www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D5.4: Fourth Report on WP5

Verification Techniques for SCs and SCEs (final version)

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 3.0 (29.4.2014)**

Lead contractor for deliverable: **UJF-VERIMAG**
Author(s): **Saddek Bensalem (UJF-Verimag), Jacques Combaz (UJF-Verimag), Lăcrămioara Aştefănoaei (UJF-Verimag), Ayoub Nouri (UJF-Verimag)**

Reporting Period: **4**
Period covered: **October 1, 2013 to March 31, 2015**
Submission date: **March 12, 2015**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

This document summarizes the work performed in Year 4 concerning the design and the implementation of correct service components (SCs) and service component ensembles (SCEs). Among others, two main contributions that both represent significant steps towards the objectives fixed for WP5 are presented in this deliverable. The first one bridges the gap between WP1 and WP5 by providing translation means of SCEL specifications into BIP models, for which various analysis tools exist. The second main contribution is an extension of the compositional verification method proposed during the project to parameterized systems, that is, systems consisting of arbitrary number of isomorphic components communicating through predefined topologies (e.g. star or ring). We identified a specific class of target properties for which the verification of a parameterized system can be achieved through the verification of a limited number of instances. This allows us to verify whole classes of systems instead of single instances.

Contents

1 Introduction **5**

2 From SCEL to BIP: towards verification of SCEL programs **6**
2.1 Presentation of SCELight 6
2.2 Translation Rules 7
2.3 Example 14
2.4 Annex: Additional Translation Rules 17

3 Compositional Verification of Timed Systems **18**
3.1 The Compositional Verification Approach 19
3.2 Handling of "Untimed" Components 22
3.3 Compositional Verification of Parameterized Timed Systems 24
3.4 Case Study 27

4 Stochastic Abstraction **29**
4.1 Preliminaries 29
4.2 Learning-based Abstraction 30

5 Conclusion **33**

1 Introduction

This year we completed previous work on correctness of Service Component Ensembles (SCEs) in three main directions.

First, we completed the ASCENS Design Flow presented in Deliverable JD2.2 by bridging the gap between the SCEL language which is used for specifying ensembles in the project, and the BIP framework which includes tools for simulation, verification and statistical model-checking. This materializes as a prototype tool translating a subset of the SCEL language called SCELight [DLLL+14] which is a static version of SCEL in which higher-order communication, dynamic creation of new names and components, as well as policies are not considered, and such that knowledge repositories are tuple spaces. In principle, using this translator allows to apply all the verification results related to the BIP language to SCELight specifications. This includes compositional verification, which is an original approach for verifying safety properties of component-based systems, that does not require to build explicitly composed systems which are often non tractable. Section 2 defines formally the set of translation rules that are implemented by our translator. It also illustrates the translation by considering an academic example expressed in SCEL which was verified by BIP tools after translation.

Second, we improved the compositional verification method for timed systems proposed previously (see Deliverables D5.2 and D5.3) in two different ways (see Section 3). On the one hand we developed specific techniques for computing invariants for "untimed" components, that is, components that are free of timing constraints. Without these new techniques, our method was not always scalable in presence of untimed components: they may have very large zone graphs, and corresponding formulae cannot be handled by SMT solvers we are based on. The new method uses regular expressions instead of zone graphs for the untimed components, which prove to considerably reduce the size of the computed invariants for these components, as shown for the Fischer protocol example considered in this deliverable.

Another amelioration of the compositional verification method is an attempt to extend it to parameterized systems. Here, a parameterized system refers to a system built as the composition of n instances of the same component, where n is left as a parameter. Connections between the components should follow regular patterns such as star topologies which are considered here. Given a parameterized system, the goal is to come up with a proof of its correctness for any number of instances, that is, for any value of n. We shown that for a specific class of target properties which is quite expressive, it is sufficient to verify the correctness of the system for a finite number of values for n, to establish the correctness for any value of n. In practice we had to verify the system only for few values of n, which we did using compositional verification. We think that this result is a significant step towards the verification of SCEs which often consist in a large and non fixed number of replicated components (e.g. swarms of robots).

Finally, to improve general applicability and scalability of statistical model-checking (SMC) approaches (considered in the ASCENS project for quantitative and performance analysis), we developed automatic methods for the construction of faithful abstractions of system models. Our approach is based on a combination of abstraction and learning techniques. Given a target property and a set of execution traces of the system, we first use abstraction to restrict the amount of visible information on traces to the minimum required to evaluate the property and then, use learning to construct a compact, probabilistic model which conforms to the abstracted sample set. This allows to apply SMC to black-box implementations, that is, even if no detailed model of the system is available.

The rest of the deliverable is structured as follows. In Section 2 we present the translation from SCELight to BIP. Section 3 reports on the improvements achieved for the compositional verification method. Finally, Section 4 explains how we compute stochastic abstractions from black-box implementations, and Section 5 concludes the deliverable.

2 From SCEL to BIP: towards verification of SCEL programs

SCEL is the language proposed in the ASCENS project for specifying software architectures of autonomic systems (see D1.1, D1.2, and D1.3). It has been extensively used in the project across various examples (see JD2.1 and JD2.2) and case studies (see D7.2 and D7.3). SCEL specifications can be executed by means of jResp which is a runtime environment for Java that provides API for using SCEL syntactic constructs.

A lot of work in WP5 for correctness of components had focused on the BIP language instead of SCEL. The main reason for that is that many of the techniques and associated tools developed under WP5 are continuation of previous work around BIP. For a better integration of them within the project we propose to establish the connection between SCEL and BIP in the form of an automatic method for translating a subset of the SCEL language into BIP. This method has been fully implemented by a prototype tool and validated on several examples.

The rest of the section is as follows. Section 2.1 gives a definition of SCELight—the subset of SCEL that we target in this work. Section 2.2 formally specifies a set of transformation rules allowing the translation of SCELight to BIP. Finally, in Section 2.3 we briefly describe the prototype tool implementing these rules and give an example of use of our tool.

2.1 Presentation of SCELight

We restricted our transformation method to a subset of the SCEL language called SCELight. SCELight has already been considered by De Nicola et al. when they used the model-checker SPIN to verify SCEL specifications [DLLL+14]. To this end [DLLL+14] they provided transformation rules allowing systematic generation of Promela[1] specifications from SCELight ones. The main restrictions considered in SCELight compared to SCEL are the following [DLLL+14]:

- policies are not part of SCELight (composition of component's processes uses the standard interleaving of actions)
- knowledge repositories are implemented as multiple distributed tuple-spaces
- higher-order communication, dynamic creation of new names and components are not considered in SCELight.

An abstract syntax for SCELight is provided by Figure 1. It corresponds to a simplified version of the concrete syntax of the actual SCELight language. Notice also that it is not refining non terminal terms $Type$, $Expression$ and $Predicate$, since we do not provide any detail of their translation in the present document. A SCELight specification consists in a set of definitions and a system. To simplify the presentation we consider only definitions of *attributes*, but SCELight also allows to define processes (with parameters), projections, constants, and functions. Attributes are special variables used to represent interfaces of components in the system. A system is a set of (parallel) *components* $\mathcal{C}_1, \ldots, \mathcal{C}_N$, each component $\mathcal{C}_i = \{\!|\mathcal{I}|\!\}\ [\!|\mathcal{K}\ \mathcal{P}_1 \mid \ldots \mid \mathcal{P}_m|\!]$ being described by its *interface* $\mathcal{I}$, its *knowledge repository* $\mathcal{K}$, and a set of *processes* $\mathcal{P}_1, \ldots, \mathcal{P}_m$ implementing the behavior of $\mathcal{C}_i$ and executing in parallel.

The interface of a component $\mathcal{C}_i$ assigns values the attributes, which are either constant or computed dynamically from the content of the knowledge repository of $\mathcal{C}_i$. The knowledge repository of a component is a tuple-space which is initialized with a predefined (possibly empty) set of tuples. Processes define the behavior of components, that is, how tuple-spaces evolve over time. A process executes a block of *commands* sequentially. A command is either a variable declaration, an action

[1] Promela is the input language for the tool SPIN.

on knowledge repositories, or a control-flow instruction "if-then-else" or "while" loop. In addition to those three types of commands, SCELight also allows instantiation of new processes but this is not considered here. Actions performed by processes on tuple-spaces are of four types, whose semantics is informally presented below. A formal definition of the operational semantics of SCEL programs is given in ASCENS Deliverable D1.2.

Put actions are used to add tuples to tuple-spaces. They specify a tuple to be added, and the destination targeted by the action. The destination can be **`self`** for the tuple-space of the component to which the process belong, an expression evaluating to an identifier of a component, or a predicate on attributes. Notice that in the later case the tuple will be added (broadcast) to all the components satisfying the predicate, i.e. to all knowledge repositories such that the current values of the attributes of the corresponding component satisfy the predicate.

Get actions are used to retrieve tuples from tuple-spaces. They specify a template and a target destination. A get operation retrieve (non-deterministically) a single tuple from a single tuple-space, even if several tuples match the template and/or several tuple-spaces match the target destination (which happens only if a predicate is used for the destination). A get operation is blocking if no matching tuple or tuple-space is found.

Query actions are the same as get actions except for the fact that they are not retrieving tuples, they only check for their presence. They are also blocking when no tuple matching both the template and the destination can be found.

Replace actions are used to modify existing tuples. They can only target destination **`self`**, thus for replace actions the destination parameter is omitted.

2.2 Translation Rules

In the following we provide a formalization of the translation rules we used for implementing our prototype translator of SCELight to BIP. To simplify the presentation, in this deliverable we are not considering replace actions as well as predicates for target destinations of actions, even those syntactic constructions are fully handled in our prototype. Moreover, the translation of expressions is not detailed here.

The translation of SCELight specifications is formalized by function $\langle\!\langle \cdot \rangle\!\rangle$, which takes syntactic elements of SCELight as input parameter and which returns BIP code. The function $\langle\!\langle \cdot \rangle\!\rangle$ is defined recursively by a set of translation rules. Notice that for convenience we may consider additional (contextual) parameters for $\langle\!\langle \cdot \rangle\!\rangle$ which are denoted by indices, e.g. $\langle\!\langle\ e\ \rangle\!\rangle_{p_1,p_2}$ represents the translation of a SCELight element e into BIP code with respect to parameters p_1 and p_2.

The input of the translation is a SCELight specification $< \mathcal{D}, \mathcal{S} >$ where $\mathcal{D}$ is a set of attribute declarations $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and $\mathcal{S}$ is a system. The translation rules presented below allow the translation of any system $\mathcal{S}$, with respect to a set of attribute declarations $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and a given bounded capacity of K for knowledge repositories. That is, the translation of $\mathcal{S}$ for $\mathcal{A}_1, \ldots, \mathcal{A}_n, K$ is formally defined as $\langle\!\langle\ \mathcal{S}\ \rangle\!\rangle_{\mathcal{A}_1,\ldots,\mathcal{A}_n,K}$. To avoid overloading of notations, parameters $\mathcal{A}_1, \ldots, \mathcal{A}_n, K$ are omited throughout the deliverable and are considered implicitly, e.g. we write $\langle\!\langle\ \mathcal{S}\ \rangle\!\rangle$ for $\langle\!\langle\ \mathcal{S}\ \rangle\!\rangle_{\mathcal{A}_1,\ldots,\mathcal{A}_n,K}$. Attribute definitions $\mathcal{A}_i$ are of the form $\mathcal{A}_i = \mathcal{AN}_i : \mathcal{AT}_i$, where $\mathcal{AN}_i$ is the attribute name $\mathcal{A}_i$ and $\mathcal{AT}_i$ its type.

System

A SCELight system $\mathcal{S} = \mathcal{C}_1 \ldots \mathcal{C}_N$ consisting in components $\mathcal{C}_i$ = **{** $\mathcal{I}_i$ **}** **[** $\mathcal{K}_i$ $\mathcal{P}_{i,1}$ **|** $\ldots$ **|** $\mathcal{P}_{i,m(i)}$ **]**, $i = 1..N$, is translated into a BIP package which is a collection of declarations of port types, connector

$$
\begin{array}{rcl}
Definitions & ::= & Attribute^{*} \\
AttributeDecl & ::= & AttributeName\,\texttt{:}\,AttributeType \\
System & ::= & Component^{*} \\
Component & ::= & Interface\,\texttt{[}\,Knowlege\;Process(\,\texttt{|}\,Process)^{*}\,\texttt{]} \\
Interface & ::= & \texttt{\{}\,AttributeInstantiation^{*}\,\texttt{\}} \\
Knowledge & ::= & Tuple^{*} \\
Process & ::= & VariableDeclaration^{*}\;Block \\
AttributeInstantiation & ::= & AttributeName\,\texttt{=}\,(\;Expression \mid Projection\;) \\
Projection & ::= & \texttt{[}\,Template\,\texttt{]->}\,Expression\,\texttt{:}\,Expression \\
Command & ::= & Block \mid Action \mid IfThenElse \mid While \\
Block & ::= & \texttt{\{}\;Command^{*}\;\texttt{\}} \\
Action & ::= & \texttt{put(}\,Tuple\,\texttt{)@}\,Target \mid \texttt{get(}\,Template\,\texttt{)@}\,Target \\
 & \mid & \texttt{query(}\,Template\,\texttt{)@}\,Target \mid \texttt{replace[}\,Template\texttt{->}\,Tuple\,\texttt{]} \\
Tuple & ::= & Expression\;(\;\texttt{,}\,Expression\;)^{*} \\
Template & ::= & TemplateField\;(\;\texttt{,}\,TemplateField\;)^{*} \\
TemplateField & ::= & Expression \mid \texttt{*} \mid \texttt{?}\,Type \mid \texttt{?}\,VariableName \\
Target & ::= & \texttt{self} \mid Expression \mid Predicate \\
IfThenElse & ::= & \texttt{if(}\,Expression\,\texttt{)}\;Command\;(\;\texttt{else}\,Command\;)? \\
While & ::= & \texttt{while(}\,Expression\,\texttt{)}\;Command
\end{array}
$$

Figure 1: Abstract syntax of SCELight.

```
⟪ C_1 ... C_N ⟫ = package PackageSCEL2BIP
                  // declarations of external C++ code
                  use TupleTemplate

                  // generic types
                  port type ItfP(int id, ⟪ A_1 ⟫, ..., ⟪ A_n ⟫)
                  port type WriteP(Tuple t, int at)
                  port type ReadP(Template tp, Tuple t, int at)
                  port type KnowP(Tuple t)
                  port type KnowAllP(Tuple t_1, ..., Tuple t_K)

                  connector type Write(WriteP p, ItfP i, KnowP k)
                    define p i k
                    on p i k provided (i.id == p.at)
                      down { k.t = p.t; }
                  end

                  connector type Read(ReadP p, ItfP i, KnowP k)
                    define p i k
                    on p i k provided (i.id == p.at && match(p.tp, k.p))
                      down { p.t = k.t; }
                  end

                  // specific types for C_1, ...C_N
                  ⟪ C_1 ⟫_1
                  ...
                  ⟪ C_N ⟫_N

                  // component type representing S
                  compound type System()
                    // components for interfaces, tuple spaces and processes
                    ∀i = 1..N component Interface_i itf_i()
                    ∀i = 1..N component Knowledge_i knw_i()
                    ∀i = 1..N, ∀j = 1..m(i) component Process_{i,j} proc_{i,j}()

                    // connectors for updating attributes
                    ∀i = 1..N connector Update_i upt_i(knw_i.qryAll, itf_i.update)

                    // connectors for actions (put, get, query)
                    ∀i = 1..N, ∀j = 1..m(i), ∀k = 1..N
                      connector Write put_{i,j,k}(proc_{i,j}.put, itf_k.attr, knw_k.put)
                      connector Read get_{i,j,k}(proc_{i,j}.get, itf_k.attr, knw_k.get)
                      connector Read qry_{i,j,k}(proc_{i,j}.qry, itf_k.attr, knw_k.qry)
                  end
                end
```

Figure 2: BIP package corresponding to a SCELight specification.

types and component types (see Figure 2). Amongst those types, some are independent from the definition of the target system $\mathcal{S}$ (i.e. generic types), while others are generated specifically from the components of $\mathcal{S}$.

The system $\mathcal{S}$ is represented by the (composite) component type `System` containing instances of component type and connector types. Each component $\mathcal{C}_i$ is represented in `System` by one component $\texttt{itf}_i$ for its interface $\mathcal{I}_i$, one component $\texttt{knw}_i$ for its knowledge $\mathcal{K}_i$, and one component $\texttt{proc}_{i,j}$ for each of its processes $\mathcal{P}_{i,j}$, $j = 1..m(i)$. Interface components $\texttt{itf}_i$ expose a unique identifier `id` and the attributes of the corresponding component, through a port of type `ItfP`. Each knowledge component $\texttt{knw}_i$ implements a bounded tuple space with pre-allocated tuple variables. It has ports `put`, `get` and `qry` exposing tuples of $\texttt{knw}_i$ for implementing the effect of corresponding actions in the tuple space. Components $\texttt{proc}_{i,j}$ implementing processes have also ports `put`, `get` and `qry` for implementing actions from the processes side. As a result, these ports expose necessary variables of processes such as the target destination (`at`), a tuple and a template.

Components of `System` are connected as follows. For each component $\mathcal{C}_i$, we use the connector $\texttt{upt}_i$ to update the values of the attributes stored in interface component $\texttt{itf}_i$ with respect to the content of the tuple space implemented by $\texttt{knw}_i$, each time $\texttt{knw}_i$ is accessed. Moreover, connectors $\texttt{put}_{i,j,k}$, $\texttt{get}_{i,j,k}$, and $\texttt{qry}_{i,j,k}$ are used to transmit data between processes and tuple spaces when actions are performed. More precisely, connectors $\texttt{put}_{i,j,k}$ transmit tuples from processes $\texttt{proc}_{i,j,k}$ to tuple spaces $\texttt{knw}_k$, and connectors $\texttt{get}_{i,j,k}$ and $\texttt{qry}_{i,j,k}$ transmit tuples in the reverse way. Notice that these connectors also check for correct target destinations using interface components, and that target tuples match templates provided for get and query actions.

The translation of each component $\mathcal{C}_i$ of the system $\mathcal{S}$ is simply obtained by the translation of its constituents, namely its interface, its knowledge and its processes. That is, if $\mathcal{C}_i$ is the component $\{\mathcal{I}_i\}\,[\mathcal{K}_i\ \mathcal{P}_{i,1} \mid \ldots \mid \mathcal{P}_{i,m(i)}]$ then the translation of $\mathcal{C}_i$ is given by:

$$\langle\!\langle\ \mathcal{C}_i\ \rangle\!\rangle_i = \langle\!\langle\ \{\mathcal{I}_i\}\,[\mathcal{K}_i\ \ \mathcal{P}_{i,1} \mid \ldots \mid \mathcal{P}_{i,m(i)}]\ \ \rangle\!\rangle_i = \langle\!\langle\ \mathcal{I}_i\ \rangle\!\rangle_i\ \ \langle\!\langle\ \mathcal{K}_i\ \rangle\!\rangle_i\ \ \langle\!\langle\ \mathcal{P}_{i,1}\ \rangle\!\rangle_{i,1} \ldots \langle\!\langle\ \mathcal{P}_{i,m(i)}\ \rangle\!\rangle_{i,n}$$

Parameters i and j involved in this rule are used to generate unique names.

Knowledge

Tuples and templates of SCELight are encoded and managed in BIP using externally defined data types `Tuple` and `Template` (see 2.4). The translation rules related to tuples and templates are provided in Section 2.4. The BIP component representing a knowledge repository $\mathcal{K}$ of a component is given by Figure 5. It contains K tuples $\texttt{t}_1, \ldots, \texttt{t}_K$ which are initialized with initial tuples $\mathcal{T}_1 \ldots, \mathcal{T}_m$ of $\mathcal{K}$. It exports four ports: `put`, `get` and `qry` for implementing actions put, get and query, and `qryAll` which is used for projecting all the tuples on attributes (see translation rules for interfaces). Its behavior is given by a Petri net in which places $\texttt{EMPTY}_j$ and $\texttt{FULL}_j$ are used to indicate whether tuples $\texttt{t}_j$ are already used or not, and to allow/disallow put, get and query actions for tuples $\texttt{t}_j$ through interface ports (`put`, `get` and `qry`) accordingly. Notice that `qryAll` is always possible thanks to the presence of place `UP`.

Interface

An interface $\mathcal{I}$ of a SCELight component $\mathcal{C}$ defines a set of attribute instantiations $\mathcal{AI}_j$ of the form $\mathcal{AIN}_j$=r_j, where $\mathcal{AIN}_j$ is the name of an attribute and r_j is either an expression or a projection. Interfaces are translated into BIP components storing the current values of the attributes. In addition, they also store identifiers used for implementing destination of actions. Besides components, we also generate connector types implementing projections of tuple spaces on attributes. They are used to

$\langle\!\langle\ \mathcal{K}\ \rangle\!\rangle_i = \langle\!\langle\ \mathcal{T}_1 \ldots \mathcal{T}_m\ \rangle\!\rangle_i =$

```
atom type Knowledge_i()
  data Tuple t_1, ..., t_K
  export port KnowP put_1(t_1), ..., put_K(t_K) as put
  export port KnowP get_1(t_1), ..., get_K(t_K) as get
  export port KnowP qry_1(t_1), ..., qry_K(t_K) as qry
  export port KnowAllP qryAll(t_1, ..., t_K) as qryAll

  place UP, EMPTY_1, ..., EMPTY_K, FULL_1, ..., FULL_K

  initial to UP, FULL_1, ..., FULL_m, EMPTY_{m+1}, ..., EMPTY_K
    do { ∀j = 1..m ⟪ T_j ⟫_{→t_j} }

  on qryAll from UP to UP

  ∀j = 1..K
    on put_j from EMPTY_j to FULL_j
    on get_j from FULL_j to EMPTY_j do { t_j.clear(); }
    on qry_j from FULL_j to FULL_j
end
```

Figure 3: BIP component corresponding to a knowledge repository.

update the values of the attributes after each modification of the corresponding tuple space. They are directly derived from the set of attribute instantiations $\mathcal{AIN}_j$=r_j where r_j is a projection. For instantiations $\mathcal{AIN}_j$=r_j such that r_j is an expression, we simply initialize the value of $\mathcal{AIN}_j$ to r_j since in this case $\mathcal{AIN}_j$ is independent from the content of the tuple space.

Process

A process $\mathcal{P}$ of a component is represented by a BIP component having ports for put, get, query actions. A SCELight process $\mathcal{P}$ consists of a set of variable declarations $\mathcal{V}_1, \ldots, \mathcal{V}_m$ and a block of code $\mathcal{B}$. Variable declarations are directly translated into corresponding declarations in the generated component. The behavior of $\mathcal{P}$ is an automaton defined by the set of control locations $\mathtt{L}_0, \ldots, \mathtt{L}_{\#\mathcal{B}-1}$ and a set of transitions corresponding to $\mathcal{B}$. The number of control locations required for translating $\mathcal{B}$ is denoted by $\#\mathcal{B}$ and is computed using the set of rules provided in Section 2.4. In the following we detail rules for the translation of SCELight commands into transitions. Rules are parameterized by index i defining from which control location the generated transitions should start (by default they start from location L_0).

Control Flow. For translation of blocks of commands, "if-then-else", and "while" commands, we generate transitions and control locations corresponding directly to the control flow of the SCELight code. We use internal transitions (`internal`) for implementing branching in the control flow. Such transitions are executed locally by a component independently from the others, and thus do not need to be synchronized further.

```
⟪ I ⟫_i = ⟪ AI_1 ... AI_m ⟫_i = component type Interface_i()
                                 data int id
                                 ∀j = 1..m data ⟪ AT_j ⟫ ⟪ AN_j ⟫

                                 export port ItfP attr(⟪ AN_1 ⟫, ..., ⟪ AN_n ⟫)

                                 place READY, UPDATE

                                 initial to UPDATE do {
                                   id = i;
                                   ∀j = 1..m such that r_j is an expression
                                     ⟪ AN_j ⟫ = ⟪ r_j ⟫;
                                 }
                                 on update from UPDATE to READY
                                 on itf from READY to UPDATE
                               end

                               connector type Update_i(KnowAllP qry, ItfP itf)
                                 data Template template
                                 define qry itf
                                 on qry itf down {
                                   ∀j = 1..m such that r_j is a projection [t]->e:e'
                                     itf.⟪ AIN_j ⟫ = ⟪ e' ⟫;

                                     ⟪ t ⟫_→template
                                     ∀k = 1..K
                                       if (match(template, qry.t_k)) then
                                         ⟪ t ⟫_qry.t_k→
                                         itf.⟪ AIN_j ⟫ = ⟪ e ⟫;
                                       fi
                                 }
                               end
```

Figure 4: BIP component and connector generated for an interface.

```
⟪ P ⟫_i,j = V_1 ... V_m B = atom type Process_i,j()
                             int at, id
                             data Tuple tuple
                             data Template template
                             ∀k = 1..m data ⟪ V_k ⟫

                             export port WriteP put(tuple, at)
                             export port ReadP get(tuple, template, at)
                             export port ReadP qry(tuple, template, at)

                             place L_0, ..., L_#B-1
                             initial to L_0 do { id = i; }
                             ⟪ B ⟫
                           end
```

Figure 5: BIP component corresponding to a process.

$\langle\!\langle$ **{** $c_1 c_2 \ldots c_n$ **}** $\rangle\!\rangle = \langle\!\langle$ **{** $c_1 c_2 \ldots c_n$ **}** $\rangle\!\rangle_0$

$\langle\!\langle$ **{** $c_1 c_2 \ldots c_n$ **}** $\rangle\!\rangle_i = \langle\!\langle\ c_1 c_2 \ldots c_n\ \rangle\!\rangle_i = \langle\!\langle\ c_1\ \rangle\!\rangle_i \langle\!\langle\ c_2 \ldots c_n\ \rangle\!\rangle_{i+\#c_1-1}$

$\langle\!\langle$ **if(**e**)** $c\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} provided (⟪ e ⟫)
internal from L_i to L_{i+#c} provided (!⟪ e ⟫)
⟪ c ⟫_{i+1}
```

$\langle\!\langle$ **if(**e**)** c_1 **else** $c_2\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} provided (⟪ e ⟫)
internal from L_i to L_{i+1+#c1} provided (!⟪ e ⟫)
⟪ c1 ⟫_{i+1}
⟪ c2 ⟫_{i+1+#c1}
internal from L_{i+#c1} to L_{i+#c1+#c2+1}
internal from L_{i+#c1+#c2} to L_{i+#c1+#c2+1}
```

$\langle\!\langle$ **while(**e**)** $c\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} provided (⟪ e ⟫)
internal from L_i to L_{i+1+#c} provided (!⟪ e ⟫)
⟪ c ⟫_{i+1}
internal from L_{#c} to L_i
```

Actions. Actions are encoded using a sequence of two transitions. The first transition is internal and it evaluates the parameters of the action, which are:

- the target destination, that is, an expression evaluating to an integer corresponding to the identifier of a component;
- the tuple (for put) or the template (for get and query).

When **self** is provided for the destination we use the identifier of the component to which process $\mathcal{P}$ belongs, which is found in the local variable `id`.

The second transition is synchronized with port `put`, `get`, or `qry` which exposes the value of the parameters. In addition, for get and query actions, the tuple received upon the execution is used for updating the values of the free variables involved in the template.

$\langle\!\langle$ **put(**t**)@**$e\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} do { at = ⟪ e ⟫; ⟪ t ⟫_{→tuple} }
on put from L_{i+1} to L_{i+2}
```

$\langle\!\langle$ **get(**t**)@**$e\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} do { at = ⟪ e ⟫; ⟪ t ⟫_{→template} }
on get from L_{i+1} to L_{i+2} do { ⟪ t ⟫_{tuple→} }
```

$\langle\!\langle$ **qry(**t**)@**$e\ \rangle\!\rangle_i =$
```
internal from L_i to L_{i+1} do { at = ⟪ e ⟫; ⟪ t ⟫_{→template} }
on qry from L_{i+1} to L_{i+2} do { ⟪ t ⟫_{tuple→} }
```

$\langle\!\langle$ **self** $\rangle\!\rangle$ = `id`

We give here a rough idea of how to translate action having predicates as target destinations, but we do not provide formal definitions to not over-complicate this document. Given a predicate p in a process $\mathcal{P}$, the idea is to use specific connector types $\texttt{Write}_p$ and $\texttt{Read}_p$ for p, instead of the generic `Write` and `Read`. Since $\texttt{Write}_p$ and $\texttt{Read}_p$ have access to attributes of the target component, they are responsible for the evaluation of p. For instance, $\texttt{Read}_p$ defined below will be used for actions of the form **get(**t**)@**p:

```
connector type Read_p(ReadP p, ItfP i, KnowP k)
  define p i k
  on p i k provided (⟪ p ⟫_i && match(p.tp, k.t)) do { p.t = k.t; }
end
```

where $\langle\!\langle\ p\ \rangle\!\rangle_{\texttt{i}}$ is a boolean expression corresponding to the evaluation of p with attributes exported by the interface port `i`. If in addition to attributes the predicate p involves local variables of the process

$\mathcal{P}$, we also need to export them through the port `p`, which requires the use of a dedicated port type for `p` instead of `ReadP`. In this case the boolean condition corresponding to the predicate not only involves attributes exported by the port `i` but also variables exported by `p`.

For put actions of process $\mathcal{P}$ of the form **`put(`**t**`)@`**p, where p is a predicate, the semantics of SCELight requires to atomically broadcast the tuple t to any component satisfying p, which is implemented by the connector type $\texttt{Write}_p$ provided below. Notice that the following definition enumerates all interactions potentially enabled by $\texttt{Write}_p$, corresponding to all possible subsets of components that could be a destination of the broadcast. Thanks to the semantics of broadcast in BIP, interactions selected at runtime are maximal, meaning that all the components satisfying the predicate p will receive the tuple t. In our implementation we actually use a hierarchical connector instead of the implementation proposed here for $\texttt{Write}_p$, having exactly the same behavior but being much more compact (i.e. whose size in linear with respect to the number of components).

```
connector type Write_p(WriteP p, ItfP i_1, KnowP k_1, ..., ItfP i_n, KnowP k_n)
  define p' i_1 k_1 ...i_n k_n
  on p provided (true) // do nothing

  ∀m, ∀1 ≤ j_1 < j_2 < ... < j_m ≤ n
    on p i_{j_1} k_{j_1} ...i_{j_m} k_{j_m}
    provided (⟪ p ⟫_{i_{j_1}} && ...&& ⟪ p ⟫_{i_{j_m}})
    do {
      k_{j_1}.t = p.t;
      ...
      k_{j_m}.t = p.t;
    }
end
```

2.3 Example

Our prototype translator tool has been implemented on top of the Eclipse platform using XText [XTe]. It is parsing SCELight specifications and builds intermediate representations in terms of Java instances. The translation rules presented in Section 2.2 are directly expressed as XTend templates.

To illustrate our method for translating SCELight specifications into BIP models, we consider a client-server example consisting of one client and two servers (see Figure 6). The SCELight code implementing this example is provided in Figure 7. Notice that its syntax does not correspond exactly to the one presented in Section 2.1, as we are providing here an actual program using the concrete syntax of SCELight

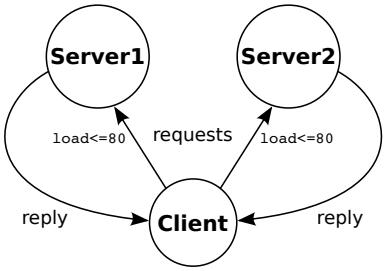


Figure 6: The client-server example.

Components have attributes `load` representing the current load of a server, and `name` to identify components. The process implementing the client repeats the following sequence of actions. It first sends a request for a service to both servers. To not overload servers, the request is sent (i.e. broadcasted) to a server only if its load is not greater than a predefined threshold (`80` here). To this end, the client puts the tuple `("invoke", "service", self)` in all repositories of components whose attributes satisfy the predicate `load <= 80 & name != "Client"`. Then, it waits for a reply from both servers, that is, it retrieves two instances of the tuple `("completed", "service")` from its own repository.

```
attribute load:int;
attribute name:string;

projection loadP = ["load", ?[int l]] -> l:0;

process Client() {
  while (true) {
    put("invoke", "service", self)@(load <= 80 & name != "Client");
    get("completed", "service")@self;
    get("completed", "service")@self;
  }
}

process Server() {
  int clientId, l;
  while(true) {
    get("invoke", "service", ?clientId)@self;
    replace["load", ?l -> "load", l+20];

    // [...]  computation on server is here

    put("completed", "service")@clientId;
    replace["load", ?l -> "load", l-20];
  }
}

system Cloud = { name="Client" } [ Client() ]
            || { name="Server1", load=loadP } [ <"load", 50>, Server() ]
            || { name="Server2", load=loadP } [ <"load", 40>, Server() ]
```

Figure 7: Client/Server example in SCELight.

A process implementing a server repeats the following sequence of actions. First, it waits for a request from the client by retrieving a tuple of the form `("invoke", "service",` id`)` from its own repository. After that, it increases temporarily its load by `20` which corresponds to the execution of the requested service. Then, it responds to the client by sending the tuple `("completed", "service")` to the destination id. Finally, it puts back its load in its initial value.

Clearly, this system executes without blocking (deadlock) if each time the client broadcasts a request, both servers are not overloaded (i.e. satisfy `load <= 80`). If (at least) one of the servers is overloaded, the client waits for responses that never arrive. To check this property, we generated

```
process FixedServer() {
  int clientId, l;
  while(true) {
    get("invoke", "service", ?clientId)@self;
    replace["load", ?l -> "load", l+20];

    // [...]  computation on server is here

    replace["load", ?l -> "load", l-20];
    put("completed", "service")@clientId;
  }
}
```

Figure 8: Fixed version of the server.

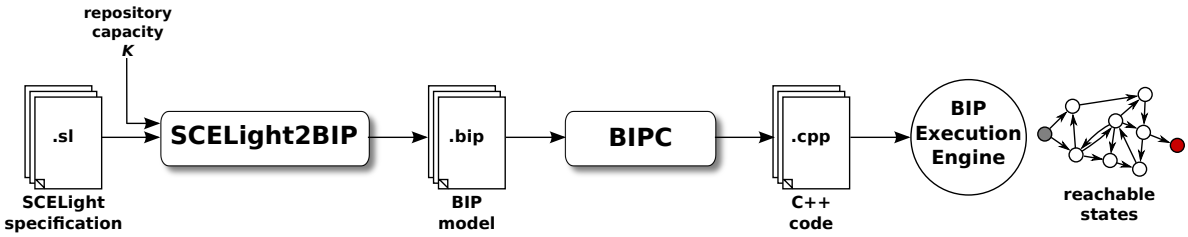


Figure 9: Example of use of the prototype SCELight2BIP.

the BIP model from the SCELight code using our prototype tool. We set a capacity of $K = 4$ for the knowledge repositories, which was sufficient for this example. We used the BIP engine in the exploration mode to compute exhaustively all the reachable states (see Figure 9), which is achieved in a reasonable amount of time for this simple example. For more complex systems and in particular infinite models, cleverer techniques such as compositional verification presented in Deliverables D5.2, D5.3 and D5.4 should be considered.

When the initial values of the attributes `load` of the two servers is not greater than `60` (which is the case in the code provided by Figure 7 with `50` and `40`), the system executes without deadlock since the load of both servers never exceeds `80`. The computation of the reachable states using the BIP engine confirms the absence of deadlocks:

```
[BIP ENGINE]: BIP Engine (version optimized )
[BIP ENGINE]:
[BIP ENGINE]: initialize components...
[BIP ENGINE]: computing reachable states:.............................
[...]... found 2365 reachable states, 0 deadlock, and 0 error in 0 state
```

If one of the initial values of attribute `load` of a server is greater than `80`, the system blocks during the first iteration of the client when it is waiting for the second reply. This deadlock situation is also confirmed by the engine with the following output when initial loads are `40` and `85`:

```
found 64 reachable states, 1 deadlock, and 0 error in 0 state
```

If one of the initial values of attribute `load` of a server is greater than `60`, the system may block if the client sends its request before servers put back their load in their initial values. Again, this is confirmed by the BIP engine for initial loads `61` and `50`:

```
found 2008 reachable states, 1 deadlock, and 0 error in 0 state
```

Such deadlock situation can be fixed by updating the load after sending the reply in the server, that is by using the implementation provided by Figure 8. This is also confirmed by the BIP engine, when considering initial loads `61` and `50` with the fixed version we have the following output:

```
found 772 reachable states, 0 deadlock, and 0 error in 0 state
```

2.4 Annex: Additional Translation Rules

We provide here additional translation rules which are needed for translating SCELight to BIP.

Number of Control Locations

The number of control locations used for representing SCELight commands is given by the operator $\#$ which is recursively defined as follows.

$\#\{c_1 c_2 \ldots c_n\} = \#c_1 + \#c_2 + \ldots + \#c_n - n + 1$

$\#(\texttt{if(}e\texttt{)}\ c) = \#c + 1$

$\#(\texttt{if(}e\texttt{)}\ c_1\ \texttt{else}\ c_2) = \#c_1 + \#c_2 + 2$

$\#(\texttt{while(}e\texttt{)}\ c) = \#c + 2$

$\#(\texttt{put(}t\texttt{)@}e) = \#(\texttt{get(}t\texttt{)@}e) = \#(\texttt{qry(}t\texttt{)@}e) = 3$

Tuples and Templates

Tuples and templates use common rules for their translation. Given a tuple or a template $\mathcal{T} = f_1, \ldots, f_n$ and a variable `t` used for recording $\mathcal{T}$, the generated code clears `t` to start from an empty tuple/template, and then add fields f_j to `t` in the right order, which is expressed by the following rule.

$\langle\!\langle\ f_1, \ldots, f_n\ \rangle\!\rangle_{\rightarrow \texttt{t}} =$ `t.clear();`
$\langle\!\langle\ f_1\ \rangle\!\rangle_{\rightarrow \texttt{t}}$
`...`
$\langle\!\langle\ f_n\ \rangle\!\rangle_{\rightarrow \texttt{t}}$

The addition of a field f_j to `t` depends on the type of f_j (whose computation is not presented here), and is done differently whether f_j is an expression e_j, $\star$ or of the form `?`v.

$\langle\!\langle\ e_j\ \rangle\!\rangle_{\rightarrow \texttt{t}} =$ `add`$Type_{e_j}$`(t,` $\langle\!\langle\ e_j\ \rangle\!\rangle$`);` `// `$Type_{e_j}$`:` name of the type of expr. e_j

$\langle\!\langle\ \texttt{*}\ \rangle\!\rangle_{\rightarrow \texttt{t}} =$ `addAny(t);`

$\langle\!\langle\ \texttt{?}v\ \rangle\!\rangle_{\rightarrow \texttt{t}} =$ `addAny`$Type_v$`(t);` `// `$Type_v$`:` name of the type of var. v

Retrieving values of free variables of templates. For templates $\mathcal{T} = f_1, \ldots, f_n$ involving free variables or wildcards (e.g. **`(10, "20", ?x)`** has the free variable **`x`**), the following set of rules are used to generate the code responsible for assigning updated values from a matching tuple `t`.

$\langle\!\langle\ f_1, \ldots, f_n\ \rangle\!\rangle_{\texttt{t}\rightarrow} =$ $\langle\!\langle\ f_1\ \rangle\!\rangle_{\texttt{t,1}\rightarrow}$
`...`
$\langle\!\langle\ f_n\ \rangle\!\rangle_{\texttt{t,n}\rightarrow}$

$\langle\!\langle\ e_j\ \rangle\!\rangle_{\texttt{t,j}\rightarrow} =$ `; // nothing to do for expressions`

$\langle\!\langle\ \texttt{*}\ \rangle\!\rangle_{\texttt{t,j}\rightarrow} =$ `; // nothing to do for *`

$\langle\!\langle\ \texttt{?}v\ \rangle\!\rangle_{\texttt{t,j}\rightarrow} = \langle\!\langle\ v\ \rangle\!\rangle =$ `get`$Type_v$`(t, j);` `// `$Type_v$`:` name of the type of var. v

External C++ Code. Functions needed for the manipulation of tuples and templates are implemented by C++ code which is called from BIP models. The following declarations are used to specify to the BIP compiler the presence such external C++ code.

```
@cpp(include="Tuple.hpp",src="Tuple.cpp")
package TupleTemplate
  extern data type Tuple
  extern data type Template

  extern function clear(Tuple)

  extern function addBool(Tuple, bool)
  extern function addInt(Tuple, int)
  extern function addDouble(Tuple, float)
  extern function addString(Tuple, string)

  extern function bool getBool(Tuple, int)
  extern function int getInt(Tuple, int)
  extern function float getFloat(Tuple, int)
  extern function string getString(Tuple, int)

  extern function clear(Template)

  extern function addAny(Template)
  extern function addAnyBool(Template)
  extern function addAnyInt(Template)
  extern function addAnyFloat(Template)
  extern function addAnyString(Template)

  extern function addBool(Template, bool)
  extern function addInt(Template, int)
  extern function addDouble(Template, float)
  extern function addString(Template, string)

  extern function bool match(Tuple, Template)
end
```

3 Compositional Verification of Timed Systems

During the project, we proposed a compositional verification method for component-based systems. Our approach relies on invariants which are state predicates satisfied by the system. The computation of these invariants is compositional, that is, they are deduced from a separate analysis of the components and the architecture which is given as a set of potential interactions between the components. The conjunction of such invariants can be understood as an over-approximation of the reachable states of the system, allowing to prove safety properties (i.e. that the system will never reach an undesirable configuration). Our method has been implemented in the tool RTD-Finder [RTD]. Due to the presence of over-approximations, it is sound but not complete: if RTD-Finder succeeds, the (safety) property is (proven to be) satisfied by the system, otherwise the tool cannot conclude, i.e. the property may or may not be satisfied by the system. In D5.2 we improved our method by computing linear invariants

for the interactions. They allow to better approximate the state space especially when trying to prove mutual exclusion properties. We also extended our approach to timed systems, which was presented in D5.3. During the last year, we improved this compositional verification approach for timed systems in two main directions.

During our past experiments, we came across a particular case for which invariants were too large to be practically handled by the SMT solver used by RTD-Finder. More specifically, the problem arises for untimed components which may be present in a timed system. The lack of local clocks leads to a too unrestricted order between the history clocks, and consequently to a considerably great number of reachable states for such components. Nevertheless, untimed components have quite a compact characterization by means of regular expressions. We will show how we can use this fact to tackle the above mentioned problem.

We also present preliminary work concerning a quite hot topic, that of the verification of parameterized systems. One of the major limitations for the application of our verification approach to Service Component Ensembles (SCEs) considered in ASCENS is that it targets only fixed architectures involving a predefined number of components. This is obviously a problem when verifying SCEs, as they may be large collections of unknown number of nodes that are dynamically changing over time. Our second contribution presents an attempt to extend our method to parameterized timed systems, which is a significant step towards the verification of SCEs. A parameterized systems consist in a fixed part interacting with an arbitrary number n of isomorphic components. We managed to show (under some conditions) that proving the target property for a limited (small) number of instances of n is sufficient for proving that any instance satisfies the property. That is, we are able to prove properties for whole classes of systems.

3.1 The Compositional Verification Approach

The compositional method proposed in [ARB+14] is based on the verification rule (VR) from [BBSN08]. Assume that a system consists of n components B_i interacting by means of an interaction set γ, and that the system property of interest is Ψ. If components B_i, respectively interactions γ, can be locally characterized by means of invariants $CI(B_i)$, respectively $II(\gamma)$, and if Ψ can be proved to be a logical consequence of the conjunction of the local invariants, then Ψ is a global invariant. This is what the rule below summarizes.

$$\frac{\vdash \bigwedge_i CI(B_i) \wedge II(\gamma) \rightarrow \Psi}{\|_\gamma B_i \vDash \Box\ \Psi} \text{ (VR)}$$

In the rule (VR), the symbol $\vdash$ is used to underline that the logical implication can be effectively proved (for instance with an SMT solver) and the notation $B \vDash \Box\ \Psi$ is to be read as "Ψ holds in every reachable state of B".

Timed Systems. Timed systems are compositions of timed automata [AD94] with respect to n-ary interactions. Timed automata represent the behavior of components. They have control locations and transitions between these locations. Transitions may have timing constraints, which are defined on clocks. Clocks can be reset and/or tested along with transition execution. Formally, a timed automaton is tuple $(L, l_0, A, T, X, \mathsf{tpc})$ where L is a finite set of control locations, l_0 is an initial control location, A a finite set of actions, X is a finite set of clocks, $T \subseteq L \times (A \times \mathcal{C} \times 2^X) \times L$ is finite set of transitions labelled with actions, guards, and a subset of clocks to be reset, and $\mathsf{tpc} : L \rightarrow \mathcal{C}$ assigns a time progress condition to each location. $\mathcal{C}$ is the set of timing constraints which are predicates on the clocks X defined by the following grammar:

$$C ::= true \mid false \mid x \# ct \mid x - y \# ct \mid C \wedge C$$

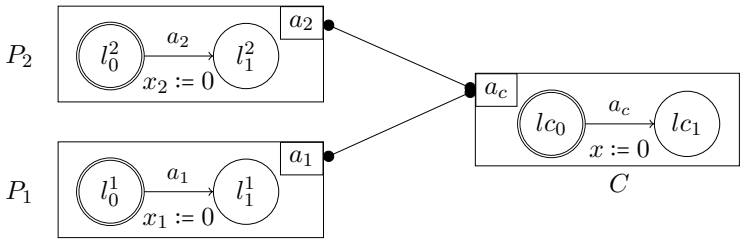


Figure 10: An Example of a Timed System

with $x, y \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $ct \in \mathbb{Z}$. Time progress conditions are restricted to conjunctions of constraints as $x \leq ct$. For simplicity, we assume that at each location l the guards of the outgoing transitions imply the time progress condition $\mathtt{tpc}(l)$ of l. For more details about timed automata the reader may refer to D5.3 and [AD94].

Examples of timed automata are provided by Figure 10. For instance, components P_i, $i \in \{1, 2\}$, are implemented by similar timed automata, consisting of two control locations l_0^i and l_1^i and one transition from l_0^i to l_1^i labelled by action a_i and resetting clock x_i. By convention non displayed guards of transitions and time progress conditions of locations are $true$.

Components interact by means of strong synchronizations between their actions. The synchronizations are specified in the so called *interactions* as sets of actions. An interaction can involve at most one action of each component. Given n components (i.e. timed automata) $B_i = (L^i, l_0^i, A^i, T^i, X^i, \mathtt{tpc}^i)$, $1 \leq i \leq n$, and a set of interactions γ, we denote by $\|_\gamma B_i$ the composition of components B_i with respect to interactions γ. States of the composition $\|_\gamma B_i$ are combinations of the states of the components B_i. In $\|_\gamma B_i$, a component B_i can execute an action a_i only as part of an interaction $\alpha \in \gamma$, $a_i \in \alpha$, that is, along with the execution of all the actions participating to α, which corresponds to the usual notion of multi-party interaction. Note that for a component B_i of a composition $\|_\gamma B_i$, the application of interactions γ can only restrict its reachable states. That is, the reachable states of B_i when executed in the composition $\|_\gamma B_i$ are included in the reachable states of B_i executed alone (i.e. as a single timed automaton). This property is essential for the correctness of the compositional verification method, summarized below.

Components and Interactions Invariants. To give a logical characterization of a system $S = \|_\gamma B_i$ one uses invariants. An invariant Ψ is a state property which holds in every reachable state of S, in symbols, $S \vDash \Box\Psi$.

Component invariants $CI(B_i)$ characterize the reachable states of components B_i when considered alone. Such invariants can easily be computed from the zone graph [HNSY94] of the corresponding timed automaton. More precisely, given the reachable (symbolic) states (l_j, ζ_j), $1 \leq j \leq m$, of component B_i, the invariant for B_i is defined by:

$$\bigvee_{1 \leq j \leq m} l_j = 1 \wedge \zeta_j,$$

where by abuse of notation l_j is a variable such that $l_j = 1$ whenever B_i is at location l_j, $l_j = 0$ otherwise. Note that zones ζ_j are timing constraints, that is, predicates on clocks. Note also that invariants $CI(B_i)$ still hold for the composed system $S = \gamma(B_1, \ldots, B_n)$, but are only over approximations of the states reached by each component B_i in S. For example, the component invariants for P_1 and C of Figure 10 are as follows:

$$CI(P_1) = (l_0^1 = 1 \vee l_1^1) \wedge (l_0^1 = 1 \wedge x_1 \geq 0 \vee l_1^1 = 1 \wedge x_1 \geq 0)$$
$$CI(C) = (l_0^c = 1 \vee l_1^c) \wedge (l_0^c = 1 \wedge x_c \geq 0 \vee l_1^c = 1 \wedge x_c \geq 0)$$

Interaction invariants $II(\gamma)$ are induced by the synchronizations and have the form of global conditions involving control locations of components. In previous work, we have considered boolean conditions [BBSN08] as well as linear constraints [LBiBB13] for $II(\gamma)$. For instance, such invariants exclude configurations such that $lc_1 = 1 \wedge l_2^i = 1$, that is, they establish $\neg\big(lc_1 = 1 \wedge (l_2^1 = 1 \vee l_2^1 = 1)\big)$.

A safety property of interest for example of Figure 10 is that one of the clocks x_i has the same value as x_c, that is, $\Psi \triangleq x_1 = x_c \vee x_2 = x_c$. Even if Ψ holds in S, it cannot be proved by applying (VR) using only component invariants $CI(B_i)$ and interaction invariant $II(\gamma)$. A counterexample is given by $l_1^c = l_1^1 = l_1^2 = 1$ and $x_1 = x_2 = 1$, and $x_c = 0$, which satisfies the invariant $CI(C) \wedge CI(P_1) \wedge CI(P_2) \wedge II(\gamma)$ but violates property Ψ, that is, $CI(C) \wedge CI(P_1) \wedge CI(P_2) \wedge II(\gamma) \not\rightarrow \Psi$. The weakness comes from that the proposed invariants cannot relate values of clocks of different components according to their synchronizations (e.g. synchronous reset of clocks).

Adding History Clocks. As explained in D5.3 and [ARB+14], the key idea behind the compositional verification method for timed systems is to use additional *history clocks* in order to track the timing of interactions between different components without modifying their behavior. We equip each component B_i (and later, interactions) with *history clocks*: one clock h_{a_i} per action of a_i of B_i. A history clock h_{a_i} is reset on all transitions executing a_i. Each time an interaction $\alpha \in \gamma$ is executed, all the history clocks corresponding to the actions participating in α are reset synchronously, and then become identical at the next state (until another interaction is executed). Moreover, history clocks of actions of the last executed interaction α are necessarily lower than the ones of actions not participating in α, since they are the last being reset. This is captured by the following invariant:

$$\mathcal{E}(\gamma) = \bigvee_{\alpha \in \gamma} \Big(\big(\bigwedge_{\substack{a_i, a_j \in \alpha \\ a_k \notin \alpha}} h_{a_i} = h_{a_j} \leq h_{a_k} \big) \wedge \mathcal{E}(\gamma \ominus \alpha) \Big),$$

where $\gamma \ominus \alpha = \{\beta \setminus \alpha \mid \beta \in \gamma \wedge \beta \not\subseteq \alpha\}$. In [ARB+14] it is shown that $\mathcal{E}(\gamma)$ is an invariant of the system. For example of Figure 10, invariant $\mathcal{E}(\gamma)$ is given by:

$$\mathcal{E}(\gamma) = h_{a_c} = h_{a_1} \leq h_{a_2} \vee h_{a_c} = h_{a_2} \leq h_{a_1}.$$

Component invariants for example of Figure 10 including the history clocks are as follows:

$$CI(P_1^h) = l_0^1 + l_1^1 = 1 \wedge (l_0^1 = 1 \wedge h_{a_1} > t_\epsilon = x_1 \vee l_1^1 = 1 \wedge t_\epsilon \geq x_1 = h_{a_1})$$
$$CI(C^h) = l_0^c + l_1^c = 1 \wedge (l_0^c = 1 \wedge h_{a_c} > t_\epsilon = x_c \vee l_1^c = 1 \wedge t_\epsilon \geq x_c = h_{a_c}).$$

Handling Conflicting Interactions. Previous invariants can be further strengthened for conflicting interactions (i.e. sharing a common action a_i) by considering the minimal time elapsed between consecutive execution of a_i. We add again history clocks h_α for each the interaction α of γ, which is reset each time α is executed by the means of an additional component and adequate synchronizations. For an action a_i of component B_i, the separation constraint $\mathcal{S}(\gamma, a_i)$ is defined as:

$$\mathcal{S}(\gamma, a_i) = \bigwedge_{\substack{\alpha, \beta \in \gamma \mid a_i \in \alpha, \beta \\ \alpha \neq \beta}} \mid h_\alpha - h_\beta \mid \geq \delta_{a_i},$$

where δ_{a_i} is a lower bound of the time elapsed between two consecutive executions of a_i in B_i, which can be statically computed from the timed automata of B_i [CY92]. It is shown that separation constraints $\mathcal{S}(\gamma, a_i)$ are invariants of the system, that is, the following is an invariant of the system:

$$\mathcal{S}(\gamma) = \bigwedge_{1 \leq i \leq n} \bigwedge_{a_i \in A_i} \mathcal{S}(\gamma, a_i).$$

In [ARB+14], it is shown that a new invariant $\mathcal{E}^*(\gamma)$

$$\mathcal{E}^*(\gamma) = \bigwedge_{1\le i\le n} \bigwedge_{a_i\in A_i} h_{a_i} = \min_{\alpha\ni a_i} h_\alpha.$$

links $\mathcal{S}$ to $\mathcal{E}$. This corresponds to the intuition that the history clock of an action a_i equals the history clock of the last executed interaction α involving a_i, which is the one having h_α minimal.

3.2 Handling of "Untimed" Components

In one of our past case studies, the Fischer protocol, we were confronted with a huge graph zone for even a small number of components. Before explaining in detail the problem and the solution, we first give a brief description of the protocol itself and of the model we adopted. The Fischer protocol is a well-studied protocol for mutual exclusion [Lam87]. It specifies how n processes can share a resource one at a time by means of a shared variable to which each process assigns its own identifier number. After θ time units, the process with the id stored in the variable enters the critical state and uses the resource. We use an auxiliary component `Id Variable` to mimic the role of the shared variable. To keep the size of the generated invariants manageable, we restrict to the acyclic version. The system with two concurrent processes is represented in Figure 11. The property Ψ of interest is mutual exclusion on critical sections cs_i, $i = 1..n$, i.e. Ψ corresponds to the formula $(cs_i = 1 \wedge cs_j = 1) \rightarrow i = j$.

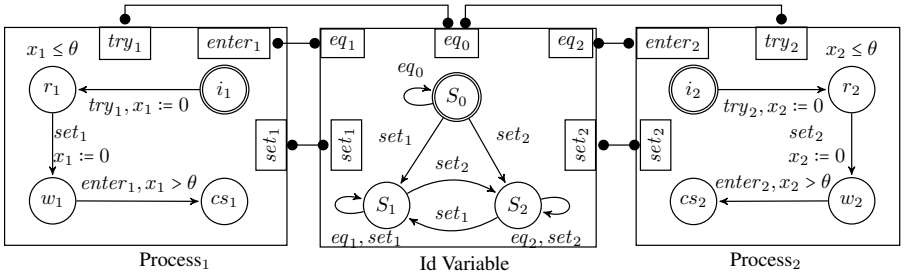


Figure 11: The Fischer Protocol

The problem we faced during this case study comes from the fact that component `Id Variable` has combinatorial behavior. This is because this component has no timing constraint, thus the constraints in the graph zone of *Id Variable*^{h} involve only history clocks. Without any timing constraint, the zone graphs become quite large when the number of processes n is also large. It turns out that the generated invariant for *Id Variable*^{h} is huge except for very small values of n. To overcome this issue, we extracted from the structure of the generated invariant a weaker inductive one which we verified for validity locally with Uppaal. Basically, it encodes information like $h_{eq_i} < h_{set_i} \rightarrow h_{set_i} < h_{eq_0}$. This invariant, together with the component invariants for the processes and $\mathcal{E}(\gamma)$ was sufficient to show that mutual exclusion holds.

Not satisfied with our provisional solution, we searched for a heuristics allowing us to treat similar cases automatically. The starting point was the elementary fact that untimed automata have elegant and concise encodings as regular expressions and this is what we will exploit in this section. More precisely, given an untimed component $B = (L, A, T)$ we show how to automatically compute an invariant describing the relations between the history clocks of B^h from the language accepted by B. The starting key observation is that only the last occurrence of each symbol should be retained at a given location. This implies that it is safe to abstract with respect to a last occurrence retention operation each regular expression characterising the language accepted at a given control location. To perform this abstraction, we rely on the simplification rules in Figure 12.

Rule 1 [Back-unfolding] : E^* $\longrightarrow E^*.E + \varepsilon$

Rule 2 [Last Occurrence Retention] : $E.a$ $\longrightarrow (E \setminus a).a$

Figure 12: Simplification Rules

In Rule 2, "$\setminus$" is what we call the *elimination operator* to eliminate a given symbol a from a regular expression. Its definition is as follows. Let a and x be two symbols and s, s_1 and s_2 strings:

$$\begin{aligned}
\epsilon \setminus a &= \epsilon \\
(s_1 + s_2) \setminus a &= (s_1 \setminus a) + (s_2 \setminus a) \\
x \setminus a &= \begin{cases} \epsilon \text{ if } x = a \\ x \text{ if } x \neq a \end{cases} \\
s^* \setminus a &= (s \setminus a)^* \\
(s_1.s_2) \setminus a &= (s_1 \setminus a).(s_2 \setminus a)
\end{aligned}$$

We consider the following strategy.

1. Choose symbols from right to left and apply Rule 2 until no longer possible (a "*" is found).
2. Apply Rule 1 and split the "+", if any.
3. Go back to 1. For each newly introduced expression and repeat until a restricted form of regular expressions is reached.

It can be shown that the above strategy terminates. Intuitively, what happens is that Rule 1 splits a large string into smaller ones and for each of these Rule 2 deletes symbols, thus makes words shorter, so, eventually it terminates.

As an example, we show the first steps of the derivations for the local invariant of the component *Id Variable*. To get the order on actions at control location S_i, $i \neq 0$, known that initially the component is at S_0, we consider the regular expression giving all the possible paths through which the automaton goes from S_0 to S_i: $\mathbf{R} = (\mathbf{a}^*\mathbf{b}\mathbf{d}^*\mathbf{c})^*\mathbf{a}^*\mathbf{b}\mathbf{d}^*$, where $i \neq j$ and:

- $\mathbf{a} = s_0 + e_0 + s_j(e_j + s_j)^* s_0$
- $\mathbf{b} = s_i + s_j(e_j + s_j)^* s_i$
- $\mathbf{c} = s_0 + s_j(e_j + s_j)^* s_0$
- $\mathbf{d} = e_i + s_i + s_j(e_j + s_j)^* s_i$

After projection on labels 0 and i:

- $\mathbf{a} = s_0 + e_0$
- $\mathbf{b} = s_i$
- $\mathbf{c} = s_0$
- $\mathbf{d} = e_i + s_i$

where s_i stands for set_i and e_i for eq_i. Using the simplification rules, we transform the regular expression $\mathbf{R}$ into $\mathbf{R'} = \mathbf{R'^1} + \mathbf{R'^2}$.

$$\begin{aligned}
\mathbf{R'^1} &= (e_0 + e_i^* s_0)^* (s_0 + e_0)^* s_i (e_i + s_i)^* (e_i + s_i) \\
&\simeq (e_0 + s_0)^* e_i.s_i + (e_0 + s_0)^* s_i.e_i && \text{(by Rule 2)} \\
\mathbf{R'^2} &= (e_0 + e_i^* s_0)^* (s_0 + e_0)^* s_i \\
&\simeq (e_0 + e_i^* s_0)^* ((s_0 + e_0)^* (s_0 + e_0) + \epsilon) s_i && \text{(by Rule 1)}
\end{aligned}$$

In the end we obtain that $\mathbf{R'} = (e_0 + s_0)^* e_i.s_i + (e_0 + s_0)^* s_i.e_i + (e_0 + e_i)^* s_0 s_i + (e_i + s_0)^* e_0 s_i + s_i$.

In the following, we assume that, by applying the above described strategy, we reach a regular expression described by the following restricted grammar:

$$ea ::= a | a + ea$$
$$e ::= ea^*.s | e + e$$

where s denotes a (possibly empty) string. As a side remark, we note that the expression $\mathbf{R'}$ is well-formed with respect to the above grammar. Next, we define a function ϕ translating such expressions into constraints.

$$\phi(e) = \begin{cases} \phi(e_1) \vee \phi(e_2) & \text{if } e = e_1 + e_2 \\ \min\limits_{a} h_a > h_{init} \geq h_{a_1} \geq h_{a_2} \geq \cdots \geq h_{a_k} & \text{if } e = a_1.a_2.\ldots.a_k \\ & \text{and } a \in A \setminus \{a_1, \ldots, a_k\} \\ \min\limits_{k} h_{a_k} \geq h_{s.head} \wedge \phi(s) \wedge h_{init} \geq h_{s.head} & \text{if } e = (a_1 + a_2 + \cdots + a_k)^*.s \end{cases}$$

It can be shown that given an untimed component B, e the regular expression characterizing the language accepted by B, and e' the resulting one after applying the strategy, $\phi(e')$ is an invariant of B^h.

As an illustration, by applying ϕ to the expression $\mathbf{R'}$ we obtained for the Fischer example and simplifying we obtain:

$$\begin{aligned} \phi = & (h_{e_0} \geq h_{e_i} \wedge h_{s_0} \geq h_{e_i} \wedge h_{e_i} \geq h_{s_i} \wedge h_{e_i} \leq h_{init}) \vee \\ & (h_{e_0} \geq h_{s_i} \wedge h_{s_0} \geq h_{s_i} \wedge h_{e_i} \leq h_{s_i} \wedge h_{s_i} \leq h_{init}) \vee \\ & (h_{e_0} \geq h_{s_0} \wedge h_{e_i} \geq h_{s_0} \wedge h_{s_0} \geq h_{s_i} \wedge h_{s_0} \leq h_{init}) \vee \\ & (h_{s_0} \geq h_{e_0} \wedge h_{e_i} \geq h_{e_0} \wedge h_{e_0} \geq h_{s_i} \wedge h_{e_0} \leq h_{init}) \vee \\ & (h_{s_i} \leq h_{init} \wedge h_{s_0}, h_{e_0}, h_{e_i} > h_{init}) \end{aligned}$$

By applying our verification method using ϕ as the component invariant of *Id Variable* we managed to prove the correctness of the Fischer protocol for up to 14 processes, which was not possible previously.

To sum up, we described an heuristic which can be applied to untimed components to automatically compute an invariant with a reasonable enough size to be handled by existing SMT solvers. Given an untimed component B, our heuristic makes use of the regular expressions characterizing the language accepted by B to avoid a direct construction of the zone graph of B^h which would result in considerably large invariants. The application of this optimization to the example of the Fischer protocol allowed us to verify instances which we could not verify with the standard compositional verification method.

3.3 Compositional Verification of Parameterized Timed Systems

In our framework, parameterized timed systems (PTSs) consist in a set of n of isomorphic components P_i and a controller component C, interacting together by means of a set of interaction γ. We note that our framework is flexible enough to consider parameterized timed systems also without controllers, however, we stick to this choice for clarity and for the ease of presentation. In what follows we adopt the notation $C \|^n_\gamma P_i$ for a PTSs with a controller C and n instances $P_1, \ldots, P_n$ of a generic component P. We prefer to use the terminology of "process" (along with the natural notation P) to refer in particular to a isomorphic component and to differentiate it from other components. Also, we use A_c, A, A_i to denote the sets of actions of the controller, of a generic process P and respectively of a process P_i.

A note on topologies. Usually in the literature on parameterized systems, the communication between components is given by the so-call network topology, which in turn is given as a graph where the vertices represent the indices of the components and the edges give the communication links [ADR+11]. In our framework, the topology is induced by the set of interactions between the controller and the processes. We restrict our setting to *binary* interactions centered in the controller and with ends in the processes. Thus, typically, the interaction sets themselves describe topologies which are variations on the *star*[2] topology, depicted in Figure 13. In fact, any interaction set γ can be seen as $\cup_{a\in A'_c}\gamma_a$ for A'_c a subset of A_c and γ_a describing the interactions of a with a fixed set of process actions, that is, $\gamma_a \subseteq \{a\} \times \cup_i A_i$.

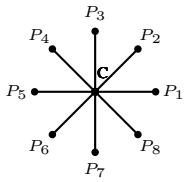


Figure 13: A Star Topology and a Possible Set of Interactions $\gamma = \{(a_c \mid a_i) \mid i \in [8]\}$.

We consider a PTS version $C\|^n_\gamma P_i$ of the example of Figure 10 (see Section 3.1) in which the controller C interacts with an arbitrary number of instances P_i according to the set of interactions $\gamma = \{(a_c \mid a_i) \mid i \in [n]\}$. Is is informally represented by Figure 14.

The application of the verification rule (VR) presented in Section 3.1 to parameterized timed systems boils down to checking the validity of the following formula:

$$\underbrace{\forall i \in [n].(CI(P_i^h) \wedge CI(C^h) \wedge II(\gamma) \wedge \mathcal{E}(\gamma) \wedge \mathcal{E}^*(\gamma) \wedge \mathcal{S}(\gamma))}_{GI} \rightarrow \Psi. \tag{1}$$

or equally the unsatisfiability of $GI \wedge \neg\Psi$.

For particular classes of target properties Ψ, such formulae can be shown to be in a decidable theory of arrays as the ones in [BMS06, GNRZ08]. Even better, they may enjoy a "small model theorem" as it is the case for the so-called LH-assertions from [JM12], that is, proving Ψ for any possible value of n is reduced to checking (1) for finitely many values of n. The formulae we work with, are, in fact, a particular[3] case of LH-assertions. We denote our working subset of LH-assertions as T-assertions. The signature of T-assertions consists of the constants 1 and n of type $\mathbb{N}$, and of a finite number of variables: (a) *index variables*: $i_1, \ldots, i_a \in \mathbb{N}$; (b) *discrete variables*: $l_1, \ldots, l_b \in L$;

[2] In a similar manner we can treat also *ring* topologies. These suit better parameterized systems without controllers and thus we leave them aside in this presentation.

[3] To be specific, by "particular" we mean that we do not need the so called "index-valued array variables" which in [JM12] model pointer variables.

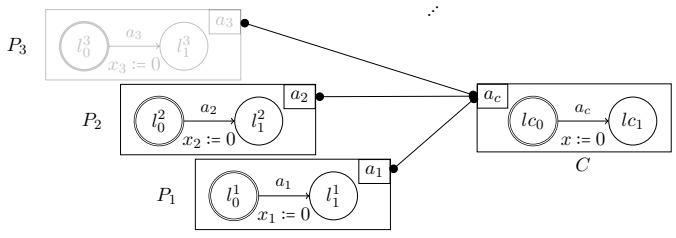


Figure 14: An Example of a Parameterized Timed System

(c) *real variables*: $x_1, \ldots, x_c \in \mathbb{R}$; (d) *discrete array variables*: $\bar{l}_1, \ldots, \bar{l}_d : [n] \to L$; (e) *real array variables*: $\bar{x}_1, \ldots, \bar{x}_e : [n] \to \mathbb{R}^+$. Terms are constructed following the BNF grammar:

$$\begin{aligned}
\texttt{ITerm} &::= 1 \mid n \mid i_j \\
\texttt{DTerm} &::= L_j \mid l_k \mid \bar{l}_j[\texttt{ITerm}] \\
\texttt{RTerm} &::= x_j \mid \bar{x}_k[\texttt{ITerm}]
\end{aligned}$$

and the formulae are structurally defined as:

$$\begin{aligned}
\texttt{Atom} &::= \texttt{ITerm} < \texttt{ITerm} \mid \texttt{DTerm} = L_k \mid a \cdot \texttt{RTerm} + b \cdot \texttt{RTerm} + c < 0 \\
\texttt{Formula} &::= \texttt{Atom} \mid \neg\texttt{Formula} \mid \texttt{Formula} \wedge \texttt{Formula}
\end{aligned}$$

where a, b, c are real values. A T-assertion is an expression

$$\forall i_1, \ldots, i_k \in [n]\ \exists j_1, \ldots, j_m \in [n].\phi$$

where ϕ is of type `Formula`. Dually, a T^c-assertion is $\exists i_1, \ldots, i_k \in [n]\ \forall j_1, \ldots, j_m \in [n].\phi$.

To illustrate the syntax of T-assertions, we show how they encode usual safety properties.

Mutual Exclusion. The T-assertion $\forall i, j.\neg(\bar{l}[i] = cs_i \wedge \bar{l}[j] = cs_j \wedge i \neq j)$ expresses mutual exclusion of processes on control locations cs_i.

Maximum Delay. The T-assertion $\forall i, j.\neg(\bar{l}[i] = \bar{l}[j] \wedge |\bar{x}[i] - \bar{x}[j]| > k_a)$ expresses a "maximal delay" of k_a between the clocks of any two components being in the same control location.

Synchronism. The safety property considered in our running example can be expressed by the T-assertion $\exists i.\bar{x}[i] = x_c$.

As already anticipated, T-assertions, as a particular case of LH-assertions, have a "small model theorem". This means that, if a formula is a T-assertion, then it is enough to check its validity for for a finite (small) number of processes. Concretely, for deciding the unsatisfiability of T-assertion Φ, the needed value is 2 + the number of universally quantified variables in Φ. Dually, for a T^c-assertion, then it is enough to check its unsatisfiablity for 2 + the number of existentially quantified variables in Φ. We will use either one of these equivalences depending on which one is handier. Next, our discussion is with respect to T^c-assertions. To be able to apply this reduction in the case of our (VR), we only need to show that $GI \wedge \Psi$ can be transformed into an equivalent T^c-assertion. For simplicity, we will only briefly give the intuition of why this is the case and instead prefer to illustrate the transformation by taking as input our running example.

The main difficulty in rewritting $GI \wedge \Psi$ as a T^c-assertion is to bring all quantifiers in front such that the existential ones follow the universal ones. We take one by one all the subformulas in GI. A first straightforward observation is that component invariants themselves are quantifier free thus they play no role in the form of GI and consequently we can ignore them in our analysis. The more problematic case is of $\mathcal{E}(\gamma)$ (those of $\mathcal{E}^*, \mathcal{S}$ are similar, but easier, so we leave them aside). At a closer look at the definition of $\mathcal{E}(\gamma)$ and recalling that the interaction sets we considered are $\cup_{a \in A'_c}$ with $\gamma_a \subseteq \{a\} \times \cup_i A_i$, it can be observed that $\mathcal{E}(\gamma)$ reduces to $\wedge_a \mathcal{E}(\gamma_a)$, because for any $\alpha \in \gamma_a$, $\gamma \ominus \alpha$ is precisely the set of remaining γ_b with $b \neq a$. In turn, each $\mathcal{E}(\gamma_a)$ is either quantifier free or have at most one existential quantifier in the case the definition of γ_a involves an arbitrary number of processes as it is the case for our running example with $\gamma_{a_c} = \{(a_c \mid a_i) \mid i \in [n]\}$. Thus the quantified fragment of $\mathcal{E}(\gamma)$ reduces to $\vee_a \exists i_a (h_a = x_{i_a})$ where all i_a are different and not related by any predicate, and can be moved in front. We assume Ψ to be a T-assertion itself, say $\forall \bar{x}\ \exists \bar{y}.\circ(\Psi)$ with $\bar{x}, \bar{y}$ disjoint of all quantified variables in GI (we can always rename them if not the case), and $\circ(\Psi)$ denoting the quantifier free part of Ψ.

This is quite a reasonable assumption and the illustrated examples of safety properties as T-assertions confirm it. All in all, it is enough to use basic equivalences like the ones below, where *op* stands for the usual logical operators:

$$\mathbf{Q}x\mathbf{Q}y.(P(x)\ op\ Q(y)) \equiv \mathbf{Q}y\mathbf{Q}x.(P(x)\ op\ Q(y))$$
$$P\ op\ \mathbf{Q}y.Q(y) \equiv \mathbf{Q}y.(P\ op\ Q(y))$$

in order to transform $\forall i GI \rightarrow \Psi$ into a T-assertion.

As an illustration, we work through the running example as shown below.

$$CI(P_i^h) = l_0^i + l_1^i = 1 \wedge (l_0^i = 1 \wedge h_{a_i} > t_\epsilon = x_i \vee l_1^i = 1 \wedge t_\epsilon \geq x_i = h_{a_i}) \tag{2}$$
$$CI(C^h) = l_0^c + l_1^c = 1 \wedge (l_0^c = 1 \wedge h_{a_c} > t_\epsilon = x_c \vee l_1^c = 1 \wedge t_\epsilon \geq x_c = h_{a_c}) \tag{3}$$

As for the interaction invariant, for the star topology in the toy example, by the linear approach after some calculations we obtain:

$$II(\gamma) = \sum_{j=1}^{n} l_0^j + l_0^c = n + 1 \vee \sum_{j=1}^{n} l_1^j + l_1^c = 2 \tag{4}$$

(or, using quantifiers)

$$\equiv (\forall j.l_0^j = 1 \wedge l_0^c = 1) \vee (l_1^c = 1 \wedge \exists j \forall k \neq j.l_1^j = 1 \wedge l_1^k = 0)$$
$$\equiv \forall j.\underbrace{(l_0^j = 1 \wedge l_0^c = 1)}_{\circ(II_1)} \vee \exists j.\underbrace{(l_1^c = 1 \wedge l_1^j = 1 > \max_k l_1^k)}_{\circ(II_2)} \tag{5}$$

The invariants $\mathcal{E}, \mathcal{S}, \mathcal{E}^*$ for the star topology are:

$$\mathcal{E}(\gamma) = \exists p.\underbrace{(h_{a_p} = h_{a_c} \leq \min_q h_{a_q})}_{\circ(\mathcal{E})} \tag{6}$$

$$\mathcal{E}^*(\gamma) = (h_{a_c} = \min_r h_{a_c|a_r}) \tag{7}$$

$$\mathcal{S}(\gamma) = \forall s, t \neq s.\underbrace{|\ h_{a_c|a_s} - h_{a_c|a_t}\ | \geq k_a}_{\circ(\mathcal{S})} \tag{8}$$

All in all, recalling that our Ψ of interest is $\exists v.x_v = x_c$, $GI \wedge \neg\Psi$ reduces to the following two formulae (as corresponding to the disjunction in Equation (5)):

$$\exists p\ \forall i, j, s, t \neq s, v.\big(CI(P_i^h) \wedge CI(C^h) \wedge \circ(II_1) \wedge \circ(\mathcal{E}) \wedge \mathcal{E}^* \wedge \circ(\mathcal{S}) \wedge x_v \neq x_c\big) \quad \vee$$
$$\exists j, p\ \forall i, s, t \neq s, v.\big(CI(P_i^h) \wedge CI(C^h) \wedge \circ(II_2) \wedge \circ(\mathcal{E}) \wedge \mathcal{E}^* \wedge \circ(\mathcal{S}) \wedge x_v \neq x_c\big)$$

which are both T^c-assertions, and thus it is enough to check the unsatisfiability of $GI \wedge \neg\Psi$ for 3, respectively 4 processes in order to decide the correctness of the system with respect to Ψ.

3.4 Case Study

Train gate controller (TGC): This is the parameterized version of the classical example from [AD94]. The system is composed of a controller, a gate and an *arbitrary* number of trains. For simplicity, Figure 15 depicts only one train interacting with the controller and the gate. The controller lowers and raises the gate when a train enters, respectively exits. The safety property of interest is that when a train is at location `in`, the gate has been lowered: $\wedge_i(in_i = 1 \rightarrow g_2 = 1)$.

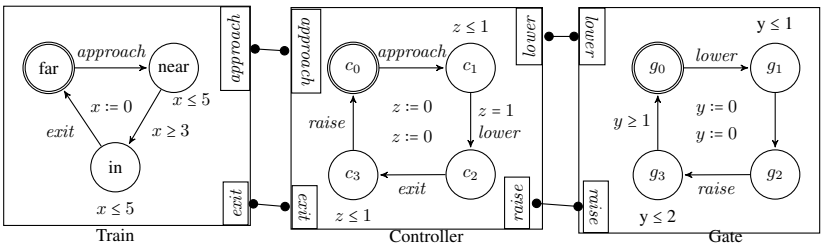


Figure 15: A Controller Interacting with a Train and a Gate

Note that as it is, the system isn't strictly a PTS, but it can easily be massaged into a PTS by considering the product of the train and the gate. Below follow the calculations needed to show that Equation (1) for this example is a T-assertion, thus we can apply the small model theorem and determine that the system is correct by checking that it is so for any number (of trains) smaller than 5, that is, 2 + the number of universally quantified variables in $\neg GI \vee \Psi$. To simplify the notation, we use $h_{ec}, h_{ac}, h_{lc}, h_{rc}$ (resp. $h_{et_i}, h_{at_i}, h_{lg}, h_{rg}$) to stand for the history clocks corresponding to *exit*, *approach*, *lower*, *raise* in the controller (resp. *exit*, *approach* in train i, and *lower*, *raise* in the gate).

$$
\begin{aligned}
CI(T(i)) =& \big((near_i = 1 \wedge x_i - h_{at_i} = 0 \wedge -x_i \geq -5 \wedge -x_i + h_{et_i} \geq 0 \wedge x_i \geq 0) \vee \\
& (far_i = 1 \wedge ((x_i - h_{et_i} = 0 \wedge x_i - h_{at_i} = 0) \vee \\
& \qquad (x_i - h_{at_i} = 0 \wedge -x_i + h_{et_i} \geq -5 \wedge x_i - h_{et_i} \geq 3))) \vee \\
& (in_i = 1 \wedge x_i - h_{at_i} = 0 \wedge -x_i \geq -5 \wedge -x_i + h_{et_i} \geq 0, x_i \geq 3)\big) \wedge \\
& far_i + near_i + in_i = 1 \\
II(\gamma) =& \sum_i far_i + c_1 + c_2 = n \wedge c_2 + c_3 + going\_up + is\_up = 1 \\
\equiv& \exists k \forall j \neq k. \underbrace{((c_0 = 1 \vee c_3 = 1) \wedge far_k = 1) \vee ((c_1 = 1 \vee c_2 = 1) \wedge far_k = 0 \wedge far_j = 1)}_{\circ(II_1)} \wedge \\
& \underbrace{c_2 + c_3 + going\_up + is\_up = 1}_{\circ(II_2)} \\
\mathcal{E}(\gamma) =& \exists p. \underbrace{(h_{ec} = h_{et_p} \wedge h_{rc} = h_{rg} \wedge h_{ac} = h_{at_p} \wedge h_{lc} = h_{lg})}_{\circ(\mathcal{E})} \\
\mathcal{E}^*(\gamma) =& h_{ec} = \min_s h_{ecet_s} \wedge h_{ac} = \min_s h_{acat_s} \wedge h_{rg} = h_{rcrg} \wedge \ldots \\
\mathcal{S}(\gamma) =& \forall q. \underbrace{(h_{ecet_q} - h_{ecet_{q-1}} \geq k_e \wedge h_{acat_q} - h_{acat_{q-1}} \geq k_a)}_{\circ(\mathcal{S})} \\
\Psi =& \forall r. \underbrace{in_r = 1 \rightarrow g_2 = 1}_{\circ(\Psi)}.
\end{aligned}
$$

That is, $\neg GI \vee \Psi$ can be rewritten as:

$$\forall k, p, r\ \exists i, j, q. \big(\neg\big(j \neq k \wedge CI(T(i)) \wedge CI(C) \wedge CI(G) \wedge \circ(II) \wedge \circ(\mathcal{E}) \wedge \mathcal{E}^* \wedge \circ(\mathcal{S})\big) \vee \circ(\Psi)\big)$$

which is a T-assertion since $CI(T(i))$, $CI(C)$, $CI(G)$, $\circ(II)$, $\circ(\mathcal{E})$, $\mathcal{E}^*$, $\circ(\mathcal{S})$, $\circ(\Psi)$ are expressions free of quantified variables. Using the "small model theorem", checking the correctness of the system for an arbitrary number of trains can be done by checking it for up to 3+2=5 trains. We used are prototype tool to successfully verified the instances $n = 1, \ldots, 5$, which prove the correctness of the system for any value of n.

4 Stochastic Abstraction

Our aim is to improve general applicability of Statistical Model Checking (SMC) [You05, SVA04, HLMP04, LLM+07] techniques. To this end, we propose to combine the use of abstraction and learning techniques to automatically construct faithful abstractions of system models towards making SMC more scalable. Nowadays, machine learning is an active field of research and learning algorithms are constantly developed and improved in order to address new challenges and new classes of problems (see [VEdlH12] for a recent survey on grammatical inference). In our context, learning is combined with abstraction as follows. Given a property of interest and a (usually large) sample of partial traces generated from a concrete system (model), we first use abstraction to restrict the amount of visible information on traces to the minimum required to evaluate the property and then, use learning to construct a compact, probabilistic model which conforms to the abstracted sample set. Under some additional restrictions discussed later, the resulting model is a sound abstraction of the concrete model with respect to the satisfaction of the property. Hence, it can be used to correctly predict/generate the entire abstract behavior of the model, in particular, as an input model for SMC.

The above approach has multiple benefits. First, the sample set of traces can be generated directly from an existing black-box implementation of the system, as opposed to a concrete detailed model. In many practical situations, such detailed system models simply do not exist and the cost for building them using reverse-engineering could be prohibitive. In such cases, learning provides an effective, automated way to obtain a model and to get some valuable insight on the system behavior. The use of projection is also mostly beneficial. In most of the cases, the complexity of the learning algorithms as well as the complexity of the resulting models are directly correlated to the the number of distinct observations (the alphabet) of traces. Moreover, under normal considerations, a large alphabet requires a large size for the sample set. Intuitively, the more complex the final model is, the more traces are needed to learn it correctly. Nevertheless, one should mention that a bit of care is needed to meaningfully combine abstraction and learning. That is, abstraction may change a deterministic model into a non-deterministic one, and henceforth has an impact on the learning algorithms needed for it.

4.1 Preliminaries

Let AP be a finite set of atomic propositions. We define the alphabet $\Sigma = 2^{AP}$ and denote the elements of Σ (subsets of AP) as symbols. The empty symbol is denoted by τ. As usual, we denote by Σ^ω (resp. Σ^*) the sets of infinite (resp. finite) words over Σ. For an infinite word $\sigma = \sigma_0\sigma_1...$ and $i \geq 0$, we define the ith suffix (resp. prefix) of σ as $\sigma[i..] = \sigma_i\sigma_{i+1}...$ (resp. as $\sigma[..i] = \sigma_0...\sigma_i$).

A labeled Markov chain (LMC) M is a tuple $\langle S, \iota, \pi, L\rangle$ where, S is a finite set of states, $\iota : S \to [0,1]$ is the initial states distribution such that $\sum_{s\in S} \iota(s) = 1$, $\pi : S \times S \to [0,1]$ is the probability transition function such that for each $s \in S$, $\sum_{s'\in S} \pi(s,s') = 1$, and $L : S \to \Sigma$ is a labeling function.

A *run* is a possible behavior (infinite execution) of the LMC. A *trace* is the sequence of labels associated to the states of the run. Let $M = \langle S, \iota, \pi, L\rangle$ be a LMC. A run of M is an infinite sequence of states $s_0s_1...s_ns_{n+1}...$ such that $\iota(s_0) > 0$ and $\pi(s_i, s_{i+1}) > 0$, for all $i \geq 0$. A trace σ associated to a run $s_0s_1...s_ns_{n+1}...$ is the infinite word $L(s_0)L(s_1)...L(s_n)L(s_{n+1})....$ A finite run (resp. trace) is any finite prefix of a run (resp. trace). We denote by $\textit{Runs}(M)$ the set of runs and by $\textit{Traces}(M)$ the set of traces of M. Moreover, we denote by Pr_M the underlying probability measure induced by M on the set of its traces. This measure is well-defined in the context of Markov chains [BK08]. Two LMCs M_1 and M_2 are said to be equivalent, and denoted $M_1 \approx M_2$ if they have identical probability measures on traces, that is, $Pr_{M_1} = Pr_{M_2}$. A labeled Markov chain is *deterministic* (DLMC) iff (i) $\exists s_0 \in S$ such that $\iota(s_0) = 1$, and (ii) $\forall s \in S$, $\forall \sigma \in \Sigma$ there exists at most one $s' \in S$ such that $\pi(s,s') > 0$ and $L(s') = \sigma$.

Linear-time temporal logic (LTL) [Pnu77] formulas φ built over a set of atomic propositions AP are defined by the following syntax:

$$\varphi := true \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid N\varphi \mid \varphi_1 U \varphi_2 \mid \varphi_1 U^i \varphi_2 \quad (p \in AP)$$

N, U and U^i are respectively the next, until and bounded until operators. Additional Boolean operators can be inferred from negation $\neg$ and conjunction $\wedge$. Moreover, temporal operators such as G (*always*) and F (*eventually*) are defined as $F\varphi \equiv true\ U\varphi$ and $G\varphi \equiv \neg F \neg\varphi$. The bounded fragment of LTL (denoted BLTL) restricts the use of the *until* operator U to its bounded variant U^i. LTL formula are interpreted on infinite traces $\sigma = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ as usual [Pnu77].

Given an LMC M and an LTL property φ, the probability for M to satisfy φ denoted by $Pr(M \vDash \varphi)$ is given by the measure $Pr_M\{\sigma \in \mathit{Traces}(M) \mid \sigma \vDash \varphi\}$.

4.2 Learning-based Abstraction

The verification problem in the stochastic setting amounts to compute $Pr(M \vDash \varphi)$ for an LMC M and an LTL property φ. As stated earlier, in order to enhance SMC performance, we would like to avoid the verification of φ on the original model M. Instead, we would like to perform it on a smaller model $M^\sharp$ which preserves the probability of φ, that is, $Pr(M \vDash \varphi) = Pr(M^\sharp \vDash \varphi)$. We propose hereafter a method to compute such an abstraction $M^\sharp$ by combining learning and projection on traces given the property φ. The idea is based on the simple observation that, when checking a model against a property, only a subset of the atomic propositions is really relevant. In fact, only the atomic propositions explicitly appearing in the property are useful while the others can be safely ignored.

The proposed approach is depicted in Figure 16. It consists of initially sampling a finite set of random finite traces T (with random lengths) from M. Second, a projection (detailed below) is applied on traces T in order to restrict the atomic propositions to the ones needed for the evaluation of the property φ. Third, the set of projected traces is used as an input to a learning algorithm. We experimented the proposed approach with the AAlergia algorithm [MCJ+11], however, any other algorithm can be used. The output of the learning step, denoted $M^\sharp$ on Figure 16, is finally used to evaluate φ.

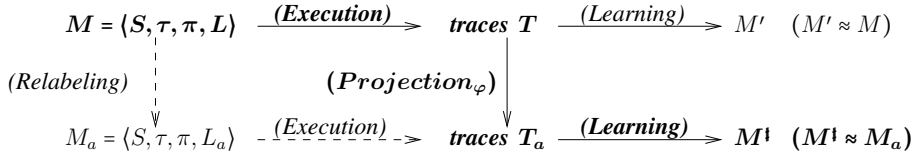


Figure 16: Learning abstract models: approach overview.

Main Steps

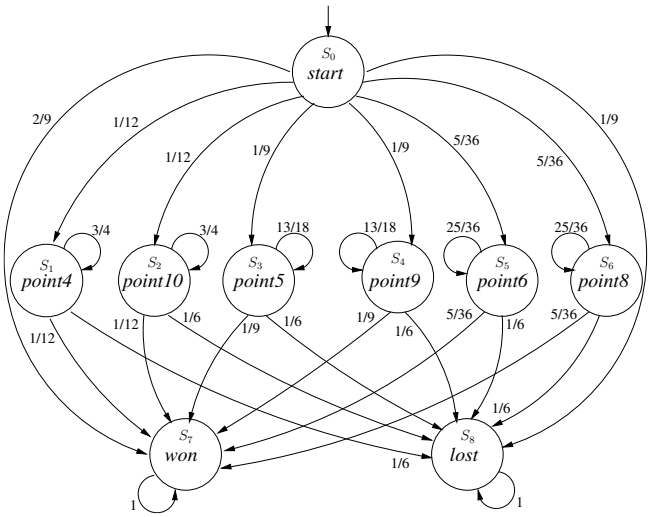
Projection. The projection is defined on traces so as to reduce the number of labels and henceforth, later on, the number of states in the learned model. It basically consists of ignoring the atomic propositions that are not relevant to the property under verification as follow. Let $V_\varphi \subseteq AP$ called the support of φ be the set of atomic propositions occurring explicitly in φ. The projection $\mathcal{P}_\varphi : \Sigma^* \to \Sigma^*$ is defined as $\mathcal{P}_\varphi(\sigma_0\sigma_1 \ldots \sigma_n) = \sigma'_0\sigma'_1 \ldots \sigma'_n$ where $\sigma'_i = \sigma_i \cap V_\varphi$ for all $i \in [0, n]$.

Learning. In this work, we rely on state merging algorithms which intuitively proceed by first constructing some large automata-based representation of the set of input traces and then progressively

compacting them, by merging states, into a smaller automaton, while preserving as much as possible trace occurrence frequencies/probabilities. Different algorithms in this family can learn either deterministic probabilistic finite automata (DPFA) models [CO94, dlHOV96, dlHO03] or general PFA models [Sto94, RST95, DEH06]. In this work, we use AAlergia [MCJ+11] which is a state merging algorithm that exclusively learn deterministic models. Given a sample of traces, the algorithm proceeds in three steps. It first builds an intermediate representation that represents all the traces in the input sample and their corresponding frequencies. Seconds, based on a compatibility criterion it iteratively merges states having the same labels and similar probability distributions until reaching a compact model. Finally, it transforms the obtained model into a DLMC. AAlergia is proven to converge to the correct model in the limit (with sufficiently big sample set of traces) [MCJ+11] if the input traces are generated, with random lengths, from an LMC model. It ensures that, a given LTL property will hold on the original and the learned model with the same probability (see [NRB+14] for a more detailed discussion on the equivalence cases and assumptions).

Example

We consider the Craps Gambling Game [BK08] to illustrate the proposed abstraction approach. In this game, A player starts by rolling two fair six-sided dice. The outcome of the two dice determines whether he wins or not. If the outcome is 7 or 11, the player wins. If the outcome is $2, 3$, or 12, the player looses. Otherwise, the dice are rolled again taking into account the previous outcome (called point). If the new outcome is 7, the player looses. If it is equal to point, he wins. For all other outcome, the dice are rolled again and the process continue until the player wins or looses. Figure 17 illustrates the DLMC model that describes the game behavior. A possible run of the DLMC below is $r = S_0S_5S_5S_7S_7\ldots$ The corresponding trace is t = *start point6 point6 won won* ... and $Pr(t) = 1 \times \frac{5}{36} \times \frac{25}{36} \times \frac{5}{36} \times 1 \times \ldots = 0.0277$.



$S = \{S_0, S_1, , ..., S_8\}$

$L(S_0) = start, \ldots, L(S_8) = lost$

$\iota(S_0) = 1$

$$\pi = \begin{array}{c} \\ S_0 \\ S_1 \\ \vdots \\ S_8 \end{array} \begin{array}{c} \begin{array}{cccc} S_0 & S_1 & \cdots & S_8 \end{array} \\ \begin{pmatrix} 0 & \frac{1}{12} & \cdots & \frac{1}{9} \\ 0 & \frac{3}{4} & \cdots & \frac{1}{12} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \end{array}$$

Figure 17: A DLMC model for the Craps Gambling Game.

Given the DLMC model in Figure 17, one could check for instance the following probabilistic (B)LTL properties:

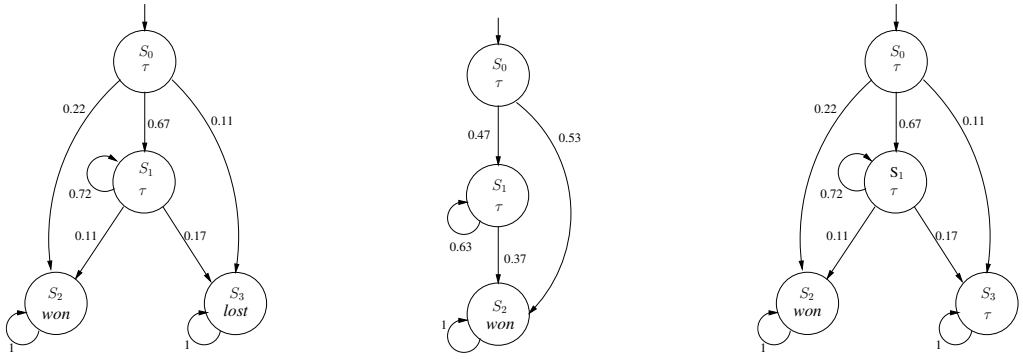
- The probability to eventually loose is $Pr(F\ lost) = 0.51$,
- The probability to eventually win is $Pr(F\ won) = 0.493$

- The probability to win in two steps is $Pr(true\ U^2\ won)$ = 0.3.

Given a set T of traces generated from the Craps Gambling Game model in Figure 17 and the properties φ_1 = $F\ won$ and φ_2 = $F\ (won \vee lost)$, we apply the projection definition to compute the corresponding sets of projected traces T_{a_1} and T_{a_2}.

- T = {*start won, start lost lost, start won won won won won won won won won, start point5, start point10 point10 point10 point10 point10, start point9 point9, ...*};
- T_{a_1} = {τ *won,* $\tau\ \tau\ \tau$, τ *won won won won won won won won won,* $\tau\ \tau$, $\tau\ \tau\ \tau\ \tau\ \tau\ \tau$, $\tau\ \tau\ \tau$, ...};
- T_{a_2} = {τ *won,* τ *lost lost,* τ *won won won won won won won won won,* $\tau\ \tau$, $\tau\ \tau\ \tau\ \tau\ \tau\ \tau$, $\tau\ \tau\ \tau$, ...}

We briefly illustrate the learning phase using AAlergia on the Craps example. Figure 18 shows three learned models of the Craps Gambling Game obtained using the set T of 5000 traces generated from the model in Figure 17. One can note, out of this figure, the important reduction of the obtained models sizes with respect to the original one. Figure 18a shows the model learned by AAlergia taking as input the set T_{a_2}, that is, with respect to property φ_2 = $F\ (won \vee lost)$. Figure 18b is obtained by applying AAlergia on the set T_{a_1}, that is, projected with respect to φ_1 = $F\ won$. Remark that this model is not equivalent but only an approximation of the original model in Figure 17. That is, in the latter there exists some non null probability to never reach the *won* state. Whereas, in the learned model the *won* state is reachable with probability 1. This approximation could however improve if a larger set of traces is used for learning as stated in the previous section. Finally, the third learned model shown in Figure 18c is equally obtained from T_{a_1} but when using an algorithm able to learn non-deterministic models such as the one proposed by Stolcke [Sto94].



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Figure 18: Learned Markov Chains for Craps Gambling Game using 5000 traces.

Once smaller models are learned, we can evaluate the considered properties on the these models. Table 1 provides results of verifying the property φ_1 = $F\ won$ on the Craps Gambling Game models. It shows that the model in Figure 18a exhibits similar probability to the original Craps model, whereas the one in Figure 18b shows different ones. The reason is that the projection introduced a non-determinism in the input sample. In addition, it seems that in this case there is no equivalent deterministic model that could be learned by AAlergia (see [NRB+14] for more details). A detailed study of the Herman's Self Stabilizing algorithm with more results is presented in [NRB+14].

More detailed results as well as a proof of correctness of the abstraction approach are presented in [NRB+14]. A detailed study of the Herman's Self Stabilizing algorithm is also presented in [NRB+14].

Models	$Pr(\varphi_1)$
Scenario 1*(Figure 18a)*	0.485
Scenario 2 *(Figure 18b)*	1
Original Model (Figure 17)	0.493

Table 1: Verifying φ_1 on the original and the learned Craps Gambling Game models using SMC.

The obtained results show that the proposed abstraction technique produces smaller models and enhances SMC analysis time (for different SMC algorithms), while preserving probability accuracy. Additional experiments using numerical probabilistic model checking algorithms are also presented for the same study. The results show that the proposed method is also applicable using this kind of analysis.

5 Conclusion

This year, we mainly worked for improving the applicability of the results on correctness of Service Component Ensembles (SCEs).

We developed a prototype of translator from a subset of SCEL called SCELight to BIP. This is an important step since it permits to apply a significant part of the verification results of the project which focused on the BIP language, to specifications written in the SCEL language which was developed specifically for the project and extensively used in this context. The prototype was validated on simple examples. As future work we need to consider larger examples, as well as to complete the prototype which has currently several restrictions on the form the input SCELight programs.

We also improved the compositional verification method for timed systems which was proposed in the project as a means for verifying SCEs, by including specific techniques for dealing with "untimed" components, i.e. components that are free of timing constraints. We have also preliminary results on the extension of the method to parameterized systems, allowing in principle to apply formal verification to systems composed of unknown (i.e. unbounded) number of components. Such an approach is very promising for the project but it needs to be evaluated on real case studies before drawing conclusions. Moreover it is currently too restrictive regarding topologies considered for the connections between components, and thus has to be completed in this respect.

Finally we presented an automatic method for the construction of faithful abstractions of system models, which is an original approach for learning stochastic models from back-box implementations of a system. It is based on a notion of projection which is currently quite simple, and could be improved to obtain coarser abstractions, e.g. by taking into account the LTL operators semantics. We shall also apply it to real-life examples.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 1994.

[ADR+11] Parosh Aziz Abdulla, Giorgio Delzanno, Othmane Rezine, Arnaud Sangnier, and Riccardo Traverso. On the verification of timed ad hoc networks. In *FORMATS*, 2011.

[ARB+14] Lacramioara Astefanoaei, Souha Ben Rayana, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Compositional invariant generation for timed systems. In *TACAS*, 2014.

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *ATVA*, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, 2006.

[CO94] Rafael C. Carrasco and José Oncina. Learning Stochastic Regular Grammars by Means of a State Merging Method. In *ICGI*, pages 139–152, 1994.

[CY92] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1992.

[DEH06] Francois Denis, Yann Esposito, and Amaury Habrard. Learning rational stochastic languages. COLT'06, 2006.

[dlHO03] Colin de la Higuera and José Oncina. Identification with Probability One of Stochastic Deterministic Linear Languages. In *ALT*, pages 247–258, 2003.

[dlHOV96] Colin de la Higuera, José Oncina, and Enrique Vidal. Identification of DFA: data-dependent vs data-independent algorithms. In *ICGI*, pages 313–325, 1996.

[DLLL+14] Rocco De Nicola, Alberto Lluch-Lafuente, Michele Loreti, Andrea Morichetta, Rosario Pugliese, Valerio Senni, and Francesco Tiezzi. Programming and verifying component ensembles. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *FPS@ETAPS*, volume 8415 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2014.

[GNRZ08] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards smt model checking of array-based systems. In *IJCAR*, 2008.

[HLMP04] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate Probabilistic Model Checking. In *VMCAI*, pages 73–84, 2004.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 1994.

[JM12] Taylor T. Johnson and Sayan Mitra. A small model theorem for rectangular hybrid automata networks. In *FMOODS/FORTE*, 2012.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 1987.

[LBiBB13] Axel Legay, Saddek Bensalem, Benoît Boyer, and Marius Bozga. Incremental generation of linear invariants for component-based systems. In *ACSD*, 2013.

[LLM+07] S. Laplante, R. Lassaigne, F. Magniez, S. Peyronnet, and M. de Rougemont. Probabilistic abstraction for model checking: An approach based on property testing. *ACM TCS*, 8(4), 2007.

[MCJ+11] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning Probabilistic Automata for Model Checking. In *QEST*, pages 111–120, 2011.

[NRB+14] Ayoub Nouri, Balaji Raman, Marius Bozga, Axel Legay, and Saddek Bensalem. Faster statistical model checking by means of abstraction and learning. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 340–355, 2014.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[RST95] Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. COLT, pages 31–40, 1995.

[RTD] RTD-finder. `http://www-verimag.imag.fr/RTD-Finder.html?lang=en`.

[Sto94] Andreas Stolcke. *Bayesian Learning of Probabilistic Language Models*. PhD thesis, Berkeley, CA, USA, 1994. UMI Order No. GAX95-29515.

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114, pages 202–215. Springer, 2004.

[VEdlH12] Sicco Verwer, Rémi Eyraud, and Colin de la Higuera. Results of the pautomac probabilistic automaton learning competition. In *ICGI*, pages 243–248, 2012.

[XTe] Xtext. `https://eclipse.org/Xtext/`.

[You05] H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.