists $b \in \gamma'$ such that $(l, \varphi_1^d, \ldots, \varphi_n^d, y) \xrightarrow{b} (l', \varphi_1'^d, \ldots, \varphi_n'^d, y')$ where $a \in b$ and $(l', \varphi_1'^d, \ldots, \varphi_n'^d, y') = (l', \varphi_1^d, \ldots, \varphi_n^d, y')$.

- If a contains no data transfer, we have $a = b$ and $(l', \varphi_1^d, \ldots, \varphi_n^d, y')$. Because no variable is updated on a, $(l', x')R(l', \varphi_1^d, \ldots, \varphi_n^d, y')$.
- If a contains some data transfer, there exists B_j' and $a \cup \{p_j\} \subset b \in \gamma'$, and $\varphi_j^d \wedge G_a \neq false \wedge \varphi_j'^d \wedge post_a(\varphi_l) \neq false$ such that $(l, \varphi_1^d, \ldots, \varphi_n^d, y) \xrightarrow{b} (l', \varphi_1^d, \ldots, \varphi_{j-1}^d, \varphi_j'^d, \varphi_{j+1}^d, \ldots, \varphi_n^d, y')$. Suppose z is updated in interaction a, we have $post_a(\varphi_l)(z) \models v(z)$. In interaction b, the transition from B_j' requires that $\varphi_j'^d \wedge post_a(\varphi_l) \neq false$. For other variables, they are updated by transitions. So we have $v(y) \wedge \bigwedge_{j=1}^n \varphi_j'^d \models v(x')$.

Example 38 *For Bakery example, since the data transfer of all the interactions involve in the set of variables $\{x_1, x_2\}$ we will generate an interaction component $B_3 = (L^d, P^d, \mathcal{T}^d)$ to replace the data transfer on these interactions.*

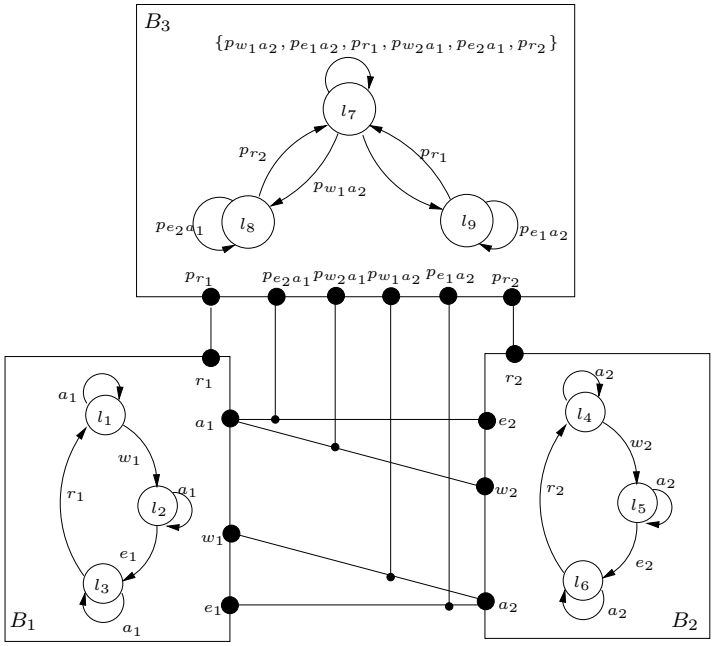


Figure 4.2: Bakery example with interaction component

First we generate the set of locations from the guards of the set of interactions $\{g_{a_1e_2}, g_{a_2e_1}\}$*:*
$g_{a_1e_2} \wedge g_{a_2e_1} = (x_1 = 0 \vee x_2 < x_1) \wedge (x_2 = 0 \vee x_1 \leq x_2) = (x_1 = 0 \vee x_2 = 0)$
$g_{a_1e_2} \wedge \bar{g}_{a_2e_1} = (x_1 = 0 \vee x_2 < x_1) \wedge (x_2 > 0) \wedge (x_1 > x_2) = (x_2 > 0 \wedge x_1 > x_2)$
$\bar{g}_{a_1e_2} \wedge g_{a_1e_2} = (x_1 > 0) \wedge (x_2 \geq x_1) \wedge (x_2 = 0 \vee x_1 \leq x_2) = (x_1 > 0 \wedge x_2 \geq x_1)$

Since all the combinations are different from false, we generate a set of three corresponding locations $L^d = \{l_7, l_8, l_9\}$ *where:*

$$l_7 = g_{a_1e_2} \wedge g_{a_2e_1} = (x_1 = 0 \vee x_2 = 0)$$
$$l_8 = g_{a_1e_2} \wedge \bar{g}_{a_2e_1} = (x_1 > 0 \wedge x_1 < x_2)$$
$$l_9 = g_{a_2e_1} \wedge \bar{g}_{a_1e_2} = (x_2 > 0 \wedge x_2 < x_1)$$

The ports and the transition system of the interaction component B_3 *is presented in the figure 4.2. They are specified from the set of locations* L^d *and the set of interactions* γ*. For example, there is a port* $p_{a_1w_2}$ *together with one of its transition* $\tau = (l_7, p_{a_2w_1}, l_8)$ *since* $l_7 \wedge g_{w_1a_2} = l_7 \neq \textbf{\textit{false}}$ *and* $post_{a_2w_1}(l_7)(x_1) = post_{a_2}(l_7) \wedge post_{w_1}(l_7) = \exists x'_1 \exists x'_2.(x'_1 = 0 \vee x'_2 = 0) \wedge (x_1 = x'_2 + 1) = (x_1 > 0)$*, hence* $post_{a_2w_1}(l_7) \wedge l_8 = (x_1 > 0) \wedge (x_1 > 0 \vee x_1 < x_2) \neq \textbf{\textit{false}}$*.*

Example 39 (Checking Mutual Exclusion) *Mutual exclusion is an important property of Bakery example: two components can not be at the critical locations* l_3 *and* l_6 *at the same time. The property is formally represented as* $P = \neg(l_3 \wedge l_6)$*.*

First we compute the component invariants of the system $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$ *where* $\Phi_1 = (l_1 \wedge x_1 = 0) \vee (l_2 \wedge x_1 > 0) \vee (l_3 \wedge x_1 > 0)$*,*

$\Phi_2 = (l_4 \wedge x_2 = 0) \vee (l_5 \wedge x_2 > 0) \vee (l_6 \wedge x_2 > 0)$ *and*
$\Phi_3 = l_7 \vee l_8 \vee l_9$ *with* $l_7 = (x_1 = 0 \vee x_2 = 0)$, $l_8 = (x_2 > 0 \wedge x_1 > x_2)$ *and* $l_9 = (x_1 > 0 \wedge x_2 \geq x_1)$.

From the Bakery abstract system computed in Example 38, we compute the abstract interaction invariants and then the concrete interaction invariants that are respectively:

$$\begin{aligned}\Psi^d = &(l_5 \vee l_7 \vee l_8)\\ &\wedge (l_2 \vee l_7 \vee l_9)\\ &\wedge (l_2 \vee l_3 \vee l_5 \vee l_6 \vee l_7)\end{aligned}$$

$$\begin{aligned}\Psi = &((l_5 \wedge x_2 > 0) \vee (x_1 = 0 \vee x_2 = 0) \vee (x_2 > 0 \wedge x_1 > x_2))\\ &\wedge ((l_2 \wedge x_1 > 0) \vee (x_1 = 0 \vee x_2 = 0) \vee (x_1 > 0 \wedge x_2 \geq x_1))\\ &\wedge ((l_2 \wedge x_1 > 0) \vee (l_3 \wedge x_1 > 0) \vee (l_5 \wedge x_2 > 0) \vee (l_6 \wedge x_2 > 0) \vee (x_1 = 0 \vee x_2 = 0))\end{aligned}$$

Finally we verify the mutual exclusion property P by using Yices to check $\Phi \wedge \Psi \wedge \neg P$. The unsat output of Yices shows that the property is guaranteed for the Bakery example.

4.4 Summary

We have presented a method for dealing with interaction models with data transfer. The idea is that we first project the changes of variables by interactions into transitions, then transform the models with data transfer into the models without data transfer on which we can apply our compositional method for checking safety properties. The transformation is done by replacing the data transfer by interaction components which allow preserving the behavior of the interactions. We have also applied the method for verifying mutual exclusion property of the Bakery example.

Although this method has not been implemented in our tool-set but the obtained result shows the perspectives of the method in dealing with interaction models with data transfer.

Part III

Implementation, Tools and Case Studies

CHAPTER 5

Implementation

Contents

5.1 The D-Finder Tool

We have implemented the compositional and incremental methods in D-Finder, a tool for verifying safety properties, specially for checking deadlock-freedom for component-based systems described in the BIP language. D-Finder consists of a set of modules interconnected as shown in Figure 5.1. It takes as input a system described in BIP and progressively finds and eliminates potential deadlocks. It basically works as follows:

1. It constructs the predicate characterizing the set of deadlock states (DIS generation module).

2. Iteratively, it constructs increasingly stronger component invariants (Φ_i generation module). This step might need quantifier elimination that requires collaboration with Omega tool.

3. Component invariants are used to compute finer finite state abstractions and increasingly stronger interaction invariants (Abstraction and Ψ generation module). The computation of interaction invariants is done by collaborating with CUDD package or Sat-solver tool Yices.

4. It verifies deadlock freedom by checking the unsatisfiability of $\bigwedge \Phi_i \wedge \Psi \wedge DIS$ (satisfiability module). If it succeeds, the system is proven deadlock-free, else it may continue or give up, according to the user's choice. The unsatisfiability is checked by a Sat-Solver tool Yices in the case of systems with data and by CUDD package in the case of systems without data.

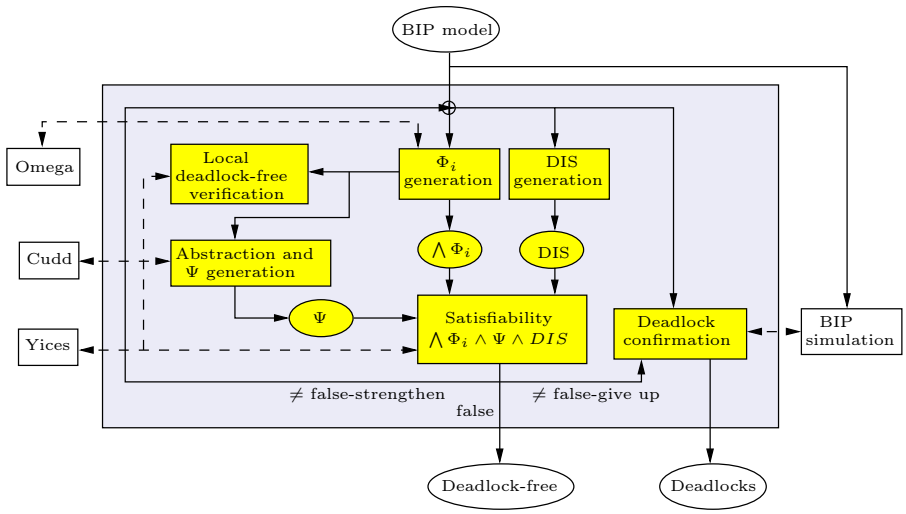


Figure 5.1: The D-Finder tool

It is also connected to the state space exploration tool of the BIP platform, for finer analysis when the heuristic fails to prove deadlock-freedom.

The main programming language used in the implementation of D-Finder is Java. However, since CUDD package is written in C, several parts which are connected to CUDD are also written in C.

We provide below in detail the description of each module in D-Finder.

5.2 DIS Generation

The implementation of DIS generation module is presented in Algorithm 3. The input is a set of interactions γ on a set of components B. The output is a predicate DIS characterizing deadlock states, i.e a set of states from which no interaction of γ can take place. The DIS generation process works as follows: first we compute the enabled condition en_a for each interaction a of γ by computing the enabled conditions for all its ports. A port p is enabled if at least one of its transitions is enabled and a transition $\tau = (l, p, g_\tau, f_\tau, l')$ is enabled if

its source location is reached and its guard is true (line 6). Hence the enabled condition of p is the disjunction of the enabled conditions of all its transitions (line 8). The interaction a is enabled if its guard G_a is true and all its ports are enabled (line 10). Finally, the predicate DIS is obtained by the conjunction of the disabled conditions (the negation of enabled conditions) of all the interactions in γ (line 12).

```
1  function generateDIS(Connector γ, Component B)
2  begin
3      for each interaction a of γ do
4          for each port p of a do
5              forall transitions τ = (l, p, g_τ, f_τ, l') do
6                  en_τ = l ∧ g_τ;
7              end
8              en_p = ⋁_{port(τ)=p} en_τ;
9          end
10         en_a = G_a ∧ ⋀_{p∈a} en_p;
11     end
12     DIS = ⋀_{a∈γ}(¬en_a);
13     return DIS;
14 end
```

Algorithm 3: DIS Predicate Generation

5.3 Component Invariant Generation

The implementation of the Component Invariant Generation module is shown in Algorithm 4. Taking as input an atomic component, we generate the component invariants by computing an invariant predicate, which is initially true, at each control location. Consider a location l, we first compute post predicate transformers of all its incoming transitions by calling *computePost* function (line 5) which returns, for a transition $\tau = (l', p, g_\tau, f_\tau, l)$ and an invariant predicate $\Phi_{l'}$ at the source location l', the propagation of $\Phi_{l'}$ by τ. Then the invariant predicate at l is obtained by the disjunction of all these post predicate transformers (line 7).

```
1  function computeCompInv(Component B)
2  begin
3      for each control location l of B do
4          for each transition τ = {l', p, g_τ, f_τ, l} do
5              post_τ = computePost(τ, Φ_{l'});
6          end
7          Φ_l = ⋁_{τ∈•l} post_τ;
8          return Φ_l;
9      end
10 end
```

Algorithm 4: Compute Component Invariants

```
1  function computePost(Transition τ, Predicate Φ)
   /* τ is of the form (l', p, g_τ(X), Y = f_τ(Z), l)                    */;
   /* Φ is a predicate on the sets of variables X, Y, Z                  */;
2  begin
3      post = true;
4      if g_τ = null then
5          g_τ = true;
6      end
7      if f_τ = null then
8          post = g_τ(X) ∧ Φ(X, Y, Z);
9          return post;
10     end
11     if X ∩ Y = ∅ then
12         post = post ∧ g_τ;
13     end
14     if Y ∩ Z = ∅ then
15         post = post ∧ (Y = f_τ(Z));
16     else
17         f = ∃X', Y', Z'.Φ(X', Y', Z') ∧ g_τ(X') ∧ (Y = fτ(Z'));
18         f = callOmega(f);
19         post = post ∧ f;
20     end
21     return post
22 end
```

Algorithm 5: Compute Post Predicate

An important function in generating component invariants is *computePost*, presented in Algorithm 5, which computes the post predicate transformer of a predicate with respect to a transition. It takes as inputs a transition τ of the form $\tau = (l', p, g_\tau(X), Y = f_\tau(Z), l)$, a predicate Φ at the source location l'. X, Y, Z are subsets of component's variable set. The function works as follows:

- If the guard g_τ is **null**, the execution condition of τ is always true, hence g_τ is assigned to **true** (lines 4, 5).
- If the update function f_τ is **null**, that is the set of variables is not affected by the transition and therefore Φ still holds after the transition, the function returns $g_\tau \wedge \Phi(X, Y, Z)$ (lines 7, 8, 9) and terminates.
- If $X \cap Y = \emptyset$ which means that the variables in the guard g_τ are not affected by the function, the guard g_τ still holds after the transition. Hence, the post predicate transformer is updated by conjuncting with g_τ: $post = post \wedge g_\tau$ (lines 11, 12).
- If $Y \cap Z = \emptyset$, the variables in the right side of the update function are not changed and the predicate $Y = f_\tau(X)$ holds after the transition. The post predicate transformer is updated: $post = post \wedge (Y = f_\tau(Z))$ (line 15).
- If $X \cap Y \neq \emptyset$, the variables in the right side of the update function are affected. The new valuation is computed by taking into account the affect of the update function:

there exists some valuation of X, Y, Z at the source location l' (denoted by X', Y', Z') such that the predicate $\Phi(X', Y', Z')$ is true, the guard $g_\tau(X')$ is true (for that the transition τ can be executed) and the new valuation is computed from the existing valuation according to the update function $Y = f_\tau(Z')$. We call the external tool Omega to eliminate the quantifiers (line 18).

The function *callOmega(f)* writes the formula f to the input file of Omega. Then it calls Omega to eliminate quantifier and get back the corresponding quantifier-free formula that we might need to convert by a parser to get the expected form. An example of the input file for the post predicate transformer of a predicate $\Phi = (x \geq 0)$ by a transition τ with $g_\tau = (x < 10)$, $f_\tau = (x := x + 1)$ is as follows:

$$F := \{[x] : exists(tmp : (tmp \geq 0)\ \&\&\ (tmp < 10)\ \&\&\ (x = tmp + 1))\}$$

The output of Omega for this entry is the quantifier-free formula $1 \leq x \leq 10$.

5.4 Checking Local Deadlock-Freedom

An assumption of our method for checking deadlock-freedom of a system is that every component is deadlock-free. Hence it is necessary to check the local deadlock-freedom of each atomic component before checking the global deadlock-freedom. An atomic component is deadlock-free if at any location, it is always able to move by taking one of its outgoing transitions. The function for checking local deadlock-freedom takes as input an atomic component together with its component invariants. At each location l, the function verifies whether its invariant Φ_l implies at least one of its outgoing transitions' guard, i.e $\Phi_l \Rightarrow \bigvee_{\tau \in l\bullet} g_\tau$. The satisfiability of this condition is checked by Yices tool. If it holds at all the locations of the component, then the function returns *deadlock-free*. Otherwise, it returns *not deadlock-free* output together with a set of locations that do not satisfy this condition.

5.5 Abstraction

Abstraction is used to transform a system with data into an equivalent system without data. The implementation of abstraction process in the D-Finder tool is presented in Algorithm 6. It takes as input a system consisting of a set of components with their component invariants and a set of interactions. The abstraction process consists of three main steps corresponding to the generations of abstract components, of abstract interactions and of abstract initial condition.

For the generation of an abstract component B^α from a concrete component $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ and its component invariants $\Phi = \bigvee_{l_j \in L}(l_j \wedge \bigvee_k \varphi_{jk})$, the abstract function works as follows:

- First, the set of abstract locations L^α are generated by splitting concrete locations according to the invariants. For each location l_j where the invariant is of the form $\bigvee_k \varphi_{jk}$, we create for each predicate φ_{jk} an abstract location $l^\alpha_{jk} = l_j \wedge \varphi_{jk}$ (lines 8, 9).

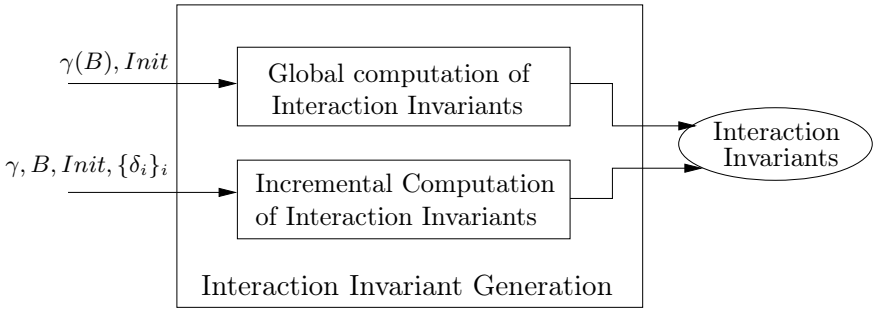


Figure 5.2: Structure of Interaction Invariant Generation Module

- Then, the set of abstract transitions $\mathcal{T}^\alpha$ are generated. For each concrete transition $\tau = (l_m, p, g_\tau, f_\tau, l_n)$, we have two sets of abstract source locations l^a_{mj} and of abstract destination locations l^α_{nk}. An abstract transition $\tau^a_{mjnk} = (l^\alpha_{mj}, p, l^\alpha_{nk})$ is established (line 15) only if the conditions $\varphi_{mj} \wedge g_\tau \neq \textbf{false}$ and $post_\tau(\varphi_{mj}) \wedge \varphi_{nk} \neq \textbf{false}$ (or equivalently $\varphi_{mj} \wedge pre_\tau(\varphi_{nk}) \neq \textbf{false}$) hold (line 14).
- The generation of abstract ports P^α is just the creation of an abstract port p^α for each concrete port p (lines 21, 22).

The generation of abstract interactions is done by creating, for each concrete interaction $\gamma_i = p_1 \dots p_n$, a new interaction composed of the corresponding abstract ports $\gamma^\alpha_i = p^\alpha_1 \dots p^\alpha_n$ (lines 29, 30).

The generation of initial conditions is to take, for each element $l_j \wedge \varphi^{init}_j$ of the set $Init$, all abstract locations $l^\alpha_{jk} = l_j \wedge \varphi_{jk}$ of l_j such that $\varphi^{init}_j \wedge \varphi_{jk} \neq \textbf{false}$ (lines 36, 37).

5.6 Interaction Invariant Generation

We have implemented two sub-modules according to global and incremental computation of interaction invariants as presented in Figure 5.2:

- global computation sub-module takes as input a global system $\langle \gamma(B), Init \rangle$ and computes globally interaction invariants of the system.
- incremental computation sub-module takes as input a connector γ over a set of components B with the initial conditions $Init$, and a set of increments $\{\delta_i\}^n_{i=1}$ and computes incrementally the set interaction invariants.

5.6.1 Global Computation of Interaction Invariants

Figure 5.3 shows the structure and the data flow for the global computation of interaction invariants. There are several sub-methods for the computation: two enumerative methods using Yices and CUDD; two symbolic methods based on positive mapping and fixed-point using CUDD package. The implementation consists of the following functions:

```
1  function abstract(γ(B_1,...,B_n),(Φ_1,...,Φ_n),Init)
2  begin
3      /* Generate abstract components */
4      for each component B_i do
5          L^α = ∅; P^α = ∅; T^α = ∅;
6          for each location l_j of B_i with Φ_{l_j} = l ∧ ⋁_k φ_jk do
7              for each predicate φ_jk do
8                  l^α_jk = l_j ∧ φ_jk;
9                  L^α = L^α ∪ l^α_jk;
10             end
11         end
12         for each transition τ = (l_m, p, g_τ, f_τ, l_n) do
13             forall l^α_mj and l^α_nk do
14                 if (φ_mj ∧ g_τ ≠ false) ∧ (post_τ(φ_mj) ∧ φ_nk ≠ false) then
15                     τ^α_mjnk = (l^α_mj, p, l^α_nk);
16                     T^α = T^α ∪ τ^α_mjnk;
17                 end
18             end
19         end
20         for each port p_j do
21             create abstract port p^α_j;
22             P^α = P^α ∪ p^α_j;
23         end
24         B^α_i = (L^α, P^α, T^α);
25     end
26     /* Generate abstract interactions */
27     γ^α = ∅;
28     for each interaction γ_i = p_1 ... p_n of γ do
29         create γ^α_i = p^α_1 ... p^α_n;
30         γ^α = γ^α ∪ γ^α_i;
31     end
32     /* Generate abstract set of initial locations */
33     Init^a = ∅;
34     for each l_j ∧ φ^init_j ∈ Init do
35         for each l^a_jk = l_j ∧ φ_jk do
36             if φ^init_j ∧ φ_jk ≠ false then
37                 Init^α = Init^α ∪ l^a_jk;
38             end
39         end
40     end
41     return ⟨γ^α(B^α_1,...,B^α_n), Init^α⟩;
42 end
```

Algorithm 6: Abstraction

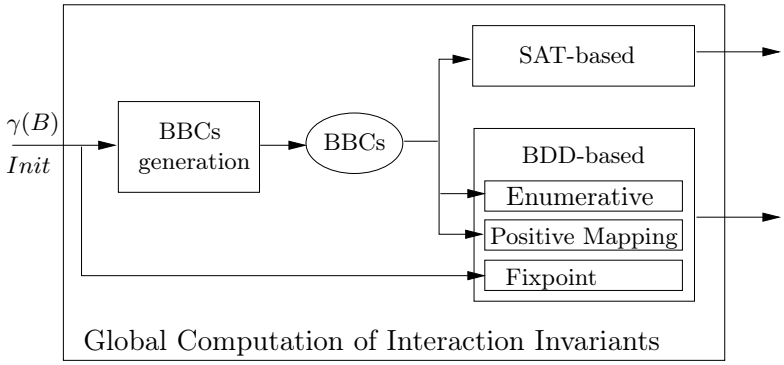


Figure 5.3: Global Computation of Interaction Invariants Module

- A function for generating Boolean Behavioral Constraints (BBCs) from the set of interactions and components.
- Different functions based on different methods for computing interaction invariants:
  - two functions based on the enumerative method which computes explicitly the set of interaction invariants. They use the Sat-Solver tool Yices or CUDD package to solve equation systems.
  - two functions based on the symbolic methods which compute the set of interaction invariants by using Positive Mapping or Fix-point. The computation is performed using CUDD package.

Below we present in detail the implementation of each function.

Boolean Behavioral Constraints (BBCs) generation

Boolean Behavioral Constraints can be generated according to Definition 17: for any location l, we need to build an implication corresponding to interactions that its outgoing transitions are involved. However, the right sides of the implications in BBCs are different even for the same interaction. For example, consider the interaction $p_1p_2p_3$ in figure 5.4, the transitions τ_1 and τ_2 are involved in the same interaction but the corresponding implications $l_1 \Rightarrow (l'_1 \vee l'_3 \vee l'_5) \wedge (l'_1 \vee l'_4 \vee l'_5)$ and $l_2 \Rightarrow (l'_2 \vee l'_3 \vee l'_5) \wedge (l'_2 \vee l'_4 \vee l'_5)$ have different right sides. Therefore we have to visit an interaction many times for different implications and the generation of BBCs is not efficient.

These two implications can be rewritten in the form $l_1 \Rightarrow l'_1 \vee (l'_3 \vee l'_5) \wedge (l'_4 \vee l'_5)$ and $l_2 \Rightarrow l'_2 \vee (l'_3 \vee l'_5) \wedge (l'_4 \vee l'_5)$ or equivalently $l_1 \Rightarrow l'_1 \vee (l'_3 \wedge l'_4) \vee l'_5$ and $l_2 \Rightarrow l'_2 \vee (l'_3 \wedge l'_4) \vee l'_5$ from which we can see that for two transitions τ_1, τ_2 of the same component, the parts of other components in the implications are the same $(l'_3 \wedge l'_4) \vee l'_5$. Hence, in the implementation, we generate a predicate once for each interaction and then reuse that predicate in generating BBCs corresponding to that interaction.

We recall that for a port p, $p^\bullet$ is a set of destination locations of its transitions. For example, in figure 5.4, $p_1^\bullet = \{l'_1, l'_2\}$, $p_2^\bullet = \{l'_3, l'_4\}$, $p_3^\bullet = \{l'_5\}$.

We define below a predicate called *Forward Predicate* for an interaction. This predicate is generated once and then will be reused in generating implications corresponding to the interaction.

Definition 39 (Forward Predicate of Interaction) *Let $a = p_1 p_2 \dots p_n$ be an interaction, we define its forward location predicate as follows:*

$$\overrightarrow{a} = \bigvee_{p_i \in a} \bigwedge_{l \in p_i^\bullet} l$$

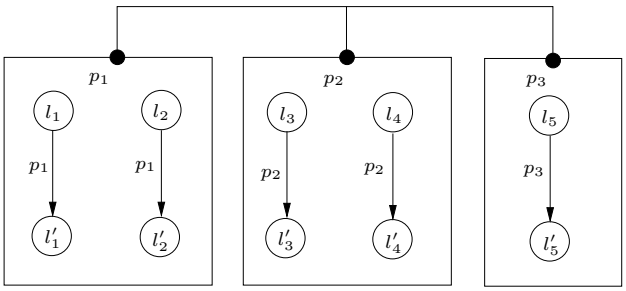


Figure 5.4: An example for BBCs

Example 40 *For the interaction $a = p_1 p_2 p_3$ in Figure 5.4, the set of destination locations of each component is respectively $\{l'_1, l'_2\}$, $\{l'_3, l'_4\}$ and $\{l_5\}$. Then we have the forward location predicate of the interaction a is $\overrightarrow{a} = (l'_1 \wedge l'_2) \vee (l'_3 \wedge l'_4) \vee l'_5$.*

Definition 40 (Forward Location Predicate) *Given a location l and a set of interactions γ, we define the forward location predicate of l with respect to γ as follows:*

$$\overrightarrow{l}^\gamma = \bigwedge_{\tau \in l^\bullet \wedge l' \in \tau^\bullet} (l' \vee \bigwedge_{port(\tau) \in a \wedge a \in \gamma} \overrightarrow{a})$$

where $port(\tau)$ is the port labeling the transition τ.

Example 41 *Consider again the interaction $a = p_1 p_2 p_3$ in Figure 5.4 with the forward predicate $\overrightarrow{a}$ in Example 40. The forward location predicate of l_1 corresponding to the transition τ_1 can be obtained from $\overrightarrow{a}$ by using Definition 40 as follows:*

$$\overrightarrow{l_1}^a = l'_1 \vee \overrightarrow{a} = l'_1 \vee (l'_1 \wedge l'_2) \vee (l'_3 \wedge l'_4) \vee l'_5 = l'_1 \vee (l'_3 \wedge l'_4) \vee l'_5$$

We can see that $\overrightarrow{l_1}^a$ is the right side of the implication from l_1 according to the interaction a.

The following proposition shows that for a location l and an involved interaction a, its Forward Location Predicate $\overrightarrow{l}^a$ is the right side of the implication according to a. Therefore we can save the computation effort by generating once, for every interaction a, the Forward Predicate of Interaction $\overrightarrow{a}$ and then reuse them for generating the implications of involved locations.

Proposition 18 *Let γ be a connector over a tuple of components $B = (B_1, \ldots, B_n)$. The Boolean Behavioral Constraints for a connector $\gamma(B)$ with a set of locations L of B can be computed as follows:*

$$|\gamma(B)| = \bigwedge_{l \in L} (l \Rightarrow \overrightarrow{l}^{\gamma})$$

Proof Given an interaction $a = p_1 p_2 \ldots p_n \in \gamma$ with the corresponding set of destination locations for port p_i is $L_i = p_i^\bullet = \{l_{i1}, \ldots, l_{im_i}\}$, we first prove that $\bigwedge_{\phi \in L_1 \times \cdots \times L_n} \bigvee_{l \in \phi} l = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{ij}$ by induction technique:

- For $n = 2$, we have:

$$\begin{aligned}
\bigwedge_{\phi \in L_1 \times L_2} \bigvee_{l \in \phi} l &= [(l_{11} \vee l_{21}) \wedge \cdots \wedge (l_{11} \vee l_{2m_2})] \\
&\wedge \ \ldots \\
&\wedge \ [(l_{m_1 1} \vee l_{21}) \wedge \cdots \wedge (l_{m_1 1} \vee l_{2m_2})] \\
&= [l_{11} \vee (l_{21} \wedge \cdots \wedge l_{2m_2})] \wedge \cdots \wedge [l_{12} \vee (l_{21} \wedge \cdots \wedge l_{2m_2})] \\
&= (l_{11} \wedge \cdots \wedge l_{1m_1}) \vee (l_{21} \wedge \cdots \wedge l_{2m_2})] \\
&= \bigvee_{i=1}^{2} \bigwedge_{j=1}^{m_i} l_{ij}
\end{aligned}$$

- suppose that for $n = k$ we have $\bigwedge_{\phi \in L_1 \times \cdots \times L_n} \phi = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{ij}$, we will prove it is also true for $n = k + 1$:

$$\begin{aligned}
\bigwedge_{\phi \in L_1 \times \cdots \times L_{k+1}} \bigvee_{l \in \phi} l &= \bigwedge_{\phi \in (L_1 \times \cdots \times L_k) \times L_{k+1}} \bigvee_{l \in \phi} l \\
&= (\bigvee_{i=1}^{k} \bigwedge_{j=1}^{m_i} l_{ij} \vee l_{(k+1)1}) \wedge \cdots \wedge (\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{ij} \vee l_{(k+1)m_{k+1}}) \\
&= (\bigvee_{i=1}^{k} \bigwedge_{j=1}^{m_i} l_{ij} \vee (l_{(k+1)1} \wedge \cdots \wedge l_{(k+1)m_{k+1}}) \\
&= \bigvee_{i=1}^{k+1} \bigwedge_{j=1}^{m_i} l_{ij}
\end{aligned}$$

According to Definition 17, the the right side of the implication for transition $\tau = (l'_{kt}, p_k, l_{kt})$ is as follows:
$\bigwedge_{\phi \in L_1 \times \cdots \times L_{k-1} \times L_{k+1} \times \cdots \times L_n} (l_{kt} \vee \bigvee_{l \in \phi} l) = l_{kt} \vee (\bigwedge_{\phi \in L_1 \times \cdots \times L_{k-1} \times L_{k+1} \times \cdots \times L_n} \bigvee_{l \in \phi} l)$. On the other hand, by using Forward Predicate of Interaction $\overrightarrow{a}$, we have $l_{kt} \vee \overrightarrow{a} = l_{kt} \vee \bigwedge_{\phi \in L_1 \times \cdots \times L_n} \bigvee_{l \in \phi} l = l_{kt} \vee \bigwedge_{\phi \in L_1 \times \cdots \times L_{k-1} \times L_{k+1} \times \cdots \times L_n} \bigvee_{l \in \phi} l$.
So the implication for transition τ by interaction a can be represented by $l'_{kt} \Rightarrow l_{kt} \vee \overrightarrow{a}$. Hence, we have:

$$|\gamma(B)| = \bigwedge_{l \in L} \left(l \Rightarrow \bigwedge_{\tau \in l^\bullet \wedge l' \in \tau^\bullet} (l' \vee \bigwedge_{port(\tau) \in a \wedge a \in \gamma} \overrightarrow{a}) \right) = \bigwedge_{l \in L} (l \Rightarrow \overrightarrow{l}^{\gamma})$$

$\overrightarrow{l}^{\gamma}$ can be used for generating image vector in the implementation of the method based on fixed-point computation which will be presented later.

The implementation for generating Forward Location Predicate of a location l according to a set of interaction γ on a set of components B is presented in Algorithm 7: for every outgoing transition $\tau = (l, p, l')$ of l, we make the conjunction of the Forward Predicates of all the involved interactions (that is the interactions contain p) and assign the result to a temporary variable tmp (line 5); then $\overrightarrow{l}^{\gamma}$ is updated by the conjunction with $l' \vee tmp$ (line 6); finally $\overrightarrow{l}^{\gamma}$, the Forward Location Predicate of l according to γ, is returned (line 8).

```
1 function generateForwardLocationPredicate(l, γ, B)
2 begin
3     l→^γ = true;
4     for each transition τ = (l, p, l') do
5         tmp = ⋀_{p∈a∧a∈γ} a→;
6         l→^γ = l→^γ ∧ (l' ∨ tmp);
7     end
8     return l→^γ;
9 end
```

Algorithm 7: Generate Forward Location Predicate

The detail implementation of the function for BBCs generation is presented in Algorithm 8:

```
1 function generateBBC(γ, B)
2 begin
3     L = ⋃_{B_i=(L_i,P_i,T_i)∈B} L_i;
4     for each location l of L do
5         l→^γ = generateForwardLocationPredicate(l, γ, B);
6     end
7     bbc = ⋀_{l∈L}(l ⇒ l→^γ);
8     return bbc
9 end
```

Algorithm 8: BBCs Generation

- The input is a set of interactions γ on a set of components B.
- First we get the union L of locations of components in B (line 3) and for each location l of L, we compute the Forward Location Predicate $\overrightarrow{l}^{\gamma}$ by calling *generateForwardLocationPredicate* function (line 5).
- Then, $BBCs$ is computed by the conjunction of the implications $l \Rightarrow \overrightarrow{l}^{\gamma}$ of all locations l of L (line 7). The function terminates by returning $BBCs$, the Boolean Behavioral Constraints of $\gamma(B)$.

Implementation of Enumerative Method

We have implemented two functions for the enumerative method: a function using SMT Sat-Solver tool Yices and a function using CUDD package. The algorithms for the implementations of two functions are similar and are presented in Algorithm 9. Taking as input the *BBCs* and the set of initial locations *Init*, the computation process works as follows:

- First, we get a formula f by the conjunction of *BBCs* with the disjunction of initials locations in *Init* (line 3) to guarantee that every interaction invariant contains at least an initial location.

- Then, we compute solutions of f. For every solution $s = \bigwedge_i l_i \wedge \bigwedge_j \overline{l_j}$, we extract all the positive valuations l_i in s to generate the corresponding interaction invariant and then add it to Ψ (line 6). We want to avoid getting others solutions of f such that their positive valuation sets are superset of that set of s because they correspond to weaker invariants, hence we update f by adding $\bigvee_i \overline{l_i}$ before getting another solution (line 7). This step is repeated until f is unsatisfiable.

- Finally, the function returns Ψ, the set of interaction invariants.

```
1  function computeEmumerativeII(BBCs, Init)
2  begin
3      f = BBCs ∧ ⋁_{l_i ∈ Init} l_i;
4      Ψ = true;
5      while f has a solution s = ⋀ l_i ∧ ⋀ l̄_j do
6          Ψ = Ψ ∧ ⋁ l_i;
7          f = f ∧ (∨ l̄_i);
8      end
9      return Ψ;
10 end
```

Algorithm 9: Enumerative Computation of Interaction Invariants

The enumerative method provides a clear, visual view of interaction invariants of the system. However, there is a risk of explosion of solutions, if exhaustiveness of solutions is necessary in the analyzing process.

Implementation of the Symbolic Method Based on Positive Mapping

In contract to the enumerative methods which extract positive variables from solutions of BBCs, the method using positive mapping symbolically removes the negative variables from BBCs. The implementation of the method is presented in Algorithm 10. Taking as input BBCs and the set of initial locations *Init*, the computation process works as follows:

- First, the disjunction of initial conditions is conjuncted with BBCs (line 3) to make sure that every invariant contains at least an initial condition.

- Then, the positive mapping function *positiveMapping* is called (line 4) which eliminates all the negative forms of the variables in BBCs. The returned result is assigned to f.

- Finally, the dual function is called for f (line 5) and the result is assigned to Ψ which is the symbolic interaction invariants.

```
1 function computeIIByPositiveMapping(bbcs, Init)
2 begin
3     f = bbcs ∧ ⋁_{l_i ∈ Init} l_i;
4     f = positiveMapping(f);
5     Ψ = dual(f);
6     return Ψ;
7 end
```

Algorithm 10: Interaction Invariant Computation based on Positive Mapping

The implementation of *positiveMapping* function is presented in Algorithm 11: given a function $f(X)$ and a subset of variables $Y \subseteq X$, this function eliminates all the negative forms of variables belonging to $X \backslash Y$ by using $cofactor$ function provided in CUDD package. If the subset Y is not provided, this function removes the negative forms of all the variables of X.

```
1 function positiveMapping(f(X), Y ⊆ X)
2 begin
3     for each variable y of X \ Y do
4         f' = cofactor(f, ȳ);
5         f = f' ∨ f;
6     end
7     return f;
8 end
```

Algorithm 11: Positive Mapping Function

The algorithm for dual function is presented in Algorithm 12. Given a symbolic formula f and a set of variables $X = \{x_1, \ldots, x_n\}$ in f, dual function first creates a set of variables $X' = \{x'_1, \ldots, x'_n\}$ such that $x'_i = \bar{x}_i$ (lines 3, 4, 5). Then, it replaces each variable $x_i \in X$ in f by $x'_i \in X'$ and obtains f' (line 7). This is done by *vectorCompose(f, X')* function provided in CUDD which creates a new BDD by substituting the BDDs for the variables of the BDD f. The negative form of f' is the dual of f (line 8).

Example 42 *Consider a formula $f = x_1x_2 + x_2x_3$ and its set of variables $X = \{x_1, x_2, x_3\}$. The dual function first creates a set $X' = \{x'_1, x'_2, x'_3\}$ where $x'_1 = \bar{x}_1, x'_2 = \bar{x}_2, x'_3 = \bar{x}_3$. Then it calls vectorCompose(f, X') function which replaces variables in X by the corresponding variables in X' and returns $f' = \bar{x}_1\bar{x}_2 + \bar{x}_2\bar{x}_3$. Finally it calls cudd_Neg(f') function which returns $\tilde{f} = (x_1 + x_2)(x_2 + x_3)$, the dual of f.*

```
1  function dual(f(X = {x_1, ..., x_n}))
2  begin
3      creates X' = {x'_1, ..., x'_n};
4      for each variable x_i ∈ X do
5          x'_i = x̄_i;
6      end
7      f' = vectorCompose(f, X');
8      f̃ = cudd_Neg(f');
9      return f̃;
10 end
```

Algorithm 12: Dual operation

Implementation of the Symbolic Method Based on Fixed-point

An alternative symbolic method for computing interaction invariants is based on fixed-point computation. Taking as input a set of interactions γ over B and a set of initial locations $Init$, the implementation of the method based on fixed-point is presented in Algorithm 13:

```
1  function computeIIByFixedpoint(γ, B, Init)
2  begin
3      𝕍 = computeImageVector(γ, B);
4      f = ⋁_{l_i ∈ Init} l_i;
5      f = computeFixedpoint(𝕍, f);
6      Ψ = dual(f);
7      return Ψ;
8  end
```

Algorithm 13: Fixed-point-based Computation of Interaction Invariants

- First, we compute functional vector $\mathbb{V}$ for all location variables by calling *computeImageVector* function (line 3).
- Then, we compute the fixed-points of the formula f which is initialized by the disjunction of initial location variables $f = \bigvee_{l_i \in Init} l_i$ (line 4). Starting from this disjunction guarantees that every interaction invariant contains at least an initial location. The computation of fixed-points is done by *computeFixedpoint* function (line 5). The returned result is assigned to f.
- Finally, the symbolic set of interaction invariants Ψ is computed by calling the dual function for f (line 6).

There are two main functions in computing interaction invariants by fixed-points: *computeImageVector* and *computeFixedpoint*.

The function *computeImageVector* allows computing, for each location variable l, an image according to its involved interactions. Taking as input a set of interactions γ on a set of components B, the implementation of the function is presented in Algorithm 14: first,

```
1  function computeImageVector(γ, B)
2  begin
3      L = ⋃_{B_i=(L_i,P_i,T_i)∈B} L_i;
4      for each location l_i of L do
5          l_i^→γ = generateForwardLocationPredicate(l_i, γ, B);
6          𝕍[i] = l_i ∧ l_i^→γ;
7      end
8      return 𝕍;
9  end
```

Algorithm 14: Image Vector Generation

we get the set of locations L of B (line 3), then for each location $l_i \in L$, we call function *generateForwardLocationPredicate* to compute its Forward Location Predicate $\overrightarrow{l_i}^{\gamma}$ (lines 4, 5). The image of l_i is obtained by the conjunction of l_i and $\overrightarrow{l_i}^{\gamma}$ (line 6).

```
1  function computeFixedpoint(𝕍, f)
2  begin
3      while true do
4          tmp = Cudd_bddVectorCompose(f, 𝕍);
5          if Cudd_EquivDC(f, 𝕍) then
6              return f;
7          end
8          else
9              f = tmp ;
10         end
11     end
12 end
```

Algorithm 15: Fixed-point Computation

The function *computeFixedpoint* allows computing the fixed-point of a given formula f according to an image function $\mathbb{V}$. The implementation of this function is presented in Algorithm 15:

- Step 1: we replace all location variables in f by their images in the image vector $\mathbb{V}$ and the intermediate result is assigned to *tmp* (line 4). This operation is done by the function *Cudd_ bddVectorCompse(f, img)* in CUDD which creates a new BDD by substituting the BDDs $\mathbb{V}$ for the variables of the BDD f. Then we continue to Step 2.
- Step 2: we compare the intermediate formula *tmp* with f by the function *Cudd_ EquivDC* provided in CUDD (line 5). If they are the same, i.e we have reached fix-points, the iteration terminates and the fixed-points are returned (line 6); otherwise, we assign the intermediate formula *tmp* to f (line 9) and return to Step 1.

5.6.2 Incremental Computation of Interaction Invariants

Similarly to the global symbolic computation of interaction invariants, there are two methods for incremental computation of interaction invariants based on Positive Mapping and

Fixed-point Computation. The implementation of the Incremental Computation of Interaction Invariants module therefore consists of:

- A sub-module for the computation based on positive mapping which is composed of two functions: a function for getting common locations involved in different connectors and a function for incremental computation of interaction invariants by using positive mapping.
- A sub-module for the computation based on fixed-point which allows computing the global image vector and the global fixed-points from the sets of image vectors and fixed-points of constituents.

Below we present in detail the implementation of these incremental methods.

Implementation of the Incremental Positive Mapping-based Method

Given connector γ over a set of components B, a set of increments $\{\delta_1, \ldots, \delta_n\}$ over γ and L_c the set of common location variables of $\{\gamma - (\sum_{i=1}^{n} \delta_i)^f, \delta_1, \ldots, \delta_n\}$, the invariants of the system $(\sum_{i=1}^{n})\gamma(B)$, according to Proposition 14, can be computed as follows:

$$I_{(\sum_{i=1}^{n} \delta_i)\gamma(B)} = dual(\bigwedge_{i=0}^{n} |\delta_i(B)|^{p(L_c)})^p \text{ where } \delta_0 = \gamma - (\sum_{i=1}^{n} \delta_i)^f$$

The incremental computation of interaction invariants for the superposition $(\sum_{i=1}^{n} \delta_i)\gamma(B)$ therefore consists of the following steps:

- First, we need to get the set of common locations L_c of the set $\{\delta_i\}_{i=0}^{n}$.
- Then, we need to build Boolean Behavioral Constraints $|\delta_i(B)|$ for each δ_i. The partial positive mapping is used to remove the negative forms of local locations variables, for each $|\delta_i(B)|$ we obtain $|\delta_i(B)|^{p(L_c)}$.
- We integrate $|\delta_0(B)|^{p(L_c)}, \ldots, |\delta_n(B)|^{p(L_c)}$, then remove the negative forms remaining of common variables and apply the dual operation to obtain the global invariant.

The implementation of the function for getting common locations of a set of connectors $\gamma_1, \ldots, \gamma_n$ is presented in Algorithm 16:

- First, we get the set of components $compList_i$ involved in each connector γ_i (line 3). A component is involved in a connector if its ports participate in any interaction of the connector. Then we get the set of common components *commonCompList* of $\{commpList_i\}_{i=1}^{n}$ (lines 5, 6, 7, 8), that is the components that belong to at least two of these sets. L is the union of the set of control locations of the common components (line 12).

- In the second step, for each location l in L, we get the set of interactions γ_l that the incoming transitions and outgoing transitions of l are involved (lines 16, 17). A transition $\tau = (l, p, l')$ is involved in an interaction a if its port p participates in a. For any two different interactions a_i, a_j of γ_l, if they belong to different connectors, then the location l is common location and is added to the set of common locations L_c (lines 19, 20, 21).

```
1  function getCommonLocations(γ_1, ..., γ_n, B)
2  begin
3      compList_i = {B_j = (L_j, P_j, T_j) ∈ B | P_j ∩ γ_i ≠ ∅};
4      commonCompList = ∅;
5      for each compList_i do
6          for each component B_j of compList_i do
7              if B_j ∈ compList_k, k ≠ i then
8                  commonCompList = commonCompList ∪ B_j;
9              end
10         end
11     end
12     L = ⋃_{B_i ∈ commonCompList} L_i;
13     L_c = ∅; /* list of common locations */
14     for each location l ∈ L do
15         γ_l = ∅;
16         for each transition τ = (l', p, l) or τ = (l, p, l') do
17             γ_l = γ_l ∪ {a ∈ γ | p ∈ a};
18         end
19         for any two interactions a_i, a_j, i ≠ j of γ_l do
20             if connectors of a_j and a_j are not the same then
21                 L_c = L_c ∪ l;
22                 break;
23             end
24         end
25     end
26     return L_c;
27 end
```

Algorithm 16: Common Location Getting

The implementation of the function for computing incrementally interaction invariants is presented in Algorithm 17. The input consists of a connector γ over a set of components $B = (B_1, \ldots, B_m)$, a set of increments $(\delta_1, \ldots, \delta_n)$ over γ, and the initial state $Init$. The computation process is as follows:

- First, we get the set of locations L of all the components $B_1, \ldots, B_m$ (line 3). We denote $\gamma - \sum_{i=1}^{n} \delta_i^f$ by δ_0 (line 4) and get the set of common locations L_c of the set $\{\delta_i\}_{i=0}^{n}$ by the function *getCommonLocations* (line 5).
- Then, for each δ_i, there are the following steps:
  - we get L_i, the set of locations involved in δ_i, i.e the union of locations of components that its ports participate in δ_i (line 7),

1 **function** `computeIncrII`$((\delta_1, \ldots, \delta_n), \gamma, (B_1, \ldots, B_m), Init)$
2 **begin**
3 $L = \cup_{L_i \in B_i} L_i$;
4 $\delta_0 = \gamma - \sum_{i=1}^{n} \delta_i^f$;
5 $L_c =$ `getCommonLocations`$(\delta_0, \ldots, \delta_n, B)$;
6 **for** *each* δ_i **do**
7 $L_i = \bigcup_{P_k \cap \delta_i \neq \emptyset \wedge B = (L_k, P_k, \mathcal{T}_k)} L_k$;
8 $Init_i = \bigvee_{l \in (Init \cap L_i)} l$;
9 $|\delta_i(B)| =$ `generateBBC`(δ_i, B);
10 $|\delta_i(B)|_{init} = |\delta_i(B)| \wedge Init_i$;
11 $|\delta_i(B)|_{\overline{init}} = |\delta_i(B)| \wedge \overline{Init_i}$;
12 $|\delta_i(B)|_{init}^{p(L_c)} =$ `positiveMapping`$(|\delta_i(B)|_{init}, L_c)$;
13 $|\delta_i(B)|_{\overline{init}}^{p(L_c)} =$ `positiveMapping`$(|\delta_i(B)|_{\overline{init}}, L_c)$;
14 **end**
15 $f_{init} =$ **true**, $f_{\overline{init}} =$ **true**;
16 **for** *each* δ_i **do**
17 $f_{init} = (f_{init} \wedge |\delta_i(B)|_{init}^{p(L_c)}) \vee (f_{init} \wedge |\delta_i(B)|_{\overline{init}}^{p(L_c)}) \vee (f_{\overline{init}} \wedge |\delta_i(B)|_{init}^{p(L_c)})$;
18 $f_{\overline{init}} = (f_{\overline{init}} \wedge |\delta_i(B)|_{\overline{init}}^{p(L_c)})$;
19 **end**
20 $f_{init}^p =$ `positiveMapping`$(f_{init}, L \setminus L_c)$;
21 $\Psi =$ `dual`(f_{init}^p);
22 **return** Ψ;
23 **end**

Algorithm 17: Incremental Computation Based on Positive Mapping

- we generate the initial condition $Init_i$ from $Init$ for δ (line 8),
- we generate the Boolean Behavioral Constraints $|\delta_i(B)|$ by function *generateBBC* (line 9).
- $|\delta_i(B)|$ is then decomposed into two parts: one part $|\delta_i(B)|_{init}$ satisfying initial condition by adding $Init_i$ (line 10) and the other part $|\delta_i(B)|_{\overline{init}}$ which does not satisfy the initial condition and is obtained by the conjunction of $|\delta_i(B)|$ and $\overline{Init_i}$ (line 11).
- the positive mapping function *positiveMapping* is called for both $|\delta_i(B)|_{init}$ (line 12) and $|\delta_i(B)|_{\overline{init}}$ (line 13) to eliminate the negative forms of non-common location variables. We obtain respectively $|\delta_i(B)|_{init}^{p(L_c)}$ and $|\delta_i(B)|_{\overline{init}}^{p(L_c)}$.

- Now we integrate the above results. f_{init} (resp. $f_{\overline{init}}$) represents the integrated formula which satisfies (resp. does not satisfy) initial condition $\bigvee Init_i$. Since each final invariant must satisfy the initial condition, we have several kinds of integrations for each connector δ_i: three integrations which return a formula satisfying initial condition: $(f_{init} \wedge |\delta_i(B)|_{init}^{p(L_c)}) \vee (f_{init} \wedge |\delta_i(B)|_{\overline{init}}^{p(L_c)}) \vee (f_{\overline{init}} \wedge |\delta_i(B)|_{init}^{p(L_c)})$ and are then assigned to f_{init} (line 17); an integration which returns a formula that does not satisfy initial condition: $(f_{\overline{init}} \wedge |\delta_i(B)|_{\overline{init}}^{p(L_c)})$ and is then assigned to $f_{\overline{init}}$ (line 18).
- After the integration, we call *positiveMapping* function for f_{init} to remove the remain-

ing negative forms of common location variables (line 20).

- Finally we call the dual function *dual* for the formula f^p_{init} (line 21) to get the global interaction invariants Ψ of the system $\langle(\sum_{i=1}^n \delta_i)\gamma(B), Init\rangle$.

Implementation of the Incremental Fixed-point-based Method

First we provide the implementation of the function for incremental computation of image vector from a set of connectors. Given a set of connectors $\gamma_1, \ldots, \gamma_n$ over a set of components B together with the set of corresponding image vectors $\mathbb{V}_{\gamma_1}, \ldots, \mathbb{V}_{\gamma_1}$, the function presented in Algorithm 18 computes the image vector $\mathbb{V}_\gamma$ for $(\sum_{i=1}^n \gamma_i)(B)$ from $\mathbb{V}_{\gamma_1}, \ldots, \mathbb{V}_{\gamma_1}$.

The computation process is as follows: for every location l of L, the set of locations of atomic components in B, we initialize the image $\mathbb{V}_\gamma(l)$ by l (line 5). Then for any γ_i such that $l \in {}^\bullet\gamma_i$, $\mathbb{V}_\gamma(l)$ is updated by the conjunction with $\mathbb{V}_{\gamma_i}(l)$ (lines 6, 7, 8). Finally, the function returns $\mathbb{V}_\gamma$, the image vector of $(\sum_{i=1}^n \gamma_i)(B)$.

```
1  function computeIncrImageVector((γ1, ..., γn), (V_γ1, ..., V_γn), B)
2  begin
3      L is set of locations of atomic components in B;
4      for each l of L do
5          V_γ(l) = l;
6          for each γi do
7              if l ∈ •γi then
8                  V_γ(l) = V_γ(l) ∧ V_γi(l);
9              end
10         end
11     end
12     return V_γ;
13 end
```

Algorithm 18: Incremental Computation Of Image Vector

Interaction invariants can be incrementally computed by using fixed-point computation for a system resulting from the superposition of a set of increments. the implementation of the method is presented in Algorithm 19. It takes as input a connector γ over a set of components B, a set of increments $\{\delta_i\}_{i=1}^n$, a set of image vector $\{\mathbb{V}_i\}_{i=0}^n$ and a set of fixed-points $\{S_i\}_{i=0}^n$ where $\mathbb{V}_i = computeImageVector(\delta_i, B)$ and $S_i = computeFixedpoint(\mathbb{V}_i, \bigvee_{l\in{}^\bullet\delta_i})$ (or $S_i = computeFixedpoint(\mathbb{V}_i, \bigvee_{l\in Init\cup{}^\bullet\delta_i})$ in the case of system with initial state $Init$) and $\delta_0 = \gamma - (\sum_{i=1}^n \delta_i)^f$.

The function compute the fixed-points for $(\sum_{i=1}^n \delta_i)\gamma(B)$ starting from the sets of constituent fixed-points above. The computation process is as follows:

- First we compute the image vector $\mathbb{V}$ for $(\sum_{i=1}^n \delta_i)\gamma(B)$ from the set of constituent image vector $\{\mathbb{V}_i\}_{i=1}^n$ by calling the function *computeIncrImageVector* (line 3).

- Then the starting point of the fixed-point iteration is set by the disjunction of the constituent fixed-points $\bigvee_{i=0}^n S_i$ (line 4).

```
1 function computeIncrIIByFixedpoint(γ, B, {δ_i}^n_{i=0}, {𝕍_i}^n_{i=0}, {S_i}^n_{i=0})
2 begin
3     𝕍 = computeIncrImageVector({δ_i}^n_{i=0}, {𝕍_{δ_i}}^n_{i=0}, B);
4     S_0 = ⋁^n_{i=0} S_i;
5     S = computeFixedpoint(𝕍, S_0);
6     return S;
7 end
```

Algorithm 19: Incremental Computation of Fixed-point

- Finally, the function *computeFixedpoint* is called to compute the global fixed-points of $(\sum_{i=1}^{n} \delta_i)\gamma(B)$ using S_0 and the global image vector $\mathbb{V}$ (line 5).

5.7 Checking Satisfiability

For systems without data, since the predicate DIS and component invariants can be symbolically represented, the checking of unsatisfiability of $\bigwedge \Phi_i \wedge \Psi \wedge DIS$ can be done by using CUDD package.

For systems with data, the check of unsatisfiability is performed by Yices, a Sat-Solver tool. Given a formula f, Yices checks whether f is satisfiable. If it is, Yices provides a *sat* output together with a solution satisfying f, otherwise an *unsat* output is produced. Since Yices provides just one solution s satisfying the formula f, we need to update f by $f \wedge (\neg s)$ to get another solution. And this step is repeated until f becomes unsatisfiable to get all the solutions. An example of Yices input language for a formula $f = (l1 \wedge (x = 0)) \vee (l2 \wedge (x \geq 1))$ is as follows:

```
(define l1::bool)
(define l2::bool)
(define x::int)
(assert (or (and l1 (= x 0)) (and l2 (>= x 1))))
(check)
```

In the symbolic methods, interaction invariants are generated and stored in a BDD, therefore a transformation is required to convert interaction invariants into a form accepted by Yices. We have implemented this transformation in the Bdd2F function.

Bdd2F Transformation

We have implemented a function Bdd2F which allows transforming a BDD to a formula. The implementation of the Bdd2F transformation function is described in Algorithm 20. Here we provide a transformation from a BDD to a general formula, for a specific form of formula, we just need to change the form of formula to be printed in an output file. Taking as input a BDD representing a formula f and an output file, this function works as follows:

- First, we provide an unique name representing the value of each node of the BDD (lines 6, 7).

- Then, every node of the BDD is considered. A node n corresponds to a variable x and it has two children : a "then" node $tNode$ and an "else" node $eNode$. The value of the node n is therefore either the value of "then" node ($n.val = tVal$) if x is **true** or the value of "else" node ($n.val = eVal$) otherwise. The values of "then" node (tNode) and "else" node (eNode) are specified as follows:

```
1  function Bdd2F(Bdd f, File file)
2  begin
3      index = 0;
4      /* Set value name for each node */;
5      for each Node n of Bdd f do
6          n.val = "n_" + index ;
7          index ++;
8      end
9      /* Convert Bdd to Yices formula */
10     String tVal, eVal ;
11     for each Node n of Bdd f do
12         /* Get ''then'' node of n */
13         Node tNode = cuddT(n);
14         tVal = if isConst(tNode) then "true" else tNode.val;
15         /* Get ''else'' node of n */
16         Node eNode = cuddE(n);
17         if isConst(eNode) then
18             eVal = if isCompl(eNode) then "false" else "true";
19         end
20         else
21             eVal = if isCompl(eNode) then "not " + eNode.val else eNode.val;
22         end
23         x = n.var;     /* get corresponding variable of n                    */
24         print(file, n.val + " = if x then " + tVal + " else " + eVal);
25     end
26     /* Assign f = true by assigning root node to true */
27     Node rNode = f.root;       /* get ''root'' node of f                     */
28     print(file, "rNode.val = true");
29 end
```

Algorithm 20: Bdd-to-formula Transformation

  - If tNode is constant, i.e tNode is a **true** node since "then" node can not be complemented, $tVal = true$. Otherwise $tVal = tNode.val$ (lines 13, 14)
  - If eNode is constant, since "else" node can be complemented, the constant can be **true** or **false** node. If eNode is complemented $eVal =$ **false**, else $eVal =$ **true** (lines 16, 17, 18). If eNode is not constant, we also consider two cases: if eNode is complemented $eVal = eNode.val$, otherwise $eVal = (not\ eNode.val)$ (line 21).
  - The formula $n.val = if\ x\ then\ tVal\ else\ eVal$ for the node n with the values of tVal and eVal specified as above is written to a file (line 24).

- Finally, we get the *root* node and assert its value to "true" (lines 27, 28).

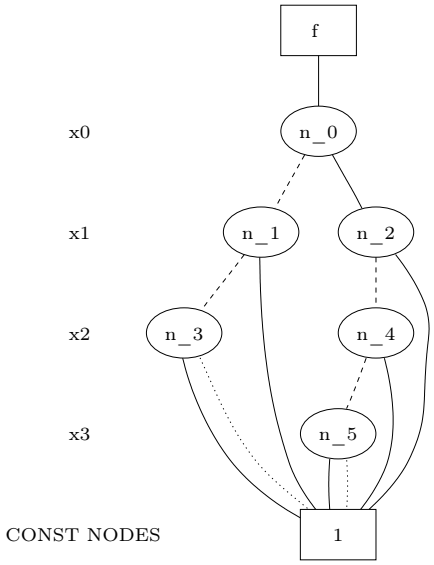


Figure 5.5: A bdd example

Example 43 *Figure 5.5 is an example of a formula represented in a BDD. The formula generated by Bdd2F function for this BDD is as follows:*

$$
\begin{array}{lcl}
n\_5 & = & \textbf{\textit{if}}\ \ x3\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ false \\
n\_4 & = & \textbf{\textit{if}}\ \ x2\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ n\_5 \\
n\_3 & = & \textbf{\textit{if}}\ \ x2\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ false \\
n\_2 & = & \textbf{\textit{if}}\ \ x1\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ n\_4 \\
n\_1 & = & \textbf{\textit{if}}\ \ x1\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ n\_3 \\
n\_0 & = & \textbf{\textit{if}}\ \ x0\ \ \textbf{\textit{then}}\ \ true\ \ \textbf{\textit{else}}\ \ n\_1 \\
f & = & n\_0 \\
\multicolumn{3}{l}{\textbf{\textit{assert}}\ (f = true)}
\end{array}
$$

5.8 Summary

We have provided an overview on the implementation of the D-Finder tool for checking deadlock-freedom of component-based systems. We have first described the structure, the procedure of D-Finder for checking deadlock-freedom together with the corresponding modules, then the detail on the implementation of each module.

An important module is Interaction Invariant Generation where we have implemented several techniques for users to choose. One is based on Yices, a Sat-Solver tool; the others are based on CUDD package and are either enumerative (enumerative method) or totally symbolic (methods based on Positive Mapping and Fixed-point Computation). Moreover,

for the symbolic computation of interaction invariants, we has implemented both global and incremental methods. The global method computes directly invariants for the entire systems from scratch and the incremental method permits computing invariants of composite components from their constituents. The implementation of different methods in D-Finder allows it to handle a large range of systems.

In the next chapter, we are going to present the experimental results obtained by D-Finder on several non-trivial case studies which show the efficiency of the methods as well as the capacity of the D-Finder tool.

CHAPTER 6

Experimentation

Contents

This chapter provides some case studies verified by the D-Finder tool. Two case studies without data are considered to check the scalability of the method. The first is Dining Philosophers, a classical example in detecting deadlocks. The second is Gas Station where we check the scalability by increasing the number of pumps and customers. We also verified two case studies with data: one is Automatic Teller Machine system and the other is a module of a robotic system, robot Dala developed at Laas laboratory. All the experimentations are done on a Linux machine Intel Pentium 4 CPU 3.0 GHz and RAM 1G.

6.1 Dining Philosophers

The Dining Philosophers problem, illustrated in Figure 6.1, can be shortly presented as a number of philosophers sitting at a table doing one of two things: eating or thinking. While they are thinking, they are not eating and while they are eating, they are not thinking. They sit at a round table and each philosopher has a spaghetti disk in front of him. Between two any next philosophers, there is a fork. A philosopher can eat if he has both forks from the left and from the right. And a philosopher can put two forks back to the table only if

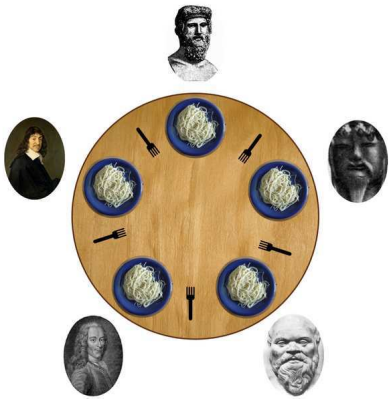


Figure 6.1: Dining Philosophers Problem

he has finished eating. Here there is a deadlock situation if they all have the same order of taking forks and every philosopher has taken one fork, so everyone can not eat because each has only one fork. However, if a Philosophers changes the order of taking forks, for example,

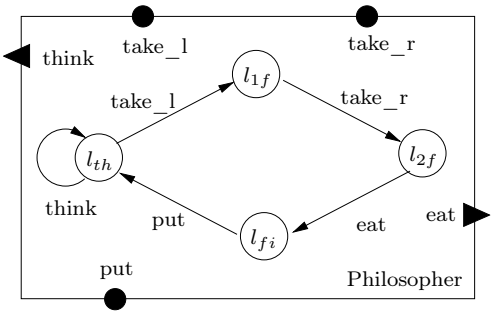


Figure 6.2: Philosopher Component

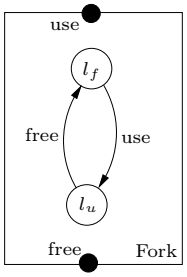


Figure 6.3: Fork Component

he takes the right one before the left one while the others take the left one before the right one, then there is no deadlock. We have considered both cases in the experimentation.

We have modeled the philosopher and the fork in BIP. Figure 6.2 represents the behavior of a philosopher: initially in state l_{th}, he can think by *think* transition or prepare for eating by first taking the left fork (*take_l* transition) and moves to l_{1f} location where he has one fork on the left hand, then taking the right fork (*take_r* transition) and moves to l_{2f} location where he has two forks on both hands. At l_{2f} he can eat by taking *eat* transition and moves to l_{fi} location where he has finished eating, hence he can put two forks by *put* transition and returns to l_t location. The *think* and the *eat* transitions are internal, they

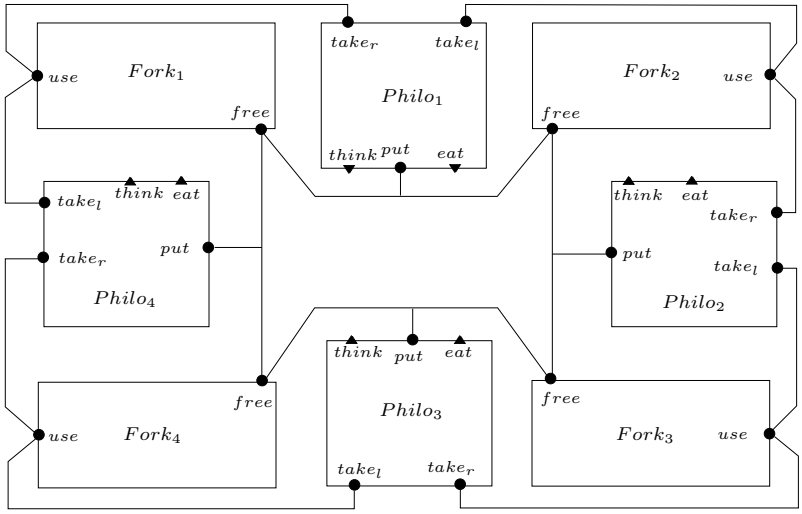


Figure 6.4: Dinning Philosophers with four Philosophers

can be executed without synchronizing with other components and hence their ports are complete, represented by triangles in the figure. The other ports (*take_l*, *take_r*, *put*) are incomplete (represented by circles) and their transitions need to synchronize with other components to be executed.

The model of a fork is quite simple (Figure 6.3): it has two locations l_f and l_u. Initially in the location l_f, a fork can be taken by *use* transition and goes to l_u location. From l_u, a fork is released if the *free* transition is executed and the fork returns to l_f location. Both ports *use* and *free* are incomplete.

6.1.1 Dining Philosophers with Deadlocks

In this model, all Philosophers have the same order in taking forks, we assume that all Philosophers take the left fork before the right fork.

Figure 6.4 shows the model for Dinning Philosophers with 4 Philosophers and 4 forks. In general case, given n philosophers and n forks, the interactions are:

$$\gamma[n] = \sum_{i=1}^{n} (th_i + t\_l_i\ u_i + t\_r_i\ u_{(i \bmod n)+1} + e_i + p_i\ fr_i\ fr_{(i \bmod n)+1})$$

where th_i, t_l_i, t_r_i, e_i, p_i respectively represent the ports *think*, *take_l*, *take_r*, *eat*, *put* of Philosopher $philo_i$, and u_i, fr_i respectively represent the ports *use*, *free* of Fork $fork_i$.

In the incremental construction, these interactions can be obtained from the superposition of a set of increments, each increment adds interactions for philosophers from n_1 to n_2 with $1 \leq n_1, n_2 \leq n$ as follows:

$$\delta_{[n_1,n_2]} = \sum_{i=n_1}^{n_2} (th_i + t\_l_i\ u_i + t\_r_i\ u_{(i\ mod\ n)+1} + e_i + p_i\ fr_i\ fr_{(i\ mod\ n)+1})$$

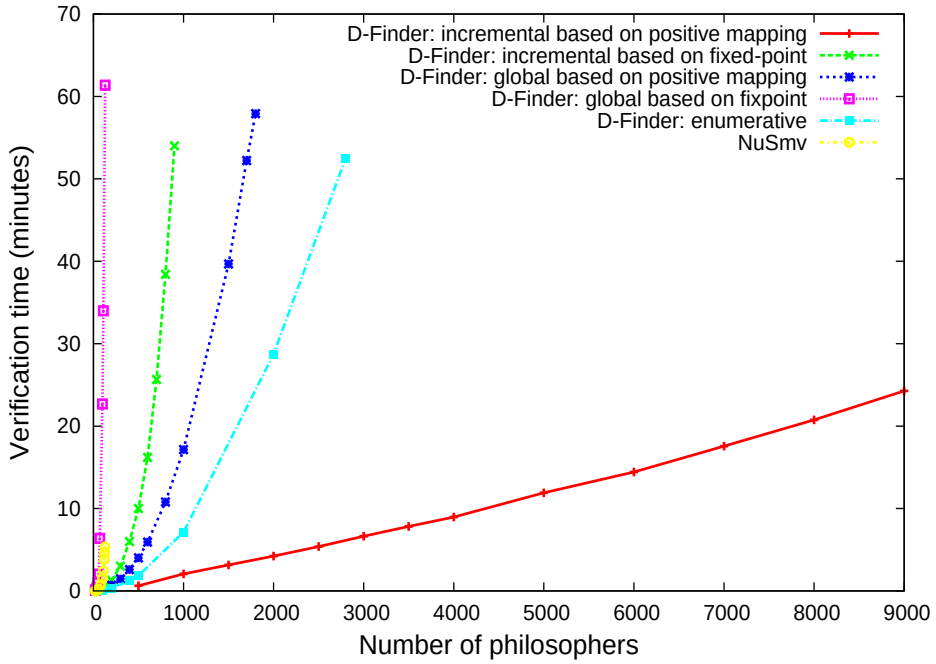


Figure 6.5: Verification Time of Dining Philosopher with Deadlock

For example, the set of interactions for a system of 4 Philosophers is the superposition of the two following increments:
$\delta_{[1,2]} = th_1 + t\_l_1u_1 + t\_r_1u_2 + e_1 + p_1fr_1fr_2 + th_2 + t\_l_2u_2 + t\_r_2u_3 + e_2 + p_2fr_2fr_3$
$\delta_{[3,4]} = th_3 + t\_l_3u_3 + t\_r_3u_4 + e_3 + p_3fr_3fr_4 + th_4 + t\_l_4u_4 + t\_r_4u_1 + e_4 + p_4fr_4fr_1$.

In Figures 6.5, 6.6 and Table 6.1, we provide experimental results on Dining Philosophers. We increase the number of Philosophers and compare the verification time and memory usage between the different methods implemented in D-Finder. We also compare our methods with the well-known verification tool NuSmv. In two figures, x axis represents the number n of Philosophers (and also the number of Forks), y axis respectively represents the verification time (in minutes) and the memory usage (in Mb).

All the methods detected a deadlock and that is the real deadlock where all Forks are at the busy location l_u, i.e. they are being occupied by some Philosopher; all Philosophers are at the location l_l, i.e. they have taken the left fork and are waiting for the right one (but no fork is available):

$$deadlock = \bigwedge_{i=1}^{n} (philo_i.l_{1f} \wedge fork_i.l_u)$$

The invariants generated by D-Finder which allows detecting the above deadlock for a

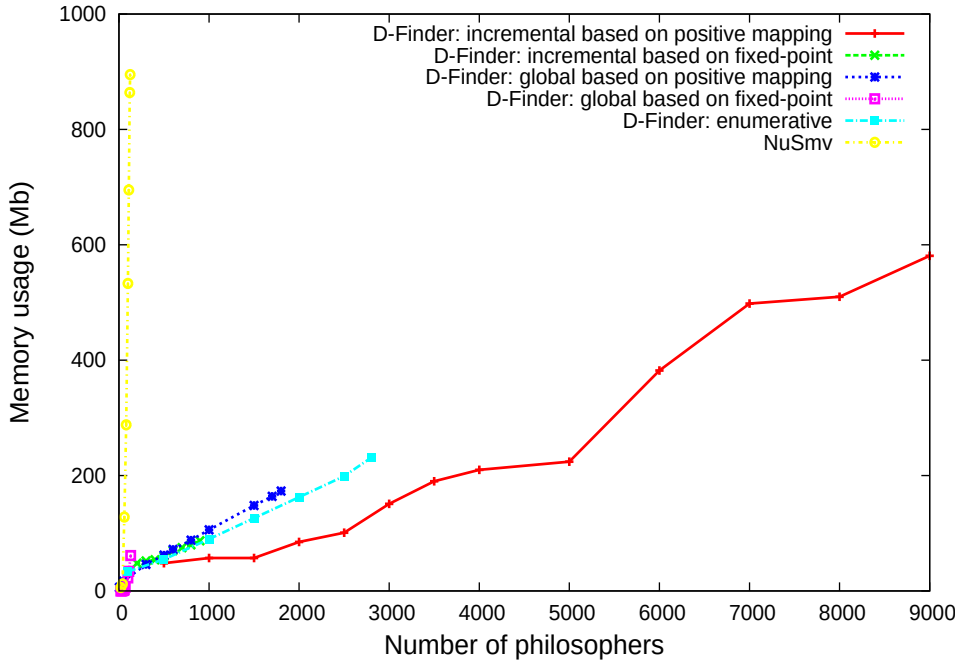


Figure 6.6: Memory Usage of Dining Philosopher with Deadlock

system consisting of n Philosophers and n Forks are as follows:

$$\Psi = \bigwedge_{i=1}^{n} (philo_i.l_{th} \vee fork_i.l_u \vee philo_{(i \bmod n)+1}.l_{th} \vee philo_{(i \bmod n)+1}.l_{1f})$$

According to the experimentation with a time out of 60 minutes, we have:

- Verification time and memory usage by NuSmv increase exponentially. At the size of 130 Philosophers, NuSmv has time out and uses 900Mb over 1000Mb of memory.
- The global verification based on positive mapping can verify up to the size 1800 Philosophers within 60 minutes. The memory used by this method is low: for 1800 Philosophers, it uses less than 200 Mb.
- The global verification based on fixed-point is not good for this example. The reason is that the Philosophers example has cycle structure which has invariants involved in all the components of the system, hence the number of iterations in fixed-point computation is big since the iteration process has to pass over the whole cycle (all the components) to get these invariants.
- The global enumerative verification is better than the global symbolic methods, it can verify up to the size of 2800 Philosophers within 60 minutes and uses less that 250 Mb of memory. The reason is that the number of interaction invariants of this

Component information			Time (minutes)						Memory (MB)					
philos	locs	intrs	*Smv*	*Enum*	*PM*	*FP*	*IPM*	*IFP*	*Smv*	*Enum*	*PM*	*FP*	*IPM*	*IFP*
100	600	500	1:32	0:06	0:13	22:41	0:4	0:19	533	34	46	75	10	32
500	3000	2500	-	1:51	4:01	-	0:34	10:03	-	55	61	-	29	60
1000	6000	5000	-	7:08	17:09	-	2:04	-	-	90	105	-	60	-
1500	9000	7500	-	19:30	39:40	-	3:09	-	-	126	148	-	74	-
2000	12000	10000	-	28:44	-	-	4:14	-	-	163	-		96	-
4000	24000	20000	-	-	-	-	8:37	-	-	-	-	-	192	-
6000	36000	30000	-	-	-	-	14:26	-	-	-	-	-	382	-
7000	42000	35000	-	-	-	-	17:34	-	-	-	-	-	498	-
8000	48000	40000	-	-	-	-	20:45	-	-	-	-	-	510	-
9000	53000	45000	-	-	-	-	24:16	-	-	-	-	-	581	-

philos:	number of philosophers	Smv:	NuSmv	PM:	Global Positive Mapping
locs:	number of locations	Enum:	Enumerative	IPM:	Incremental Positive Mapping
intrs:	number of interactions	FP:	Global Fixed-point	IFP:	Incremental Fixed-point

Table 6.1: Comparison between different methods on Dining Philosophers

example does not increase exponentially in the size of the systems. For example, the system of 2000 Philosophers has 10003 interaction invariants and the system of 2800 Philosophers has 13003 interaction invariants.

- The incremental method based on positive mapping consumes almost linearly time according to the size of the system. Each increment adds a set of interactions for 500 Philosophers and 500 Forks. The incremental verification of the system built from 18 increments (corresponding to a system of 9000 Philosophers and 9000 Forks) is done in 25 minutes and uses less than 600Mb of memory.

6.1.2 Dining Philosophers without Deadlocks

In this model, all Philosophers have the same order in taking forks except a Philosopher have different order, we assume that the first Philosopher takes the right fork before the left, and the others take the left fork before the right.

We did the experimentation with the methods implemented in D-Finder and NuSmv. All the methods report that the system is deadlock-free. The interaction invariants generated by D-Finder which allow proving deadlock-freedom of a system having n Philosophers and n Forks are as follows:

$$\Psi = \bigwedge_{i=2}^{n-1}(philo_i.l_{th} \vee fork_i.l_u \vee philo_{(i \bmod n)+1}.l_{th} \vee philo_{(i \bmod n)+1}.l_{1f})$$

$$\bigwedge(philo_1.l_{2f} \vee philo_1.l_{fi} \vee fork_1.l_f \vee philo_2.l_{2f} \vee philo_2.l_{fi})$$

$$\bigwedge(philo_1.l_{th} \vee philo_1.l_{1f} \vee fork_1.l_u \vee philo_2.l_{th} \vee philo_2.l_{1f})$$

$$\bigwedge(philo_1.l_{th} \vee fork_n.l_u \vee philo_n.l_{th})$$

The performance of the different methods of D-Finder is almost similar to the performance in verifying the Philosophers system with deadlocks presented in the previous subsection.

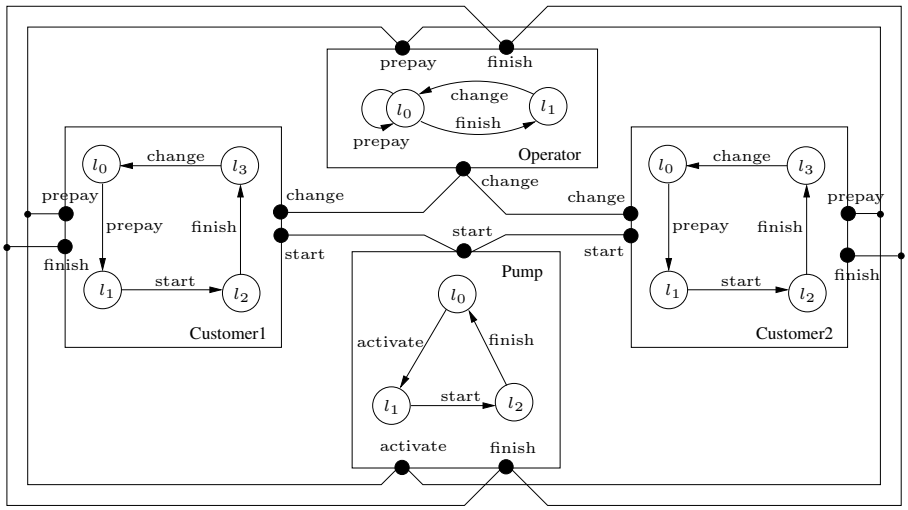


Figure 6.7: Sketch of Gas Station

6.2 Gas Station

Gas Station [HL85] consists of an Operator with a computer, a set of pumps, and a set of customers. Each pump can be used by a fixed number of customers. The set of the atomic components involved in a system with n pumps and m customers for each pump is denoted by $B[n,m] = \{Operator, \{pump_i\}_{1\leq i\leq n}, \{customer_{ij}\}_{1\leq i\leq n, 1\leq j\leq m}\}$.

Before using a pump, each customer has to prepay for the transaction. Then the customer uses the pump, collects his change and goes to a state from which he may start a new transaction.

Before being used by a customer, the pumps have to be activated by the Operator. When a pump is shut off, it can be re-activated for the next operation.

Figure 6.7 gives the model for Gas Station system for one pump and two customers. The Operator has two control locations and three ports.The transition labeled with *prepay* accepts a customer's prepay and activates the pump for the customer. When a customer is served, the transition labeled with $finish$ will synchronize the pump and the customer. A pump has three control locations and three ports. Besides the synchronization between the Operator and customer through *activate* and $finish$ ports, a pump and a customer are synchronized through *start* ports.

We abbreviate port names by using only their first three letters. The ports of Operator are respectively pre, fin, cha; the ports of $pump_i$ are respectively act_i, sta_i, fin_i and the ports of $customer_j$ of $pump_i$ are pre_{ij}, sta_{ij}, fin_{ij}, cha_{ij}. The interactions for a system of n pumps, each one used by m customers, are

$$\gamma[n,m] = \sum_{i=1}^{n}(\sum_{j=1}^{m}(pre\ act_i\ pre_{ij} + sta_i\ sta_{ij} + fin\ fin_i\ fin_{ij} + cha\ cha_{ij}))$$

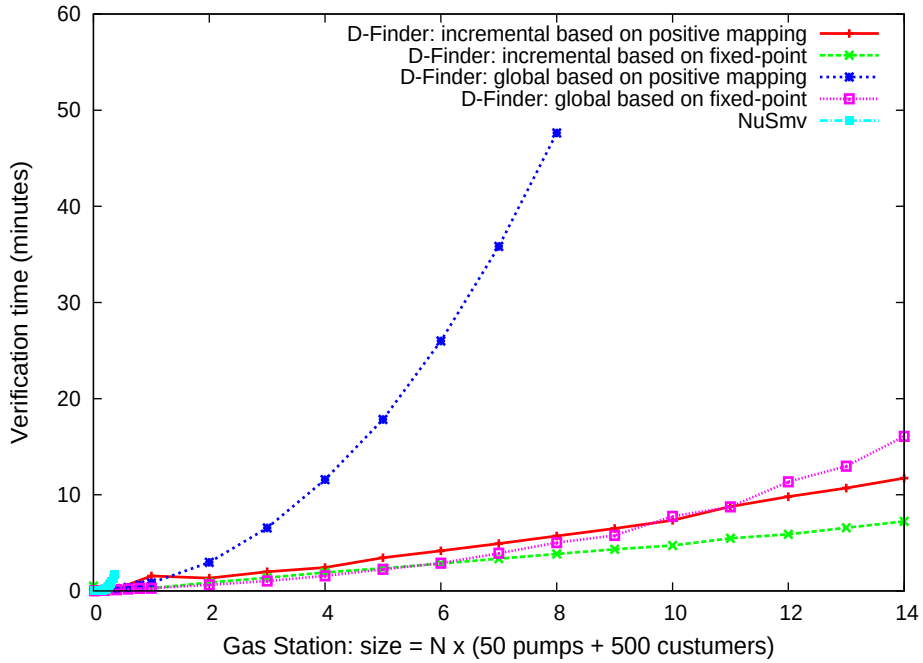


Figure 6.8: Gas Station - Verification Time

The system consisting of n pumps and m customers can be incrementally built by the superposition of a set of increments, each increment adds interactions for a set of pumps, a set of customers and Operator. If each pump connects to m customers, the increment that adds interactions for the pumps from n_1 to n_2 is:

$$\delta[n_1, n_2, m] = \sum_{i=n_1}^{n_2}(\sum_{j=1}^{m}(pre\ act_i\ pre_{ij} + sta_i\ sta_{ij} + fin\ fin_i\ fin_{ij} + cha\ cha_{ij}))$$

For example, the system of two pumps and two customers can be built from two increments, each increment contains interactions over a pump, a customer and Operator as follows:

$$\delta_1[1,1,1] = pre\ act_1\ pre_{11} + sta_1\ sta_{11} + fin\ fin_1\ fin_{11} + cha\ cha_{11}$$
$$\delta_2[2,2,1] = pre\ act_2\ pre_{21} + sta_1\ sta_{21} + fin\ fin_1\ fin_{21} + cha\ cha_{21}$$

D-Finder reports deadlock-freedom of the Gas Station system. The interaction invariants generated by D-Finder which prove the deadlock-freedom of a Gas Station system consisting of n pumps, each pump $pump_i$ has m_i customers, are as follows:

$$\Psi = \bigwedge_{i=1}^{n}(pump_i.l_0 \vee pump_i.l_1 \vee \bigvee_{j=1}^{m_i} customer_{ij}.l_2)$$

$$\wedge \bigwedge_{i=1}^{n}(pump_i.l_0 \vee pump_i.l_2 \vee \bigvee_{j=1}^{m_i} customer_{ij}.l_1)$$

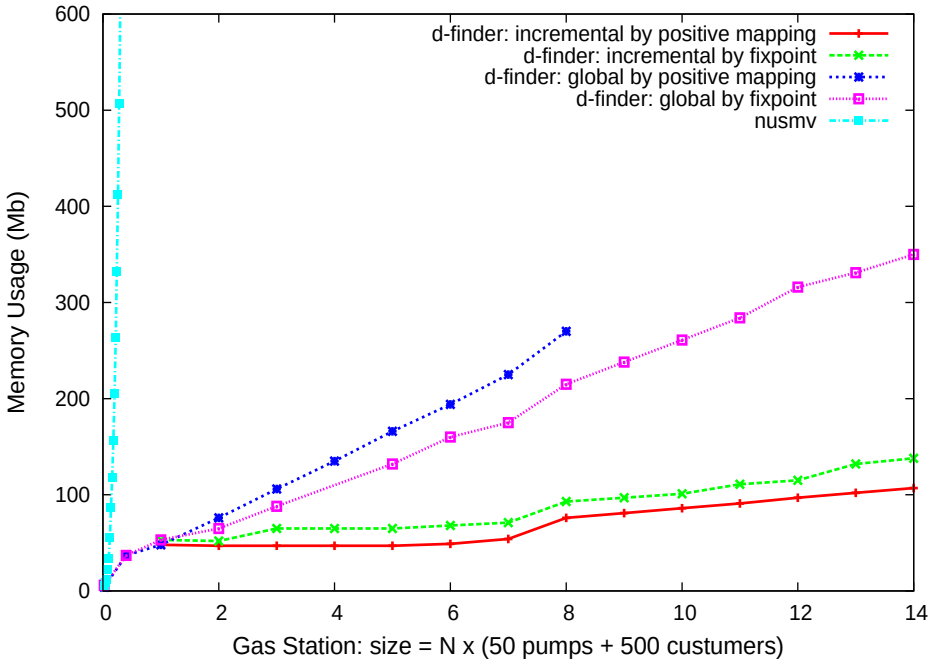


Figure 6.9: Gas Station - Memory Usage

$$\wedge \bigwedge_{i=1}^{n} (operator.l_1 \vee pump_i.l_1 \vee pump_i.l_2 \vee \bigvee_{j=1}^{m_i} customer_{ij}.l_0)$$

$$\wedge \bigwedge_{i=1}^{n} (operator.l_0 \vee \bigvee_{i=1}^{n} \bigvee_{j=1}^{m_i} customer_{ij}.l_3)$$

Figures 6.8, 6.9 and Table 6.2 present the verification time and memory usage for Gas Station system by different methods implemented in D-Finder and by NuSmv. For the incremental verification methods, N is number of increments and each increment adds interactions for 50 pumps and 500 customers. Hence, the size of a system built from N increments is $N \times 50$ pumps and $N \times 500$ customers. For the global verification method, the number of pumps is $N \times 50$ and the number of customers is $N \times 500$. We also set the time out to 60 minutes. According to the experimentation:

- NuSmv reaches timeout at the size (18 pumps + 180 customers) and uses 920Mb over 1000Mb of memory.
- The global symbolic method based on positive mapping can verify up to the size 8×(50 pumps + 500 customers) within 50 minutes and uses 300Mb of memory.
- The incremental method based on positive mapping has almost linear verification time and memory usage according to the size of the system. It can verify up to the size 14×(50 pumps + 500 customers) in 12 minutes and uses 107 Mb of memory.

Component information			Time (minutes)					Memory (MB)				
pumps	locs	intrs	*NuSMV*	*PM*	*FP*	*IPM*	*IFP*	*NuSMV*	*PM*	*FP*	*IPM*	*IFP*
18	776	720	1:43	0:18	0:8	0:9	0:9	921	48	35	17	17
50	2152	2000	-	0:50	0:17	0:49	0:17	-	48	53	47	53
100	4302	4000	-	2:58	0:38	1:51	0:52	-	76	65	47	52
200	8602	8000	-	11:34	1:34	2:26	1:55	-	135	107	47	65
300	12902	12000	-	26:00	2:53	4:11	2:85	-	194	160	49	68
400	17202	16000	-	47:38	5:01	5:34	3:51	-	270	215	76	93
500	21502	20000	-	-	7:45	7:21	4:43	-	-	261	86	101
600	25802	24000	-	-	11:21	9:05	5:53	-	-	316	97	115
700	30102	28000	-	-	16:04	11:44	7:14	-	-	350	107	138

pumps : number of pumps
locs : number of locations
intrs : number of interactions
Smv : NuSmv
PM : Global Positive Mapping
FP : Global Fixed-point
IPM : Incremental Positive Mapping
IFP : Incremental Fixed-point

Table 6.2: Comparison between different methods on Gas Station

Figure 6.10: ATM Structure

- The global symbolic method and incremental method based on fix-point have very good performance compared to the corresponding methods based on positive mapping. The reason is that there is a small cycle of interactions between the Operator, a Pump and a Customer, hence the number of iterations to reach fixed-points is small (7 iterations) because the iteration process just has to pass over these three components to reach a fix-point.

Figure 6.9 shows that the incremental methods gain not only in verification time but also in memory usage compared to the global methods.

6.3 Automatic Teller Machine ATM

Automatic Teller Machine (ATM) is a computerized telecommunication device that provides services to access to financial transactions in a public space without the need for a cashier, human clerk or bank teller.

The structural model of the ATM system [CEFH01] is presented in Figure 6.10. The system is composed of the following components: User, ATM (modeling a cash dispenser) and Bank (modeling some aspects of bank operation). User and Bank interact only with ATM, but not with each other.

Figure 6.11 presents the modeling of ATM in BIP:

- Initially at location l_0, user can insert the card by *insert* transition and enter the confidential code (*enter* transition). Then there are two cases: if the code is invalid,

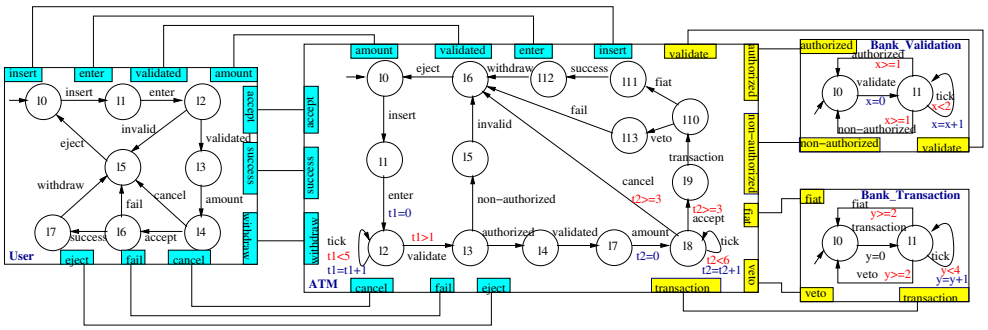


Figure 6.11: Modeling of ATM system in BIP

user gets back the card by *eject* transition and returns to the initial state l_0; otherwise, user continues by entering the amount of cash he/she wants to withdraw. If the amount is not accepted, the transaction is canceled (*cancel* transition); else there are two cases: transition fails (*fail* transition) or is ready (*success* transition) for user to withdraw the money. Finally user gets back their card.

- Initially at location l_0, ATM is waiting for user to insert the card (*insert* transition) and then to enter the confidential code (*enter* transition). The time-out for entering the code is 5 time units then it validates the entered code. If it receives non-authorized for the code by *non_authorized* transition, *invalid* transition takes place and then it ejects the card. If it receives authorized signal by *authorized* transition, the transition *validated* takes place and it moves to a location where user can enter the amount of cash. The timeout for entering the amount is 6 time units. If the user cancels the transaction or the amount is not allowed, it returns to l_6 to eject the card; else it accepts the amount and starts the transaction. If the transaction is forbidden (*veto* transition), it will announce to user by *fail* transition; else it will wait for user to withdraw the cash (*withdraw* transaction) and eject the card to finish the transaction.

- For Bank, there are two components: *BankValidation* component checks the validity of PIN code and *BankTransaction* component checks whether the transaction is forbidden (*veto*) or allowed (*fiat*). The use of these parallel components allows supporting multi Users and multi ATMs.

We abbreviate port names by using their first three letters except *val_ed* for *validated* and *non_aut* for *non_authorized*. We also use $[port]_i^u$, $[port]_i^a$, $[port]^{bt}$, $[port]^{bv}$ to represent respectively the ports of $user_i$, atm_i, *BankTransaction* and *BankValidation* components. The set of interactions for an ATM system with n ATMs and n Users is as follows (Figure 6.11):

$$\gamma = \sum_{i=1}^{n}(ins_i^u ins_i^a + ent_i^u ent_i^a + val\_ed_i^u val\_ed_i^a + inv_i^u inv_i^a + amo_i^u amo_i^a + can_i^u can_i^a$$

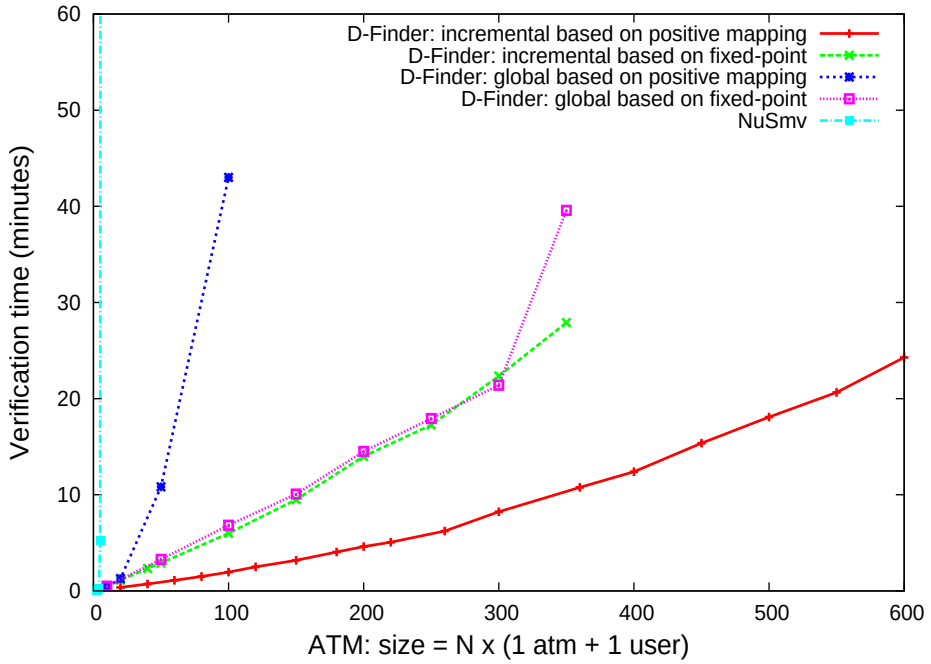


Figure 6.12: ATM Verification Time

$$+acc_i^u acc_i^a + fai_i^u fai_i^a + suc_i^u suc_i^a + wit_i^u wit_i^a + eje_i^u eje_i^a + aut_i^a aut^{bv} + val_i^a val^{bv}$$

$$+non\_aut_i^a non\_aut^{bv} + fia^a fia^{bt} + vet^a vet^{bt} + tra^a tra^{bt} + tic^a + tic^{bv} + tic^{bt})$$

In the incremental construction, the ATM system with n ATMs and n Users is built by the superposition of a set of n increments, each increment adds interactions for a set of ATMs and a set of Users. An increment which consists of interactions overs the ATMs from n_1 to n_2 is as follows:

$$\delta[n_1, n_2] = \sum_{i=n_1}^{n_2} (ins_i^u ins_i^a + ent_i^u ent_i^a + val\_ed_i^u val\_ed_i^a + inv_i^u inv_i^a + amo_i^u amo_i^a$$

$$+can_i^u can_i^a + acc_i^u acc_i^a + fai_i^u fai_i^a + suc_i^u suc_i^a + wit_i^u wit_i^a + eje_i^u eje_i^a + aut_i^a aut^{bv}$$

$$+val_i^a val^{bv} + non\_aut_i^a non\_aut^{bv} + fia^a fia^{bt} + vet^a vet^{bt} + tra^a tra^{bt} + tic^a + tic^{bv} + tic^{bt})$$

D-Finder reports deadlock-freedom on the ATM system. Figure 6.12 and Table 6.3 shows the experimental results on ATM system by different methods:

- Global method based on positive mapping can verify up to (100 atms + 100 users) within 50 minutes.

Component information				Time (minutes)				
atms	locs	intrs	vars	*NuSmv*	*PM*	*FP*	*IPM*	*IFP*
5	1104	902	102	5:15	0:3	0:2	0:4	0:4
50	1104	902	102	-	10:49	3:17	1:23	2:20
100	2204	1802	202	-	43:00	6:50	1:57	6:00
200			402	-	-	14:30	4:37	14:00
250	5504	4002	502	-	-	17:56	4:46	17:16
300			602	-	-	21:24	8:14	22:21
350	7704	6302	702	-	-	39:35	8:14	27:54
400			802	-	-	-	12:24	-
500			1002	-	-	-	18:05	-
600	13204	10802	1202	-	-	-	24:17	-

atms : number of atms
locs : number of locations
intrs : number of interactions
vars : number of variables

PM : Global Positive Mapping
FP : Global Fixed-point
IPM : Incremental Positive Mapping
IFP : Incremental Fixed-point

Table 6.3: Comparison between different methods on ATM system

- The time for the verification by the incremental method based on positive mapping is almost linear. We built the ATM system by the superposition of a set of increments, each increment adds interactions for (10 atms + 10 users) and two Bank components. The reason for the good performance is that the number of common components between increments is small (just two Bank components), and therefore the number of common locations is also small.

- The global and incremental methods based on fixed-point have almost the same performance when the size of system is not too big (up to 300 atms + 300 users). The reason is that the number of iterations to get fixed-points is the same for any number of atms and users and this number is quite small, 19 iterations. Therefore the global method based on fixed-point is quite fast. Moreover, in the incremental method based on fixed-point, in the final step where we compute global fixed-points starting from fixed-points obtained from increments, the number of iterations is 11 which is not much smaller than the number of iterations in the global method. That is the reason for the same performance of two methods based on fixed-points.

6.4 NDD Module of Dala Robot

Robot Dala, an iRobot ATRV, has been developed at LAAS laboratory. It is composed of three layers (Figure 6.13):

- Functional layer includes all the basic built-in robot actions and perception capacities (image processing, motion control, etc.)

- Decisional layer produces the task plan and supervises its execution.

- Execution control level is an interface between the decisional and functional layers that controls the execution of services in the functional layer according to some safety constraints.

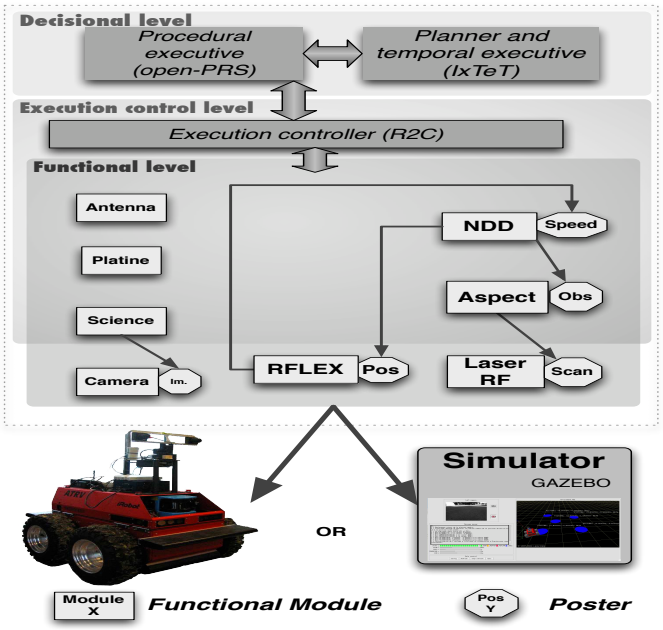


Figure 6.13: The Dala Robot Architecture

We have used BIP to model the execution control and functional layers of the robot Dala. The functional layer consists of a set of modules. A module has a set of services, a set of execution tasks and a set of posters where the produced data is stored. A service has a controller and an activity. An execution task is composed of a timer, a scheduler and an activity. We proposed the following grammar which allows building the functional level starting from basic components:

$$
\begin{aligned}
&\textit{Functional level} ::= (\textit{Module})^{+} \\
&\textit{Module} ::= (\textit{Service})^{+}.(\textit{Execution task})^{+}.(\textit{Poster})^{+} \\
&\textit{Service} ::= (\textit{Service controller}).(\textit{Activity}) \\
&\textit{Execution task} ::= (\textit{Timer}).(\textit{Scheduler activity})
\end{aligned}
$$

where $^{+}$ (plus) means the presence of one or more subcomponent and . (dot) means the composition of different components.

We have used D-Finder to check deadlock-freedom of a module in the functional level, module NDD (Figure 6.14), which is one of the most complex modules. It has totally 27 components, 144 control locations, 117 connectors between components, 16 boolean variables and 11 integer variables. NDD module is responsible for the navigation of the robot, that is to reach a goal while avoiding obstacles. It consists of the following control elements:

- *InterfaceServer* is the interface of the module with the decisional layer. It checks the mailbox which is a shared memory and if there is any message, it will read the content and then sends requests to the corresponding service.
- *ExecutionControl* keeps information about the number of services running in the module. If a service is triggered, it increases the number by 1, if a service finishes, it decreases the number by 1.
- *ExecutionTask* runs periodically to synchronize the executions of different services, that is different services can be executed within a period but a service can not be executed more than one time within a period.

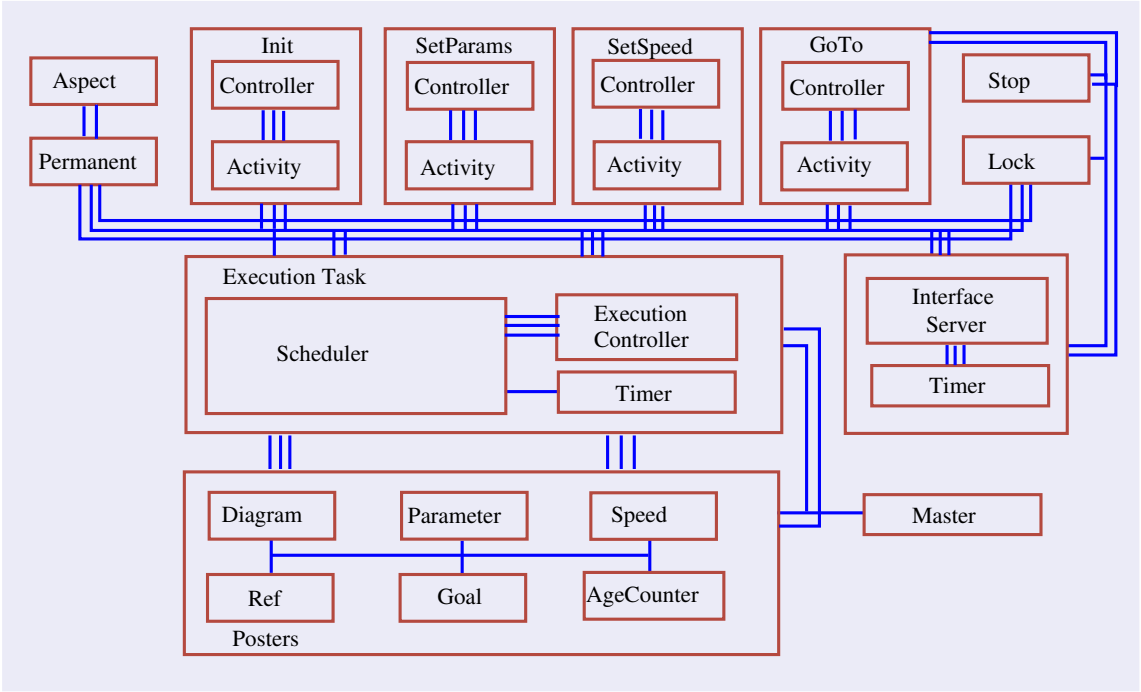


Figure 6.14: NDD Module

and the following services:

- *PermanentTask* computes the speed of the robot and it is executed periodically during the execution of the robot.
- *Init* service initializes the module.
- *SetParams* service sets the necessary parameters of the module.
- *SetSpeed* service sets the moving speed of the robot which is computed by *PermanentTask*.

- *GoTo* service allows the robot moving to a given destination.
- *Stop* service allows stopping the robot at any time.

NDD also has a set of components called Poster (*SpeedPoster, ParamPoster, DiagramPoster, AspectPoster, RefPoster, GoalPoster, AgePoster, MasterPoster*) where data produced by the services of the module is stored and exchanged between different services of NDD or with services of other modules.

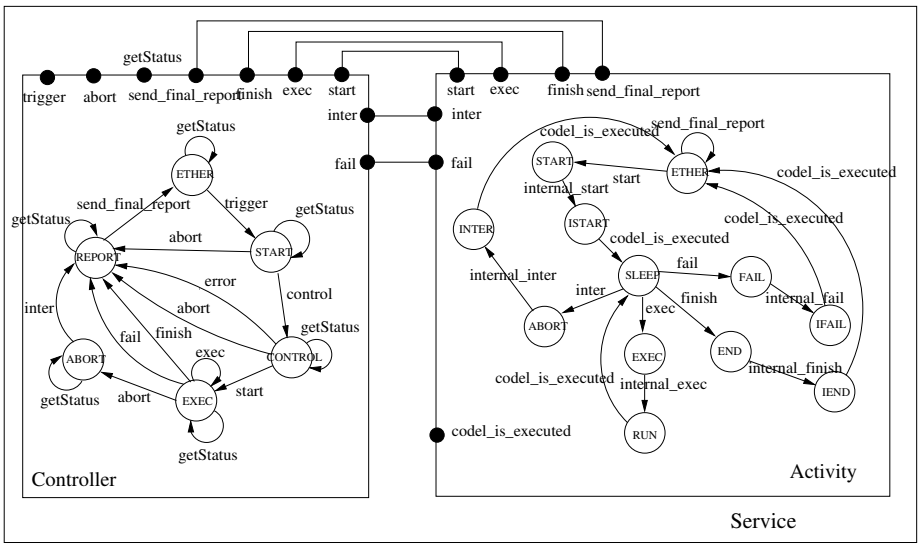


Figure 6.15: A service

A service basically has two components (Figure 6.15): a Controller and an Activity. The Controller receives requests (*trigger* transition), checks parameters and execution conditions (*control* transition) and if everything is fine, the Controller will trigger the Activity (*start* transition) to perform the request. The Controller can cancel the request if there is an error (*error* transition) or conflict (*abort* transition). The Controller updates the status of the Activity by $finish$, *inter* and $fail$ transitions. And finally, it sends a report ($send\_final\_report$ transition) to the *ExecutionControl* component.

The Activity is triggered by the Controller (*start* transition) and then it executes its functions to perform the requested task (*exec*, $internal\_exec$ transitions). The execution may finish normally ($finish$ transition), may fail ($fail$ transition) or may be interrupted (*inter* transition). In any case, the Activity informs the result to the Controller.

We have first used the global symbolic method implemented in D-Finder to check the deadlock-freedom of NDD. We have found potential deadlocks due to the strong synchronization between the timers. We have then fixed these problems and verified again. Finally we obtained the result proving deadlock-freedom of the module. The verification time of the global method by positive mapping is up to 3 hours. However, the incremental verification method based on positive mapping reduces dramatically the verification time: the

module	comps	locs	intrs	vars	time (minutes)	memory (mb)
RFLEX	56	308	227	35	9:39	165
NDD	27	152	117	27	14:25	113
SICK	43	213	202	29	6:07	92
Aspect	29	160	117	21	1:00	80
Battery	30	176	138	23	4:34	65
Heating	26	149	116	19	0:39	57
Platine	37	174	151	25	8:47	97

module : module name intrs : number of interactions
comps : number of components vars : number of variables
locs : number of locations

Table 6.4: Time and memory usage for the verification of Dala robot modules

deadlock-freedom of the module is proven within 20 minutes. In the incremental construction and verification process, we build and check the deadlock-freedom of the NDD module from the following increments:

- increment 1 consisting of interactions between *PermanentTask*, *Init*, AspectPoster, *InterfaceServer*, *ExecutionTask*, Lock and *ExecutionControl*,
- increment 2 consisting of interactions between *SetParams*, *SetSpeed*, *InterfaceServer*, *ExecutionTask*, Lock and *ExecutionControl*,
- increment 3 consisting of interactions between GoTo, Stop, *InterfaceServer*, *ExecutionTask*, *Lock* and *ExecutionControl*,
- increment 4 consisting of interactions between *DiagramPoster, RefPoster, ParamPoster, GoalPoster, SpeedPoster, AgePoster and MasterPoster*.

Table 6.4 shows the verification time and memory usage for checking deadlock-freedom of other modules in the robot Dala by D-Finder using the incremental positive mapping method. D-Finder detected deadlocks in several modules. Based on the detected deadlocks, we fixed their models and then we successfully proved the deadlock-freedom of these modules.

6.5 Summary

We have provided the experimental results on the verification of several case studies by D-Finder. The experimental results have shown the efficiency of the methods and the capacity of D-Finder. D-Finder is able to check deadlock-freedom of systems up to more than ten thousands of components, more than fifty thousands of control locations and about the same number of interactions. Besides the scalability, D-Finder also shows the capacity in dealing with complex systems with a significant number of boolean and integer variables. Moreover, the different methods implemented in D-Finder allows handling many kinds of systems such as cycle-structure or star-structure systems.

We also compare the verification time and memory usage by D-Finder of some case studies with an well-known model checking tool NuSmv. In all the case studies, D-Finder provides much better performance than NuSmv in both verification time and memory usage. Moreover, the experimental results also show the significant gain by the incremental verification methods compared to the global verification.

Part IV

Conclusions and Perspectives

CHAPTER 7

Conclusions and Perspectives

Contents

In this chapter, we conclude the thesis describing the main objectives of the work, the goals we have achieved, the future work directions and its perspectives.

7.1 Conclusions

Constructing correct systems is alway an essential requirement for engineers. However with the growing demand of the complexity and of the size of systems, it becomes more and more difficult to build correctly a system with respect to its specification. The violation of some property in systems, specially in critical systems, might cost expensively. Therefore the check of correctness is essential to guarantee the correct behavior of the system.

In this thesis, we propose a compositional verification method for checking safety properties of component-based systems described in the BIP language. Our method is based on the the use of two kinds of invariants: component invariants which approximate reachable states of atomic components and interaction invariants which characterize global constraints induced by strong synchronizations between atomic components.

There are two key issues in the application of the method. The first is the choice of component invariants depending on the property to be proved. The second is the computation of the corresponding interaction invariants. Here there is a risk of explosion, if exhaustiveness of solutions is necessary in the analysis process. However, this issue is solved by using symbolic computation using BDDs.

For the computation of component invariants, we use lightweight techniques which allow computing increasingly stronger component invariants by using forward propagation over

the transitions of components.

For the computation of interaction invariants, we have proposed different methods using both SMT Sat-Solver and BDDs. The enumerative method using CUDD package or Yices computes explicitly the set of interaction invariants. This method provides a clear, visual view on the invariants and it is efficient for the systems of which the number of invariants is not too big. The symbolic methods using CUDD package compute interaction invariants based on symbolic operations performed directly on the BDDs. One symbolic method is based Positive Mapping operation which removes symbolically the negative valuations of variables from Boolean Behavioral Constraints. The other symbolic method uses fixed-point computation technique to compute, starting from the set of single locations, all the locations that can be reached from these locations by any interaction of the system.

The application of our verification method for proving deadlock-freedom of component-based systems is promising. The class of component invariants that we use captures well enough guarantees for component deadlock-freedom. Their computation does not involve fixed-points and avoids state space explosion. The verification method applies an iteration process for computing progressively stronger invariants. Best precision is achieved when component reachability sets are used as component invariants. This is feasible for finite state components. There are no restrictions on the type of data as long as we stay within theories for which there exist efficient procedures.

We have also improved significantly the compositional verification method by proposing incremental construction and verification method. The incremental method take advantage of properties of the construction process based on the assumption that composite components can be obtained from a set of atomic components by superposition of increments. The verification should be applied in each stage of incremental construction process in order to detect early the violations. Hence the reuse of established properties in the checking of global properties is essential to reduce the verification cost. We proposed the rules on invariant preservation from which the established properties are not violated during the incremental construction. For the general systems where the preservation rules may not hold, we proposed a method for incrementally computing invariants from the established invariants of the increments.

Since the compositional and incremental methods are limited to systems without data transfer over interactions, we have provided a method allowing dealing with systems with data transfer. The method is based on the transformation of systems with data transfer into equivalent systems without data transfer on which the compositional method can be applied. The transformation is done by taking into account the data transfer in computing component invariants and by replacing data transfer over a set of interactions by a component called “interaction component”. This method has not been implemented but the results obtained on several examples show the perspectives of the method.

We have fully implemented compositional and incremental methods in the D-Finder tool. D-Finder allows checking safety properties, specially deadlock-freedom, of component-based systems described in BIP. The implementation of different methods using different techniques make D-Finder efficient in dealing with many kinds of systems, for example “star-structural” systems, “cycle-structural” systems. It also allows users choosing the appropriate method and technique according to their systems to be verified.

We have used D-Finder for checking deadlock-freedom of several non-trivial case studies which showed the capacities of D-Finder as well as the efficiency of the methods. Two case studies without data (Dining Philosopher, Gas Station) are used to check the scalability of the methods. Two other case studies with data (ATM machine, NDD module of Dala robot) are used to show the capacities of the methods in dealing with complex systems. Specially, NDD is a module of Dala robot, a real complex case study of our projects, where D-Finder detected errors in the model. The obtained experimental results by D-Finder are really convincing. D-Finder can handle very large systems with acceptable verification time and memory usage. The experimental results also show significant gain of incremental compositional methods compared to global compositional methods in both verification time and memory usage.

We also compared D-Finder to a well-known monolithic tool NuSmv. In [CAC08], Cobleigh et al. show for a set of finite state benchmarks that only for 30% of the considered benchmarks, assume-guarantee tools outperform model-checking tools. On the contrary, for all the case studies that we have verified by using D-Finder and monolithic model checkers, D-Finder outperform these tools, in particular for large systems, in both verification time and memory usage. Of course this comparison is not completely balanced because D-Finder uses heuristics and is tuned for checking deadlock-freedom.

7.2 Perspectives

The main perspectives of our works can be categorized in two directions:

- The extension of the usability of component verification which involves the following aspects:
  - First, the tool should be able to handle richer models, e.g., the method for dealing with data transfer should be implemented in D-Finder.
  - Second, the methods should provide counter-example guided abstraction refinement. The idea is to apply the method presented in [BM07] which allows generating an inductive invariant corresponding to a counter-example. If we succeed to find this invariant, the counter-example is excluded. Moreover, we can use this invariant to strengthen the already established invariants and therefore to eliminate many others spurious counter-examples.
  - Third, we can develop heuristics for properties other than deadlock-freedom, e.g., partial deadlocks, live-lock. We have exploited the invariants corresponding to notions of traps in Petri-net. However, there is another kind of constraints to be exploited which corresponds to the notion of locks in Petri-net. That is if a set of locations is not reached, it will be not reached forever. By combining the these two kinds of constraints, we can detect local deadlocks of systems.
- Although the experimental results already shows the scalable capacity of the methods, we still want to increase the scalable capacity by using other techniques for generating components invariants, by developing connection with Z3 SMT Solver or by considering compositional abstraction methods.

Publications

Journals

1. *Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen and Joseph Sifakis*: **Compositional Verification for Component-based Systems and Application**. IET Software Journal, Special Issue on Automated Compositional Verification: Techniques, Applications, and Empirical Studies, 2010.

2. *Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul and Thanh-Hung Nguyen*: **Designing autonomous robots**, Special Issue on Software Engineering for Robotics of the IEEE Robotics and Automation Magazine Vol. 16, No 1, Pages 67-77 (03/2009).

International Conferences

3. *Saddek Bensalem, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis and Rongjie Yan*: **Incremental Invariant Generation for Compositional Design**. TASE'10, 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, 2010.

4. *Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen and Joseph Sifakis*: **D-Finder: A Tool for Compositional Deadlock Detection and Verification**. In CAV'09 21st International Conference on Computer Aided Verification, June 26 - July 02, 2009, Grenoble, France.

5. *Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen and Joseph Sifakis*: **Compositional Verification for Component-based Systems and Application**. In ATVA'08 6th International Symposium on Automated Technology for Verification and Analysis, October 20-23, 2008, Seoul, S. Korea.

6. *Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand and Joseph Sifakis*: **Incremental Component-based Construction and Verification of a Robotic System**. In ECAI'08 18th European Conference on Artificial Intelligence, July 21-25, 2008, Patras, Greece.

Under Review

7. *Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis and Rongjie Yan*: **Incremental Component-based Construction and Verification using Invariants**. Submitted to FMCAD 2010.

Workshops

8. *Jan Olaf Blech, Thanh-Hung Nguyen and Michael Perin*: **Invariants and Robustness of BIP Models**. WING'09 2nd International Workshop on Invariant Generation, March 29, 2009, York, UK.

9. *Ananda Basu , Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand and Joseph Sifakis*, **Incremental Component-Based Construction and Verification of a Robotic System**. IROS'08 International Workshop on Current Software Frameworks in Cognitive Robotics integrating different computational paradigms, September 22nd, 2008, Nice, France.

10. *Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen and Joseph Sifakis*, **Compositional Deadlock Detection and Verification for Component-based Systems.** In 2nd International Workshop on Verification and Evaluation of Computer and Communication Systems, July 2-3, Leeds, UK, 2008.

11. *Matthieu Gallien, Fahmi Gargouri, Imen Kahloul, Moez Krichen, Thanh-Hung Nguyen, Saddek Bensalem and Félix Ingrand*: **D'une approche modulaire à une approche orientée composant pour le développement de systèmes autonomes : Défis et principes**. CAR'08 The 3rd National Workshop on Control Architectures of Robots, May 29 - 30, 2008, Bourges, France.

12. *Ananda Basu, Saddek Bensalem, Félix Ingrand, Matthieu Gallien, Thanh-Hung Nguyen and Joseph Sifakis*: **Modular Architecture for Autonomous Systems**, 2007 International Advanced Robotics Program (IARP). International Workshop on Technical Challenges for Dependable Robots in Human Environnements, April 14-15 2007, Rome, Italy.

Bibliography

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine, *The algorithmic analysis of hybrid systems*, TCS **138** (1995), no. 1, 3–34.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever, *A proof system for communicating sequential processes*, ACM Trans. Program. Lang. Syst. **2** (1980), no. 3, 359–385.

[Bas08] Ananda Basu, *Component-based Modeling of Heterogeneous Real-Time Systems in BIP*, Ph.D. thesis, Université Joseph Fourier, 2008.

[BBL+09] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan, *Incremental component-based construction and verification using invariants*, Tech. Report TR-2009-12, Verimag Research Report, 2009.

[BBNS08] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis, *Compositional verification for component-based systems and application*, ATVA, 2008, pp. 64–79.

[BBNS09] ———, *D-finder: A tool for compositional deadlock detection and verification*, CAV, 2009, pp. 614–619.

[BBNS10] ———, *Compositional verification for component-based systems and application*, Special Issue on Automated Compositional Verification: Techniques, Applications, and Empirical Studies. (2010), accepted for publication.

[BBS06] A. Basu, M. Bozga, and J. Sifakis, *Modeling heterogeneous real-time components in BIP*, International Conference on Software Engineering and Formal Methods (SEFM) (Pune, India), 2006.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu, *Symbolic model checking without bdds*, TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (London, UK), Springer-Verlag, 1999, pp. 193–207.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long, *Symbolic model checking with partitioned transition relations*, In International Conference on Very Large Scale Integration, North-Holland, 1991, pp. 49–58.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Proceedings of the 5th Syposium on Logic in Computer science, 1990, pp. 428–439.

[Bei90] Boris Beizer, *Software testing techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BGI+09] Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul, and Thanh-Hung Nguyen, *Designing autonomous robots*, Robotics Automation Magazine, IEEE **16** (2009), no. 1, 67 –77.

[BGL+08] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis, *Incremental component-based construction and verification of a robotic system*, Proceeding of the 2008 conference on ECAI 2008 (Amsterdam, The Netherlands, The Netherlands), IOS Press, 2008, pp. 631–635.

[BIL09] Marius Bozga, Radu Iosif, and Yassine Lakhnech, *Flat parametric counter automata*, Fundam. Inf. **91** (2009), no. 2, 275–303.

[BJS09] Marius Bozga, Mohamad Jaber, and Joseph Sifakis, *Source-to-source architecture transformation for performance optimization in bip*, SIES, 2009, pp. 152–160.

[BL99] S. Bensalem and Y. Lakhnech, *Automatic generation of invariants*, FMSD **15** (1999), no. 1, 75–92.

[BLN+10] Saddek Bensalem, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan, *Incremental invariant generation for compositional design*, Tech. Report TR-2010-06, Verimag Research Report, 2010.

[BLO98a] S. Bensalem, Y. Lakhnech, and S. Owre, *Computing abstractions of infinite state systems automatically and compositionally*, CAV'98, LNCS, vol. 1427, 1998, pp. 319–331.

[BLO98b] _____, *Invest: A tool for the verification of invariants*, CAV'98, LNCS, vol. 1427, 1998, pp. 505–510.

[BM79] Robert S. Boyer and J. Strother Moore, *A computational logic*, Academic Press, New York, 1979.

[BM88] ———, *A computational logic handbook*, Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[BM07] Aaron R. Bradley and Zohar Manna, *Checking safety by inductive generalization of counterexamples to induction*, FMCAD, 2007, pp. 173–180.

[CAC08] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, *Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning*, ACM Transactions on Software Engineering and Methodology **17** (2008), no. 2, 1–52.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu, *Bounded model checking using satisfiability solving*, Form. Methods Syst. Des. **19** (2001), no. 1, 7–34.

[CC77] P. Cousot and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, 4th ACM symp. of Prog. Lang., ACM Press, 1977, pp. 238–252.

[CE81] E. M. Clarke and E. A. Emerson, *Synthesis of synchronisation skeletons for branching time temporal logic*, Workshop on Logic of Programs 1981 (D. Kozen, ed.), LNCS, vol. 131, Springer-Verlag, 1981.

[CEFH01] M.R.V Chaudron, E.M. Eskenazi, A.V. Fioukov, and D.K. Hammer, *A framework for formal component-based software architecting*, OOPSLA Specification and Verification of Component-Based Systems Workshop, 2001, pp. 73–80.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long, *Model checking and abstraction*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 5, 1512 – 1542.

[CGP99] Edmund M. Clarke, Orna Grumberg, , and Doron A. Peled, *Model checking*, The MIT Press, 1999.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu, *Learning assumptions for compositional verification*, TACAS, 2003, pp. 331–346.

[CJ98] Hubert Comon and Yan Jurski, *Multiple counters automata, safety analysis and Presburger arithmetic*, CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification (London, UK), Springer-Verlag, 1998, pp. 268–279.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan, *Compositional model checking*, Proceedings of the 4th Annual Symposium on LICS, IEEE Computer Society Press, 1989, pp. 353–362.

[CMP94] E. Chang, Z. Manna, and A. Pnueli, *Compositional verification of real-time systems*, Symposium on Logic in Computer Science, IEEE, 1994.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis, *Translating aadl into bip - application to the verification of real-time systems*, MoDELS Workshops, 2008, pp. 5–19.

[DF95] J. Dingel and Th. Filkorn, *Model checking for infinite state systems using data abstraction*, Computer Aided Verification (P. Wolper, ed.), LNCS, vol. 939, Springer-Verlag, 1995, pp. 54–69.

[dRdBH+00] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers, *Concurrency verification: Introduction to compositional and noncompositional methods.*, Cambridge University Press., New York, NY, USA, 2000.

[FFMR07] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier, *A compositional testing framework driven by partial specifications*, TestCom/FATES, 2007, pp. 107–122.

[GH93] John V. Guttag and James J. Horning, *Larch: languages and tools for formal specification*, Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GM93] Michael Gordon and T. Melham, *Introduction to HOL: a theorem proving environment for higher-order logic*, Cambridge University Press, 1993.

[God91] Patrice Godefroid, *Using partial orders to improve automatic verification methods*, CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification (London, UK), Springer-Verlag, 1991, pp. 176–185.

[GPB02] Dimitra Giannakopoulou, Corina S. P"s"reanu, and Howard Barringer, *Assumption generation for software component verification*, ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering (Washington, DC, USA), IEEE Computer Society, 2002, p. 3.

[GS05] Gregor Gössler and Joseph Sifakis, *Composition for component-based modeling*, Sci. Comput. Program. **55** (2005), no. 1-3, 161–183.

[GW92] Patrice Godefroid and Pierre Wolper, *Using partial orders for the efficient verification of deadlock freedom and safety properties*, CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification (London, UK), Springer-Verlag, 1992, pp. 332–342.

[HL85] David Heimbold and David Luckham, *Debugging Ada tasking programs*, IEEE Softw. **2** (1985), no. 2, 47–57.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani, *You assume, we guarantee: Methodology and case studies*, CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification (London, UK), Springer-Verlag, 1998, pp. 440–451.

[Jon83] Cliff B. Jones, *Specification and design of (parallel) programs*, IFIP Congress, 1983, pp. 321–332.

[Lam74] Leslie Lamport, *A new solution of dijkstra's concurrent programming problem*, Commun. ACM **17** (1974), no. 8, 453–455.

[LBBO01] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre, *Incremental verification by abstraction*, TACAS, 2001, pp. 98–112.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem, *Property preserving abstractions for the verification of concurrent systems*, Formal Methods in System Design **6** (1995), no. 1, 11 – 44.

[Lon93] D. E. Long, *Model checking, abstraction, and compositional reasoning*, Ph.D. thesis, Carnegie Mellon, 1993.

[MC81] J. Misra and K. M. Chandy, *Proofs of networks of processes*, IEEE Trans. Softw. Eng. **7** (1981), no. 4, 417–426.

[McM93] K.L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, Boston, 1993.

[MP95] Zohar Manna and Amir Pnueli, *Temporal verification of reactive systems: safety*, Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Mye04] Glenford J. Myers, *The art of software testing, second edition*, 2 ed., Wiley, June 2004.

[ORR+96] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas, *PVS: Combining specification, proof checking, and model checking*, CAV, lncs, vol. 1102, Springer, 1996, pp. 411–414.

[Pel94] D. Peled, *Combining partial order reductions with on-the-fly model-checking*, Computer Aided Verification, LNCS, vol. 818, 1994, pp. 377–390.

[Pnu85] A. Pnueli, *In transition from global to modular temporal reasoning about programs*, Logics and models of concurrent systems **F13** (1985), 123–144.

[PPRS06] Marc Poulhiès, Jacques Pulou, Christophe Rippert, and Joseph Sifakis, *A methodology and supporting tools for the development of component-based embedded systems*, Monterey Workshop, 2006, pp. 75–96.

[QS82] J. P. Queille and J. Sifakis, *Specification and verification of concurrent systems in CESAR*, Proc. 5th Int. Sym. on Programming, Lecture Notes in Computer Science, vol. 137, Springer-Verlag, 1982, pp. 337–351.

[Sha98] Natarajan Shankar, *Lazy compositional verification*, COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference (London, UK), Springer-Verlag, 1998, pp. 541–564.

[Tre90] Jan Tretmans, *Test case derivation from lotos specifications*, FORTE '89: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (Amsterdam, The Netherlands), North-Holland Publishing Co., 1990, pp. 345 – 359.

[TSL+90] Herv J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, , and Alberto Sangiovanni-Vincentelli, *Implicit state enumeration of finite state machines using bdd's*, IEEE International Conference Computer-Aided Design, 1990, pp. 130 – 133.

[UP83] Zerksis D. Umrigar and Vijay Pitchumani, *Formal verification of a real-time hardware design*, DAC '83: Proceedings of the 20th Design Automation Conference (Piscataway, NJ, USA), IEEE Press, 1983, pp. 221–227.