Project Number 318772

D4.5 Compositional verification techniques and tools for distributed MILS - part 2

Version 1.0
1 September 2015
Final

Public Distribution

Fondazione Bruno Kessler, RWTH Aachen University, Université Joseph Fourier, University of York

Project Partners: Fondazione Bruno Kessler, fortiss, Frequentis, Inria, LynuxWorks, The Open Group, RWTH Aachen University, TTTech, Université Joseph Fourier, University of York

Every effort has been made to ensure that all statements and information contained herein are accurate, however the D-MILS Project Partners accept no liability for any error or omission in the same.

© 2015 Copyright in this document remains vested in the D-MILS Project Partners.
# Project Partner Contact Information

<table>
<thead>
<tr>
<th>Fondazione Bruno Kessler</th>
<th>fortiss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alessandro Cimatti</td>
<td>Harald Ruess</td>
</tr>
<tr>
<td>Via Sommarive 18</td>
<td>Guerickestrasse 25</td>
</tr>
<tr>
<td>38123 Trento, Italy</td>
<td>80805 Munich, Germany</td>
</tr>
<tr>
<td>Tel: +39 0461 314320</td>
<td>Tel: +49 89 36035 22 0</td>
</tr>
<tr>
<td>Fax: +39 0461 314591</td>
<td>Fax: +49 89 36035 22 50</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:cimatti@fbk.eu">cimatti@fbk.eu</a></td>
<td>E-mail: <a href="mailto:ruess@fortiss.org">ruess@fortiss.org</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequentis</th>
<th>LynuxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wolfgang Kampichler</td>
<td>Yuri Bakalov</td>
</tr>
<tr>
<td>Innovationstrasse 1</td>
<td>Rue Pierre Curie 38</td>
</tr>
<tr>
<td>1100 Vienna, Austria</td>
<td>78210 Saint-Cyr-l’Ecole, France</td>
</tr>
<tr>
<td>Tel: +43 664 60 850 2775</td>
<td>Tel: +33 1 30 85 06 00</td>
</tr>
<tr>
<td>Fax: +43 1 811 50 77 2775</td>
<td>Fax: +33 1 30 85 06 06</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:wolfgang.kampichler@frequentis.com">wolfgang.kampichler@frequentis.com</a></td>
<td>E-mail: <a href="mailto:ybakalov@lnxw.com">ybakalov@lnxw.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RWTH Aachen University</th>
<th>The Open Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joost-Pieter Katoen</td>
<td>Scott Hansen</td>
</tr>
<tr>
<td>Ahornstrasse 55</td>
<td>Avenue du Parc de Woluwe 56</td>
</tr>
<tr>
<td>D-52074 Aachen, Germany</td>
<td>1160 Brussels, Belgium</td>
</tr>
<tr>
<td>Tel: +49 241 8021200</td>
<td>Tel: +32 2 675 1136</td>
</tr>
<tr>
<td>Fax: +49 241 8022217</td>
<td>Fax: +32 2 894 5845</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:katoen@cs.rwth-aachen.de">katoen@cs.rwth-aachen.de</a></td>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TTTech</th>
<th>Université Joseph Fourier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilfried Steiner</td>
<td>Saddek Bensalem</td>
</tr>
<tr>
<td>Schonbrunner Strasse 7</td>
<td>Avenue de Vignate 2</td>
</tr>
<tr>
<td>1040 Vienna, Austria</td>
<td>38610 Gieres, France</td>
</tr>
<tr>
<td>Tel: +43 1 5853434 983</td>
<td>Tel: +33 4 56 52 03 71</td>
</tr>
<tr>
<td>Fax: +43 1 585 65 38 5090</td>
<td>Fax: +33 4 56 03 44</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:wilfried.steiner@tttech.com">wilfried.steiner@tttech.com</a></td>
<td>E-mail: <a href="mailto:saddek.bensalem@imag.fr">saddek.bensalem@imag.fr</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of York</th>
<th>Inria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tim Kelly</td>
<td>Axel Legay</td>
</tr>
<tr>
<td>Deramore Lane</td>
<td>Inria - Campus de Beaulieu</td>
</tr>
<tr>
<td>York YO10 5GH, United Kingdom</td>
<td>35042 Rennes, France</td>
</tr>
<tr>
<td>Tel: +44 1904 325477</td>
<td>Tel: +33 2 99 84 73 15</td>
</tr>
<tr>
<td>Fax: +44 7976 889 545</td>
<td>Fax: +33 2 99 84 71 71</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:tim.kelly@cs.york.ac.uk">tim.kelly@cs.york.ac.uk</a></td>
<td>E-mail: <a href="mailto:axel.legay@inria.fr">axel.legay@inria.fr</a></td>
</tr>
</tbody>
</table>
## Contents

1. **Summary**  
1

2. **Introduction**  
2

3. **Technical description**  
3

   3.1 Overview of the verification framework  
   3.2 Intransitive Non-Interference of MILS-AADL  
      3.2.1 I/O Components and System Architectures  
      3.2.2 Information Flow Policies  
      3.2.3 Information Flow Policy of I/O Components  
      3.2.4 Refinement  
   3.3 Transitive Non-Interference for Component-Based Systems with secureBIP  
      3.3.1 Information Flow Security  
      3.3.2 Checking Non-interference  
      3.3.3 Synthesis of Secure Annotations  
      3.3.4 Use Case: Home Gateway  
   3.4 Analyzing Cryptographically-Masked Information Flow  
      3.4.1 Possibilistic Non-Interference  
      3.4.2 The Type Checking Approach  
      3.4.3 The Influence Graph Approach  
   3.5 Contract-Based Verification of Functional, Real-Time, Security, and Safety Properties with OCRA and nuXmv  
      3.5.1 Properties as Temporal Formulas  
      3.5.2 Contract-Based Compositional Approach  
      3.5.3 Layered Approach for Contract-Based Reasoning  
      3.5.4 Verification of Infinite-State Systems  
      3.5.5 Verification of Real-Time and Hybrid Systems  
      3.5.6 Safety  
      3.5.7 Starlight example  
   3.6 Compositional Verification for Deadlock Freedom of Timed Systems with RT-Dfinder  
      3.6.1 Timed Invariant Generation  
      3.6.2 Implementation and Experiments  
      3.6.3 Extension for Parametric Timed Systems  
   3.7 Monolithic Verification with COMPASS  

## 4 Tool-set

4.1 COMPASS Front-End ................................................. 43
4.2 Verification with OCRA ............................................. 43
   4.2.1 Integration .................................................. 43
   4.2.2 Annotations .................................................. 44
   4.2.3 GUI and Guidelines ......................................... 46
4.3 Verification with secureBIP .......................................... 46
   4.3.1 Integration .................................................. 46
   4.3.2 Guidelines .................................................. 47
4.4 Verification with RT-DFinder ......................................... 47
   4.4.1 Integration .................................................. 47
   4.4.2 Guidelines .................................................. 48
4.5 Verification with COMPASS .......................................... 48
   4.5.1 Integration .................................................. 48
   4.5.2 GUI and Guidelines ......................................... 48

## 5 Assurance Considerations ........................................... 49

### Glossary .......................................................... 50

### Acronyms .......................................................... 51

### References ....................................................... 52
### Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Document started</td>
<td>26 January 2015</td>
</tr>
<tr>
<td>0.2</td>
<td>First version completed</td>
<td>4 August 2015</td>
</tr>
<tr>
<td>0.3</td>
<td>Integration of new contributions and feedback</td>
<td>31 August 2015</td>
</tr>
<tr>
<td>1.0</td>
<td>First released version</td>
<td>1 September 2015</td>
</tr>
</tbody>
</table>
1 Summary

This deliverable describes the final formal framework and the corresponding formal verification methodology unifying the different verification activities for distributed MILS as result of a first iteration of the methodology and tools on the Starlight example. It will also describe the final results on the use of compositional reasoning for each of the functional, safety and security verification activities. Finally, it will report on the result of integration of the different verification subsystems in a unique toolset.
2 Introduction

A fundamental part of the D-MILS methodology is the verification of the architecture. This defines and proves a set of properties of the architecture that encompass functional, real-time, safety and security properties. The assurance of such properties relies on, first, the correctness of the system architecture and in the refinement of such properties in local policies for the application components; second, on the correctness of the configuration compiler and the platform, which must correctly implement the specified architecture; and, third, on the guarantees of the application components, which must correctly implement the specified local policies. This deliverable focuses on the first point, namely on the techniques used to probe properties on the system architecture.

The rest of the report is organized as follows. Section 3 gives first an overview of the framework, and then describes in details the different verification techniques, including those already described in D4.4. Section 4 describes the tools that are used in the verification framework and how they are interconnected and gives some guidelines on how to use the front end, how to annotate the MILS-AADL model with directives for the verification, and how to interpret the results. Finally, Section 5 makes some consideration on the assurance provided by the verification framework.

The document builds on the results presented in D4.4 and integrates D4.4 and new developed techniques in a more structured and uniformed presentation. In particular, the sections that contain D4.4 parts are Sections 3.3, 3.5.2, 3.5.3, 3.5.4, 3.5.5, 3.6. All the remaining parts are new in D4.5.
3  Technical description

3.1  Overview of the verification framework

This report focuses on the integration of the MILS architectural approach with compositional verification techniques to prove properties on the system architecture. Both MILS and compositional verification approaches focus on architecture, and do so in a complementary way. MILS regards information flow policy as an abstraction of architecture, and seeks to maximize the correspondence between architectural structure and the desired information flow policy of a system, which may rely on the behavior of key components to enforce local policies that further restrict the maximal information flow permitted by the architecture. Compositional verification approaches employ formalization and a method to prove that the architecture decomposition represented in the set of properties of the components is a proper refinement of the system requirements. Formal verification techniques are used to check that the derivation of the local policies from the system requirements is correct.

An architecture is only as valuable as the integrity of its components and connections. Recognizing the importance of integrity, MILS provides an implementation platform that can be configured to the “shape” of the architecture by initializing it with specific configuration data compiled to embody the global information flow policy.

The two methods are complementary and their combination yields strong results. The compositional verification methods prove that the composition of components that satisfy their properties will meet the system requirements, provided that their integrity is protected. The MILS platform guarantees the integrity of components and their configured connections, preventing interference that could cause a verified component to fail to satisfy its properties.

In Figure 1 we show the approach applied to an abstract example. The system $A$ is decomposed into subsystems $B$ and $C$, and $B$ in turn is decomposed into $D$ and $E$. Each component is enriched with a property (represented here by green scrolls). If the property refinement is correct, we have associated with the architecture a formal proof that the system is correct provided that the leaf components ($D$, $E$, and $C$) satisfy their properties. Namely, if $D$ and $E$ satisfy their properties ($D \models P_D$, $E \models P_E$) and the property refinement of $B$ is correct ($\gamma_B(P_D,P_E) \leq P_B$), then the composition of $D$ and $E$ satisfies the property of $B$ ($\gamma_B(D,E) \models P_B$). Moreover, if $C$ satisfies its property ($C \models P_C$) and the property refinement of $A$ is correct ($\gamma_A(P_B,P_C) \preceq P_A$), then the composition of $B$ and $C$ satisfies the property of $A$ ($\gamma_A(B,C) \models P_A$).

In MILS terms, the architecture defines three subjects ($D$, $E$ and $C$) and prescribes that the only allowed communications must be the ones between $D$ and $E$ and between $E$ and $C$. This is translated into a configuration for the D-MILS platform (taking into account other deployment constraints in terms of available resources), which in this example encompasses two MILS nodes.

In the next section, we will formalize the intransitive information flow policy that is guaranteed by construction by the MILS approach through the MILS platform configuration compiler (MPCC) and the MILS platform. In the following sections, we will describe the compositional techniques used to prove different properties on the architecture.
Figure 1: The architecture is used for 1) formal reasoning to prove that the system requirements are assured by the local policies, 2) configuration of the platform to ensure the global information flow policy and the integrity of the architecture.

3.2 Intransitive Non-Interference of MILS-AADL

3.2.1 I/O Components and System Architectures

In this section, we formalize the notion of information flow policy that is inherently defined by a system architecture specified in MILS-AADL and guaranteed by construction by the MILS approach. In particular, given an architecture \( arc \), we can define an information flow policy \( I(\text{arc}) \) such that every system implementation of \( arc \) is secure for \( I(\text{arc}) \).

In order to simplify the definition and to reuse standard notions and results about information flow policies, we restrict the semantics of components to concurrent systems interacting through message passing (event data). Also, we consider only the specification of the system architecture in MILS-AADL without taking into account the implementations that can be specified in MILS-AADL. The component implementations defined in this section are deterministic state machines used at the semantics level to define the information flow policy of the system architecture.

Each component has a set of ports that define the interaction of the component with its environment. We consider only event data ports, which represent events carrying some data (message passing). These include also simple event ports (without data). We divide the event data ports of a component \( u \) into input event data ports \( IE_u \) and output event data ports \( OE_u \). Every event data port \( e \) is associated with a tuple of data variables. Each data variable has a type. We denote with \( W_u \) the collection of all possible interpretations of data variables associated to the events of \( u \). Given a component \( u \) let \( E_u \) be the union \( IE_u \cup OE_u \).

A decomposition of a component \( u \) defines:
D4.5 Compositional verification techniques and tools for distributed MILS

- a set $\text{Sub}_u$ of subcomponents;
- a mapping $\gamma_u : (\text{OE}_u \cup \bigcup_{v \in \text{Sub}_u} \text{IE}_v) \to (\text{IE}_u \cup \bigcup_{v \in \text{Sub}_u} \text{OE}_v)$ such that
  - the tuples of the associated data of the connected event is compatible (same size, same types);
  - if $e \in \text{OE}_u$ then $\gamma(e) \notin \text{IE}_u$ (no short-circuit from input to output, i.e. no direct pass-through);
  - if $e \in \text{IE}_v$ for some $v \in \text{Sub}_u$, then $\gamma(e) \notin \text{OE}_u$ (no short-circuit from output to input, i.e. no direct feedback);
  - for all $v \in \text{Sub}_u$, for all $e, e' \in \text{IE}_v$, if $e \neq e'$, then $\gamma(e) \neq \gamma(e')$; similarly, for all $e, e' \in \text{OE}_u$, if $e \neq e'$, then $\gamma(e) \neq \gamma(e')$ (for each component, only one event port is possible).

Intuitively, the mapping $\gamma$ represents the connections among event data ports and define a set of equalities so that every output port of a composite component is equal to the output port of a subcomponent and every input port of a subcomponent is equal to the output port of another subcomponent or the input port of the parent component. Note that fan-out of events is possible since the input port of a composite component and the output ports of the subcomponents can be involved in multiple equalities.

The last three constraints are necessary in case of asynchronous systems to preserve the mutual exclusion of input/output event ports: in fact, a trace of the component concatenates transitions where each transition has only one “active” event port; if we allowed direct through-pass or direct feedback, we would obtain a contradiction where an input event port is fired together with an output event port.

We call action a pair given by an event port and a value for each associated data variable (i.e., an element of $W_u$). An input [output] action is an action corresponding to an input [resp. output] event. Let $\text{IA}_u$ and $\text{OA}_u$ resp. the set of input and output actions of the component $u$. $\gamma$ is extended to input/output actions in the straightforward way.

We use input/output state machines as implementations of a component. More specifically, the implementation of a component $u$ is a tuple $m_u = \langle S, \epsilon, \tau_u, \text{istep}, \text{ostep}, \text{tstep} \rangle$ where:

- $S$ is the set of states;
- $\epsilon \in S$ is the initial state;
- $\tau_u$ is the set of internal events;
- $\text{istep} : S \times \text{IE}_u \times W_u \to S$ is the transition function for input events;
- $\text{ostep} : S \times \text{OE}_u \to W_u \times S$ is the transition function for output events;
- $\text{tstep} : S \times \tau_u \to S$ is the transition function for internal events.

The actions $A_u$ of the state machine is the union $(\text{IE}_u \times W_u) \cup \tau_u$. A trace of $u$ is a sequence $\alpha = \alpha_1, \ldots, \alpha_k \in A^*$ of actions in $A_u$. A computation of $m$ over a trace $\alpha \in A^*$ is a sequence of states $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_k \in S^\epsilon$ such that $i = \sigma_0$ and for every $j$, $1 \leq j \leq k$, if $\alpha_j = \langle e, v \rangle$ with $e \in \text{IE}_u$ then $\sigma_j = \text{istep}(\sigma_{j-1}, e, v)$, else if $\alpha_j = \langle e, v \rangle$ with $e \in \text{OE}_u$ then $\langle v, \sigma_j \rangle = \text{ostep}(\sigma_{j-1}, e)$, else $\alpha_j \in \tau_u$ and $\sigma_j = \text{tstep}(\sigma_{j-1}, \alpha_j)$.

If the component $u$ is decomposed into $n$ subcomponents $\text{Sub}_u = \{v_1, \ldots, v_n\}$ with the connection $\gamma_u$, then given an implementation $m_i$ for every subcomponent $v_i \in \text{Sub}_u$, we define the product implementation of $u$, denoted with $\gamma_u(m_1, \ldots, m_n)$ as the tuple $\langle S, \epsilon, \text{step} \rangle$ where:
D4.5 Compositional verification techniques and tools for distributed MILS

\[
S := S_1 \times \ldots \times S_n; \text{ given a state } s \text{ of the product we denote with } s_j \text{ the projection of } s_j \text{ on } S_j;
\]

\[
i := \langle i_1, \ldots, i_n \rangle;
\]

\[
\tau_a := \tau_1 \cup \ldots \cup \tau_n \cup \bigcup_j \{e_j \in OE_j \mid \forall e \in OE_a \gamma(e) \neq e_j\};
\]

\[
s' = \text{istep}(s, e, v) \iff \text{for all } j \text{ either there exists } e_j \in IE_j \text{ such that } \gamma(e_j) = e \text{ and } s'_j = \text{istep}_j(s_j, e_j, v) \text{ or } s'_j = s_j;
\]

\[
\langle d, s' \rangle = \text{ostep}(s, e) \iff \text{there exists } j \text{ such that } \langle d, s'_j \rangle = \text{ostep}(s_j, \gamma(e)) \text{ and for all } k \neq j \text{ either there exists } e_k \in IE_k \text{ such that } \gamma(e_k) = e \text{ and } s'_k = \text{istep}_k(s_k, e_k, d) \text{ or } s'_k = s_k;
\]

\[
s' = \text{tstep}(s, e) \iff \text{there exists } j \text{ such that } s'_j = \text{tstep}(s_j, e) \text{ and for all } k \neq j \text{ or } e \in OE_j \text{ and there exists } d \in OD_j \text{ such that } \langle d, s'_j \rangle = \text{ostep}(s_j, e) \text{ and for all } k \neq j \text{ either there exists } e_k \in IE_k \text{ such that } \gamma(e_k) = e \text{ and } s'_k = \text{istep}_k(s_k, e_k, d) \text{ or } s'_k = s_k.
\]

If the component is hierarchically decomposed, its implementation is given by the product of the implementations of the leaf components.

Given two components \( u \) and \( v \), and two events \( e_u \in E_u \) and \( e_v \in E_v \), we say that there is a connection from \( e_u \) to \( e_v \) (denoted with \( e_v = \gamma^*(e_u) \)) if there exists a sequence \( v_1, \ldots, v_n \) of components with ports \( e_{v_1}, \ldots, e_{v_n} \) (with \( e_{v_1} \in E_{v_1} \)) such that \( v_1 = u, v_n = v, e_u = e_{v_0}, e_v = e_{v_n} \) and for all \( i, 1 \leq i < n, e_{v_{i+1}} = \gamma(e_{v_i}) \). Thus, if \( e_{v_i} \) is an input port, then \( v \) must be a subcomponent of \( u \), while if \( e_{v_i} \) is an output port then either \( u \) is a subcomponent of \( v \) or \( u \) and \( v \) are subcomponents of the same component.

Given two components \( u \) and \( v \) in the hierarchy, we say that there is a connection from \( u \) to \( v \) iff there exist two events \( e_u \in E_u \) and \( e_v \in E_v \) such that there is a connection from \( e_u \) to \( e_v \).

3.2.2 Information Flow Policies

We recall the definition of Mealy state machines that are used for interpreting information flow policies. A Mealy state machine is a tuple \( M = \langle S, i, A, O, \text{step}, \text{output} \rangle \) where:

- \( S \) is the set of states
- \( i \in S \) is the initial state
- \( A \) is the set of actions
- \( O \) is the set of outputs
- \( \text{step} : S \times A \rightarrow S \) is the transition function
- \( \text{output} : S \times A \rightarrow O \) is the output function.

Note that the sets of states, actions, and outputs may be infinite (differently from the standard definition).

An information flow policy for the state machine \( M \) is a tuple \( \langle D, \rightarrow, \text{dom} \rangle \) where:

- \( D \) is the set of domains
- \( \rightarrow \subseteq D \times D \) is the information flow relation
- \( \text{dom} : A \rightarrow D \) associates a domain with each action.

Let us consider the definition of secure system given by van der Meyden [42]. The function \( ta_u \), which maps a sequence \( \alpha \) of actions to the maximal information that domain \( u \) is permitted to have
after \( \alpha \) (represented by a tree over actions in \( \alpha \)) is defined as follows: \( t_{\alpha}(\epsilon) := \epsilon \); if \( \text{dom}(a) \not\rightarrow v \), then \( t_{\alpha}(\alpha a) := t_{\alpha}(\alpha) \); if \( \text{dom}(a) \sim v \), then \( t_{\alpha}(\alpha a) := \langle t_{\alpha}(\alpha), t_{\alpha}(\text{dom}(a))(a), a \rangle \).

A state machine is secure with respect to an information flow policy, the state machine is secure for that information flow policy.

In [42], van der Meyden proved that if there exists a weak unwinding for a state machine with respect to a given system architecture is secure with respect to an information flow policy derived from the architecture.

Suppose we have for each domain \( u \) a relation \( \sim_u \) over the states of a state machine. Such family of relations is called a weak unwinding iff the following three conditions hold for all states \( s, t \), for all domains \( u \):

- **OUTPUT CONSISTENCY**: if \( s \sim_u t \), then \( \text{output}(s, \text{dom}(u)) = \text{output}(t, \text{dom}(u)) \);
- **WEAK STEP CONSISTENCY**: if \( s \sim_u t \) and \( s \sim_{\text{dom}(a)} t \), then \( \text{step}(s, a) \sim_u \text{step}(t, a) \);
- **LEFT RESPECT**: if \( \text{dom}(a) \not\rightarrow u \), then \( s \sim_u \text{step}(s, a) \).

In [42], van der Meyden proved that if there exists a weak unwinding for a state machine with respect to an information flow policy, the state machine is secure for that information flow policy.

### 3.2.3 Information Flow Policy of I/O Components

Let us consider a system architecture with root component \( w \) and leaf components \( \text{leaves}(w) \). Let \( m_w = \langle S, i, \tau_w, i\text{step}, o\text{step}, t\text{step} \rangle \) be the composite implementation of \( w \). We define the corresponding Mealy state machine \( \langle S, i, A, O, \text{step}, \text{output} \rangle \) as follows:

- \( A \) is the union of the input actions of \( w \) and the output and internal events of all leaf components (the value of all other ports are derived through the connections);
- \( O := W_w \) is the set of data;
- the functions \( \text{step} \) and \( \text{output} \) are defined as follows:
  - if \( a \) is an input action of \( w \), then \( \text{output}(s, a) = \bot \) and \( s' = \text{step}(s, a) \) iff \( a = \langle e, d \rangle \), \( s' = i\text{step}(s, e, d) \);
  - if \( e_u \) is an output event of a leaf component \( u \), then \( s' = \text{step}(s, e_u) \) and \( d = \text{output}(s, e_u) \) iff either
    - there exists \( e \in OE_w \) such that \( \gamma^{*}(e) = e_u \) and \( \langle d, s' \rangle = o\text{step}(s, e, d) \) or
    - \( d = \bot \) and \( s' = t\text{step}(s, e_u) \).

We enrich a system architecture with root \( w \) with the following notions that define the information flow policy:

- \( D := \text{leaves}(w) \cup \{ env_w \} \) where \( env_w \) represents the environment of \( w \);
- \( u \sim v \) iff there is a connection from \( u \) to \( v \) or \( u = v \);
- \( \text{dom}(a) := u \) iff either \( a \) is an output event of a leaf component \( u \) or \( u = env_w \) and \( a \) is an input action of \( w \).
For every domain $u$, for all states $s, t \in S$, we define the relation $s \approx_u t$ as $s_u = t_u$, i.e., $s \approx_u t$ if the local component of $u$ is the same.

We prove that $s \approx_u t$ is a weak unwinding.

**OUTPUT CONSISTENCY:** If $a$ is an input action of $w$, then trivially $\text{output}(s, a) = \text{output}(t, a) = \bot$; consider an output event of the leaf component $u$; then if $s \approx_{\text{dom}(e)} t$ then $s_{\text{dom}(e)} = t_{\text{dom}(e)}$ and so $\text{output}(s, e) = \text{output}(t, e)$.

**WEAK STEP CONSISTENCY:** If $\text{dom}(e) = u$, then if $s \approx_u t$ then $s_u = t_u$ and so $\text{step}(s, e) = \text{step}(t, e)$. Consider instead the case $\text{dom}(e) = v \neq u$. If $s \approx_v t$, then $\text{output}(s, e) = \text{output}(t, e) = d$ for some $d$; thus if $s \approx u t$, then $\text{istep}(s_u, e', d) = \text{istep}(t_u, e', d)$ and thus $\text{step}(s, e) \approx_u \text{step}(t, e)$.

**LEFT RESPECT:** If $\text{dom}(a) \not\rightarrow u$ then $u \neq \text{dom}(a)$ and there is no connection from $\text{dom}(a)$ to $u$.

Therefore, by the definition of $\text{step}$, $s \approx_u \text{step}(s, a)$.

Thus, by the result of van der Meyden, the system implementation is secure for the above information flow policy.

### 3.2.4 Refinement

In the previous section, we showed how given an architecture $arc$, we can define an information flow policy $I(arc)$ such that every system implementation of $arc$ is secure for $I(arc)$. If we consider a refinement $arc'$ of $arc$ obtained by decomposing some leaf components into subcomponents, we can consider the new information policy architecture $I(arc')$. This is indeed a refinement of $I(arc)$ so that we obtain that any system implementation of $arc'$ is also secure for $I(arc')$.

Let $w$ be the root component of $arc$ and $arc'$ and let $leaves$ be the leaf component of $arc$ and $leaves'$ be the leaf components of $arc'$. Then, we can build the mapping $r$ from $D'$ to $D$ such that $r(\text{env}_w) = \text{env}_w$ and $r(u') = u$ iff $u' \in \text{Sub}_u$. Clearly, $r$ is onto $D$, and if $u' \sim u' \sim v$ then $r(u') \sim r(v)$.

Note that MILS-AADL uses “subject” components to define the “meant” information flow policy of the system architecture. The formal notion of information flow policy is therefore $I(arc)$ where $arc$ is the system architecture considering the “subjects” as leaves (and ignoring the subcomponents of the subjects).

### 3.3 Transitive Non-Interference for Component-Based Systems with secure-BIP

In this section we summarize our recent work on information flow security for components based systems [9]. We have extended the BIP framework [8] with security features, leading to the secure-BIP framework. secureBIP allows for building secure, complex and hierarchically structured systems from atomic components behavior and interactions, annotated with security information. For secure-BIP we study two types of non-interference: event non-interference, tracking information flow about occurrences of interactions, and data non-interference, tracking information flow about data changes. Considering both notions of non-interference provide a finer-grain information flow security model.
D4.5 Compositional verification techniques and tools for distributed MILS

compared to solutions addressing a unique model. Moreover, we identified a set of sufficient syntactic conditions allowing to automate verification of non-interference. This work has been published in [9] and an extended version can be found as the technical report [39].

3.3.1 Information Flow Security

In order to track information flows, we adopt the classification technique and we define a classification policy where we annotate the information by assigning security levels to different parts of BIP model (data variables, ports and interactions). The policy describes how information can flow from one classification with respect to the other. As an example, we can classify public information as a Low (L) security level and secret (confidential) information as High (H) security level. Intuitively High security level is more restrictive than Low security level and we denote it by $L \subseteq H$.

Security domain and annotations

In general, security levels are elements of a *security domain*, defined as a lattice of the form $\langle S, \subseteq, \cup, \cap \rangle$ where:

- $S$ is a finite set of security levels,
- $\subseteq$ is a partial order "can flow to" on $S$ that indicates that information can flow from one security level to an equal or a more restrictive one,
- $\cup$ is a "join" operator for any two levels in $S$ and that represents the upper bound of them,
- $\cap$ is a "meet" operator for any two levels in $S$ and that represents the lower bound of them.

In the example above, the set of security levels is $S = \{L, H\}$ and the "can flow to" partial order relation is defined as $L \subseteq L, L \subseteq H, H \subseteq H$.

A formal introduction of BIP has been provided in deliverable D3.3 [27]. To avoid duplication, we refer to this deliverable for the BIP syntax and semantics as well as for BIP notations used hereafter.

Let $C = \gamma(B_1, \ldots, B_n)$ be a BIP composite component. Let $X$ (resp. $P$) be the set of all variables (resp. ports) defined in all atomic components $(B_i)_{i=1,n}$. Let $\langle S, \subseteq, \cup, \cap \rangle$ be a security domain, fixed.

A *security assignment* for component $C$ is a mapping $\sigma : X \cup P \cup \gamma \to S$ that associates security levels to variables, ports and interactions such that, moreover, the security levels of ports matches the security levels of interactions, that is, for all $a \in \gamma$ and for all $p \in P$ it holds $\sigma(p) = \sigma(a)$. In atomic components, the security levels considered for ports and variables allow to track intra-component information flows and control the intermediate computation steps. Moreover, inter-components communication, that is, interactions with data exchange, are tracked by the security levels assigned to interactions.

Figure 2 is a composite component that contains three atomic components interacting trough connectors (lines) between distinct ports. Variables, ports and interactions are tagged with High (H) and Low (L) security levels (graphically represented with dashed squares).
Event and Data Non-Interference

We will now formally introduce the notions of non-interference for our component model. We start by providing few additional notations and definitions. Let $\sigma$ be a security assignment for $C$, fixed. For a security level $s \in S$, we define $\gamma \downarrow^s$ the restriction of $\gamma$ to interactions with security level at most $s$ that is formally, $\gamma \downarrow^s = \{a \in \gamma \ | \ \sigma(a) \subseteq s\}$.

For a security level $s \in S$, we define $w\upharpoonright_s$ the projection of a trace $w \in \gamma^s$ to interactions with security level lower or equal to $s$. Formally, the projection is recursively defined on traces as $\epsilon \upharpoonright_s = \epsilon$, $(aw)\upharpoonright_s = a(w\upharpoonright_s)$ if $\sigma(a) \subseteq s$ and $(aw)\upharpoonright_s = w\upharpoonright_s$ if $\sigma(a) \not\subseteq s$. The projection operator $\upharpoonright_s$ is naturally lifted to sets of traces $W$ by taking $W\upharpoonright_s = \{w\upharpoonright_s \ | \ w \in W\}$.

For a security level $s \in S$, we define the equivalence $\approx_s$ on states of $C$. Two states $q_1, q_2$ are equivalent, denoted by $q_1 \approx_s q_2$ iff (1) they coincide on variables having security levels at most $s$ and (2) they coincide on control states having outgoing transitions labeled with ports with security level at most $s$. We are now ready to define the two notions of non-interference.

The security assignment $\sigma$ ensures event non-interference of $\gamma(B_1, \ldots, B_n)$ at security level $s$ iff, $$\forall q_0 \in Q_C^0 : \ \text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)\upharpoonright_s = \text{TRACES}((\gamma \downarrow^s)(B_1, \ldots, B_n), q_0)$$

Event non-interference ensures isolation/security at interaction level. The definition excludes the possibility to gain any relevant information about the occurrences of interactions (events) with strictly greater (or incomparable) levels than $s$, from the exclusive observation of occurrences of interactions with levels lower or equal to $s$. That is, an external observer is not able to distinguish between the case where such higher interactions are not observable on execution traces and the case these interactions have been actually statically removed from the composition. This definition is very close to Rushby’s [38] definition for transitive non-interference. But, let us remark that event non-interference is not concerned about the protection of data.

The security assignment $\sigma$ ensures data non-interference of $C = \gamma(B_1, \ldots, B_n)$ at security level $s$ iff

$$\forall q_1, q_2 \in Q_C^0 : q_1 \approx_s q_2 \Rightarrow$$

$$\forall w_1 \in \text{TRACES}(C, q_1), w_2 \in \text{TRACES}(C, q_2) : w_1\upharpoonright_s = w_2\upharpoonright_s \Rightarrow$$

$$\forall q'_1, q'_2 \in Q_C : q_1 \xrightarrow{a_1} q'_1 \land q_2 \xrightarrow{a_2} q'_2 \Rightarrow q'_1 \approx_s q'_2$$

Figure 2: BIP Composite component annotated
Data non-interference provides isolation/security at data level. The definition ensures that, all states reached from initially indistinguishable states at security level $s$, by execution of arbitrary but identical traces whenever projected at level $s$, are also indistinguishable at level $s$. That means that observation of all variables and interactions with level $s$ or lower excludes any gain of relevant information about variables at higher (or incomparable) level than $s$. Compared to event non-interference, data non-interference is a stronger property that considers the system’s global states (local states and valuation of variables) and focus on their equivalence along identical execution traces (at some security level).

In the secureBIP framework, a security assignment $\sigma$ is said secure for a component $\gamma(B_1, \ldots, B_n)$ iff it ensures both event and data non-interference, at all security levels $s \in S$.

### 3.3.2 Checking Non-interference

We provide hereafter sufficient syntactic conditions that aim to simplify the verification of non-interference and reduce it to local constrains check on both transitions (inter-component verification) and interactions (intra-component verification). Especially, they give an easy way to automate the verification.

Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let $\sigma$ be a security assignment. We say that $C$ satisfies the security conditions for security assignment $\sigma$ iff:

(i) the security assignment of ports, in every atomic component $B_i$ is locally consistent:

- for every pair of causal transitions:

$$\forall \tau_1, \tau_2 \in T_i : \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \tau_2 = \ell_2 \xrightarrow{p_2} \ell_3 \Rightarrow (\ell_1 \neq \ell_2 \Rightarrow \sigma(p_1) \subseteq \sigma(p_2))$$

- for every pair of conflicting transitions:

$$\forall \tau_1, \tau_2 \in T_i : \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \tau_2 = \ell_1 \xrightarrow{p_2} \ell_3 \Rightarrow (\ell_1 \neq \ell_2 \Rightarrow \sigma(p_1) \subseteq \sigma(p_2))$$

(ii) all assignments $x := e$ occurring in transitions within atomic components and interactions are sequentially consistent, in the classical sense:

$$\forall y \in use(e) : \sigma(y) \subseteq \sigma(x)$$

(iii) variables are consistently used and assigned in transitions and interactions, that is,

$$\forall \tau \in \bigcup_{i=1}^{n} T_i \forall x, y \in X : x \in def(f_\tau), y \in use(g_\tau) \Rightarrow \sigma(y) \subseteq \sigma(p_\tau) \subseteq \sigma(x)$$

$$\forall a \in \gamma \forall x, y \in X : x \in def(F_a), y \in use(G_a) \Rightarrow \sigma(y) \subseteq \sigma(a) \subseteq \sigma(x)$$

(iv) all atomic components $B_i$ are port deterministic:

$$\forall \tau_1, \tau_2 \in T_i : \tau_1 = \ell_1 \xrightarrow{p} \ell_2, \tau_2 = \ell_1 \xrightarrow{p} \ell_3 \Rightarrow (g_{\tau_1} \land g_{\tau_2}) \text{ is unsatisfiable}$$
D4.5 Compositional verification techniques and tools for distributed MILS

In the above, we used the classical notations $use(e)$, $def(f)$ to denote respectively the set of variables used in the expression $e$ and defined by the assignment $f$. Moreover, recall that for a component transition $\tau$ we denote respectively by $p_\tau$ its associated port, by $g_\tau$ its guard expression and by $f_\tau$ its update action, that is, an assignment of local variables. Similarly, for an interaction $a$, we denote respectively by $G_a$ its associated guard expression and by $F_a$ its associated data transfer, an assignment to variables of interacting components.

The first family of conditions (i) is similar to Accorsi’s conditions [2] for excluding causal and conflicting places for Petri net transitions having different security levels. Similar conditions have been considered in [29, 30] and lead to more specific definitions of non-interferences and bisimulations on annotated Petri nets. The second condition (ii) represents the classical condition needed to avoid information leakage in sequential assignments. The third condition (iii) tackles covert channels issues. Indeed, (iii) enforces the security levels of the data flows which have to be consistent with security levels of the ports or interactions (e.g., no low level data has to be updated on a high level port or interaction). Such that, observations of public data would not reveal any secret information. Finally, condition (iv) enforces deterministic behavior on atomic components.

We proved in [9] that the above security conditions are sufficient to ensure both event and data non-interference. Formally, whenever the security conditions hold, the security assignment $\sigma$ is secure for the composite component $C$.

As an illustration, consider the composite component in Figure 2. It can be relatively easily checked that the security conditions hold, henceforth, the composite component is secure.

### 3.3.3 Synthesis of Secure Annotations

In addition to non-interference checking, we develop a synthesis tool [10] assisting a designer to ensure end-to-end information flow security. The system designer describes his components and defines partial security annotations for a subset of model elements (some of the data, ports and the interactions). The synthesis tool attempts to extend the partial annotations towards a complete/total annotation while fulfilling all security constraints due to implicit and explicit data/ports dependencies in the system. The calculated configuration is moreover optimal that is only data that need to be protected is configured and its security level is minimal. It is indeed, very important to reduce the security processing overhead like cryptography encryption and decryption, signature calculation, certificate verification, etc. In case a total configuration is generated by the tool, the system information flow is then guaranteed non-interferent. Otherwise, the system is interferent and system designer has to re-consider the input initial configuration. As a security advisor, automated configuration synthesis allows designers, developers and administrators to focus on functional constraints and be confident that their secret data is protected and people privacy respected.

### 3.3.4 Use Case: Home Gateway

As an illustrative example, we consider a Home Gateway system remotely managed via the Internet network and inspired from a real application [18]. To maintain energy consumption, "Temperature" and "Presence" Web services (WS) that encapsulate the sensors and the actuators embedded in the
building, collect information and then send it to an "Analyzer" WS to compare them and to take decisions to reset temperature value according to the presence or absence of a person in the building. This system is given in MILS-AADL language in deliverable D2.2.

This example presents a typical non-interference issue. Considering that presence information is confidential. Thus, by a simple observation of temperature-related action message from "Analyzer" WS to "Temperature" WS, one can easily deduce the presence information.

The modeling of the system using secureBIP involves two main distinct steps: first, functional requirements modeling reflecting the system behavior, and second, security annotations enforcing the desired security policy. As depicted in Figure 3, the model of the system contains three components denoted: Temperature, Presence and Analyzer. The components Temperature and Presence send to the Analyzer component respectively variables \( T \) and \( P \). The Analyzer calculates the temperature deviation \( \Delta T \) from some \( \Theta_1 \) and \( \Theta_2 \) variables, temperatures threshold depending on \( P \). Then, \( \Delta T \) is sent to the Temperature component to reset the temperature value. For instance, when there is no one in the building (\( P \) is set to false) and the value of \( T \) is over 15, a negative \( \Delta T \) value is sent to Temperature component to update the \( T \) value.

The system can be proven non-interferent if it satisfies the syntactic security conditions. Indeed, following the security assignment these conditions hold for the system model. In order to avoid non-interference, the \( \Delta T \) has to be confidential, that is assigned with high security level since it depends on the confidential \( P \) variable. Besides, since \( T \) is reset depending on the received \( \Delta T \) then, it has also to be confidential and assigned with high security level.
3.4 Analyzing Cryptographically-Masked Information Flow

We explore an analysis technique based on possibilistic non-interference [35], which extends traditional non-interference to deal with cryptographic primitives. This technique was published at the MILS Workshop 2015 [43] and is based on security type checking. It can handle encryption, but cannot handle non-determinism. Another technique using influence graphs is presented as future work, which can additionally handle non-determinism.

3.4.1 Possibilistic Non-Interference

Standard non-interference states that low confidentiality outputs may not change when high confidentiality inputs are changed. This is violated in systems which output encrypted confidential information over untrusted channels. However, such systems should be considered safe because encryption masks the information from any attackers. Thus, standard non-interference rejects legitimate uses of encryption.

A variant of non-interference aims to repair this defect. In possibilistic non-interference [35], we look at the set of possible values after encryption instead of the actual value. Varying the message does not change the possible outcomes of encryption: any ciphertext is still a possible public output. Care must be taken, however, in defining what ciphertexts are indistinguishable to an attacker. Askarov et al. [4] assume an indistinguishability equivalence \( = \) on ciphertexts such that for any \( v, k, v', k' \)

\[
\forall u \in \text{encrypt}(v, k). \exists u' \in \text{encrypt}(v', k'). u \neq u', \text{ and } \\
\exists u \in \text{encrypt}(v, k), u' \in \text{encrypt}(v', k'). u \neq u'.
\]

The first condition allows for safe usages in the sense of possibilistic encryption. It excludes the existence of a ciphertext \( u \in \text{encrypt}(v, k) \) that is distinguishable from any \( u' \in \text{encrypt}(v', k') \). The second condition prevents implicit flow. It states that not all ciphertexts are indistinguishable. Both conditions are realistic for encryption schemes with the computational security properties Indistinguishability under Chosen Plaintext Attack (IND-CPA) and Integrity of Plaintexts (INT-PTXT) (by a previous result from Laud [34]). Assuming such an indistinguishability relation exists for the encryption scheme used, the type checking approach can detect unwanted data leaks while allowing intended and correct usage of encryption.

3.4.2 The Type Checking Approach

To describe what data is confidential and what is public, we enhance the MILS-AADL type system as described in Document D2.1 [26] by attaching a security type to the relevant syntactic constructs. We then type check the annotated model to verify that it is non-interfering.

Some restrictions to the models apply. These are assumed to hold but not checked by the type checking procedure. Here we only briefly enumerate the associated requirements; for details we refer to [43]. First, the models must be deterministic. Second, the next low confidentiality transition on a path is determined by the last low confidentiality transitions on a path. There can be high confidentiality
transitions in between, but these should converge to the same low confidentiality transition. Third, performing high confidentiality transitions indefinitely is not possible \[28\].

**Syntax**

The user annotates each mode, port and variable with a confidentiality level which describes what data is confidential and what is public. W.l.o.g., we assume that there are two levels, $H$ (high) and $L$ (low). Intuitively, the high confidentiality level means that information is to be kept secret and not made visible to the outside world. No such restriction is placed on data with a low confidentiality level. The data marked $L$ may also be considered $H$ (but certainly not the other way round!), denoted $H \sqsubseteq L$. We assume the relation $\sqsubseteq$ is a preorder. This relation will later be used to define subtyping.

For port and variable values, this confidentiality level can be lowered by encrypting. These annotations yield a function $T$ mapping data ports and variables to their declared security type, and mapping modes and event ports to their declared confidentiality level.

Confidentiality level $\sigma ::= H \mid L$

Basic type $t ::= \text{int} \mid \text{bool} \mid \text{enc} \tau$

Security type $\tau ::= t \sigma \mid \text{key} \sigma \mid (\tau, \ldots, \tau)$

For keys, we fix the confidentiality level as follows: private keys are always $H$, public keys are $L$. A generalized confidentiality level for keys is possible, but does not add any interesting cases.

We use the function $\text{lvl}$ to give the confidentiality level of a type. Formally, $\text{lvl}$ is defined as follows:

\[
\text{lvl}(t \sigma) := \sigma \quad \text{lvl}(\text{key} \sigma) := \sigma \quad \text{lvl}((\tau_1, \ldots, \tau_n)) := \text{lvl}(\tau_1) \sqcup \cdots \sqcup \text{lvl}(\tau_n)
\]

where $\sqcup$ denotes the least upper bound operation of preorder $\sqsubseteq$. We lift the definition of $\text{lvl}$ to expressions in the obvious way. Moreover, for a mode $m$, we let $\text{lvl}(m)$ be its declared confidentiality level, and similarly for an event port $p$.

**Type Checking**

For a system annotated with confidentiality levels, we verify non-interference by type checking.

**Expressions** We first consider type checking for expressions. The type rules for expressions are as follows:

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Type Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>$T \vdash n : \text{int} \ L$</td>
</tr>
<tr>
<td>Boolean</td>
<td>$T \vdash b : \text{bool} \ L$</td>
</tr>
<tr>
<td>Variable</td>
<td>$T(x) = \tau$</td>
</tr>
<tr>
<td>Tuple</td>
<td>$T \vdash e_1 : \tau_1 \ldots T \vdash e_n : \tau_n$</td>
</tr>
<tr>
<td>Projection</td>
<td>$1 \leq i \leq n \quad T \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)$</td>
</tr>
<tr>
<td>$T \vdash e_1 : \tau_1$</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>$T \vdash e_2 : \tau_2 \quad T \vdash e_1 \oplus e_2 : t (\sigma_1 \sqcup \sigma_2)$</td>
</tr>
<tr>
<td>Encryption</td>
<td>$T \vdash e_1 : \tau \quad T \vdash e_2 : \text{key} \ L$</td>
</tr>
<tr>
<td>$T \vdash \text{encrypt}(e_1, e_2) : \text{enc} \tau \ L$</td>
<td></td>
</tr>
<tr>
<td>Decryption</td>
<td>$T \vdash e_1 : \text{enc} \tau \sigma \quad T \vdash e_2 : \text{key} \ H$</td>
</tr>
<tr>
<td>$T \vdash \text{decrypt}(e_1, e_2) : \tau^\sigma$</td>
<td></td>
</tr>
</tbody>
</table>
Compositional verification techniques and tools for distributed MILS

For obvious reasons, the type rules are considered in the context of $T$. Integers and booleans are of a low confidentiality by default. They can be considered secret through subtyping, as discussed below. Tuples and projection are standard. For operators, we consider total binary operators where the basic types are standard and the confidentiality level is the highest of the operands’ confidentiality levels.

Encryption lowers the confidentiality level to $L$, i.e., ciphertexts can be made public. Decryption expressions are of the type of the contents tainted with the confidentiality level of the ciphertext, where tainting a type $\tau$ with a security level $\sigma$, denoted $\tau^{\sigma}$, is defined as:

$$((t^{\sigma})^{\sigma'} := t(\sigma \sqcup \sigma') \quad (\text{key } L)^{\sigma} := \text{key } L \quad (\text{key } H)^{\sigma} := \text{key } H$$

Note that tainting keys with higher levels of confidentiality is not allowed. This is due to fixing the confidentiality levels of private and public keys: public keys cannot become private through tainting.

We denote that a type $\tau$ is a subtype of type $\tau'$ as $\tau <: \tau'$. Subtyping allows us to succinctly describe that high confidentiality data may not be assigned to low confidentiality variables, i.e., restrict explicit data flows. The subtyping rules are fairly standard:

<table>
<thead>
<tr>
<th>Types</th>
<th>Subtyping Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>$\sigma \sqsubseteq \sigma'$</td>
</tr>
<tr>
<td>$\text{int } \sigma &lt;: \text{int } \sigma'$</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>$\sigma \sqsubseteq \sigma'$</td>
</tr>
<tr>
<td>$\text{bool } \sigma &lt;: \text{bool } \sigma'$</td>
<td></td>
</tr>
<tr>
<td>Keys</td>
<td>$\text{key } \sigma &lt;: \text{key } \sigma$</td>
</tr>
</tbody>
</table>

Connections We need to restrict physical connections so that confidential ports are not physically connected to a public port. This restricts explicit data flow. The corresponding type rule infers a confidentiality level for the connection only if it is properly connected. The corresponding type rule is:

$$T \vdash s_1.p_1 : \tau_1 \quad T \vdash s_2.p_2 : \tau_2 \quad \tau_1 <: \tau_2 \quad T \vdash (s_1.p_1, s_2.p_2) : \tau_1$$

(data connection)

For event ports, the type rule is similar:

$$T \vdash s_1.p_1 : \sigma_1 \quad T \vdash s_2.p_2 : \sigma_2 \quad \sigma_1 \sqsubseteq \sigma_2 \quad T \vdash (s_1.p_1, s_2.p_2) : \sigma_1$$

(event connection)

Modes and transitions We add type rules for modes and transitions to reject systems with unwanted implicit data flows, where confidential data is leaked through the control flow. The user declares modes as high or low confidentiality. The confidentiality levels of transitions are inferred automatically. Even though we pose no direct requirements on this declaration, the type rule for transitions indirectly restricts what modes may be declared to be of high or low confidentiality. The type rule is as follows:

$$T \vdash p : \tau_p \quad T \vdash g : \tau_g \quad \tau_x <: \tau_e \quad \text{lvl}(\tau_x) \sqsubseteq \sigma \quad \text{lvl}(\tau_p) \sqsubseteq \sigma \quad \text{lvl}(\tau_g) \sqsubseteq \sigma \\
T \vdash x : \tau_x \quad \text{lvl}(m) \sqsubseteq \text{lvl}(\tau_p) \quad \sigma \sqsubseteq \text{lvl}(\tau_g) \\
T \vdash e : \tau_e \quad \text{lvl}(m) \sqsubseteq \text{lvl}(\tau_x) \quad \sigma \sqsubseteq \text{lvl}(\tau_g) \\
T \vdash m - \{ p \ \text{when} \ g \ \text{then} \ x := e \} \rightarrow m' : \sigma$$

Confidentiality: Public Distribution
The type rule for transition can be summarized as follows:

1. The effect must be properly typed, i.e., the type of $x$ is a subtype of that of $e$: $\tau_x \prec \tau_e$. This enforces traditional type safety as well as prevents unwanted explicit data flow.

2. A transition from a high mode to another mode may not be triggered by a low confidentiality event, i.e., the source mode has a higher confidential level than the event: $\text{lvl}(m) \not\subseteq \text{lvl}(\tau_p)$ Intuitively, observable behavior only occurs in the transitions from a low confidentiality mode to any other mode.

3. A transition from a high mode to another mode may not assign any value to a low confidentiality variable, i.e., the source mode has a higher confidentiality level than the effect’s variable: $\text{lvl}(m) \not\subseteq \text{lvl}(\tau_x)$.

4. A high transition may not assign any value to a low confidentiality variable, i.e., the variable $x$ has a higher confidential level than the transition: $\text{lvl}(\tau_x) \not\subseteq \sigma$. This could make the high transition observable.

5. A transition is high if it is triggered by a high confidentiality event, i.e., the transition has a higher confidential level than the event: $\sigma \not\subseteq \text{lvl}(\tau_p)$.

6. A transition is high if its guard expression has a high confidentiality, i.e., the transition has a higher confidential level than the guard: $\sigma \not\subseteq \text{lvl}(\tau_g)$.

Intuitively, all modes are of low confidentiality except when they deal with confidential information. If one were to make too many modes of a low confidentiality, the restrictions defined at the start of this section will reject any branching on confidential data. On the other hand, if one were to make too many modes of a high confidentiality, transitions which update public data will not be typeable. When the system is typeable, the confidential data cannot spill over into public outputs, which is exactly what we aim for.

A proof sketch of this soundness property of the type system as well as application examples can be found in [43].

### 3.4.3 The Influence Graph Approach

The influence graph approach goes beyond the type checking approach in that it can handle non-deterministic models. As with the type checking approach, it can handle cryptographic operators. Additionally, the influence graph can be easily extended and can be used in a slicing algorithm where we infer security levels for some variables and ports. Here we present some preliminary definitions and outline future work.

We consider the syntactic description of the model (in MILS-AADL) and not the semantics of expressions. This lowers the complexity of the analysis at the expense of precision. However, some semantic information is useful and we assume this is supplied by some external verification procedure or by the designer: Each set of outgoing transitions with the same variables occurring on their guards must be exhaustive, meaning that for any valuation of these variables, there is at least one
guard in the set that is enabled. These sets are called *exhaustive sets*. The guards are allowed to be *overlapping*, meaning that for any valuation of these variables, strictly more than one guard in the set is enabled. A guard in a set is called overlapping if it can be enabled simultaneously with some other guard in the set.

**Construction of Influence Graphs**

To calculate whether a component is non-interfering, we construct an influence graph. In its most basic form, the nodes of the graph are variables and an edge $x \rightarrow y$ states that if some predicate on $y$ is known, some other predicate on $x$ can (potentially) be deduced. We abstract from what information can be deduced, only that some values for $y$ may be inconsistent with some observations for $x$. In the construction of the influence graph, we overapproximate this influence relation.

In this section, we formally introduce the node types of the influence graph with accompanying rules for the influence relation. Each node type refines the basic notion of influence graph with more control over what information may be deduced, allowing for a finer approximation.

**Expression nodes** The expression node, denoted $\langle e, m \rangle$ is the value of expression $e$ when the system is in mode $m$.

The subexpressions can influence an expression with a binary operator, index expression or tuple.

$$\langle e, m \rangle \leadsto \langle e', e, m \rangle, \langle e \oplus e', m \rangle, \langle e[e'], m \rangle, \langle \ldots, e, \ldots, m \rangle$$  \hspace{1cm} (1)

In an assignment, the value of the expression before the assignment, influences the value of the variable after the assignment.

$$\langle e, m \rangle \leadsto \langle x, m' \rangle \text{ if } m \xrightarrow{\text{then } x := e} m'$$  \hspace{1cm} (2)

In transitions where a variable is not assigned to, the value in the target mode is influenced by the value in the source mode.

$$\langle x, m \rangle \leadsto \langle x, m' \rangle \text{ if } m \xrightarrow{l} m' \text{ where } l \text{ does not contain then } x := \ldots$$  \hspace{1cm} (3)

**Visited mode nodes** The visited mode node, denoted $m$, indicates that execution has been in mode $m$ in the past. This for example means that, if certain regions of the model can only be reached by taking specific transitions, the guards on these transitions must have been true and the assignments on these transitions have occurred.

If we can potentially know something about a variable’s value in a mode, then we can potentially derive that execution has been in that mode:

$$\langle x, m \rangle \leadsto m$$  \hspace{1cm} (4)

The value of the variable after the assignment can reveal that the mode before the assignment was reached.

$$m \leadsto \langle x, m' \rangle \text{ for each } m \xrightarrow{\text{then } x := e} m'$$  \hspace{1cm} (5)

Reachability of a mode after a guarded transition can reveal that the guard was true.

$$\langle g, m \rangle \leadsto m' \text{ for each } m \xrightarrow{\text{when } g} m'$$  \hspace{1cm} (6)
Choice nodes  The choice node, denoted $\sigma_m$, is the nondeterministic choice made at mode $m$. There is a nondeterministic choice if there are more than one exhaustive sets or if an exhaustive set is overlapping. We make no additional assumptions on this choice. Specifically, it is possible this choice depends on confidential data and cause unintended information flow. The variables influencing the nondeterministic choice are possibly all variables in the component’s scope.

$x \leadsto \sigma_m$ for each $x$ in the current component’s scope \hspace{1cm} (7)

If there is more than one exhaustive set, the nondeterministic choice influences all target modes.

$\sigma_m \leadsto m'$ for each $m \xrightarrow{l} m'$ if there are multiple exhaustive sets \hspace{1cm} (8)

If an exhaustive set is overlapping, the nondeterministic choice influences the target modes of all overlapping transitions.

$\sigma_m \leadsto m'$ for each overlapping $m \xrightarrow{l} m'$ \hspace{1cm} (9)

Cryptographic operators  For the cryptographic operators, the subexpressions do not directly influence the ciphertext: this is intended use of encryption. Only when the encryption and decryption keys are both influenced by the same key pair, can they possibly match.

$\langle v,m \rangle \leadsto \langle \text{decrypt}(\text{msg},k'),m' \rangle$ if $\langle \text{encrypt}(v,k),m \rangle \leadsto \langle \text{msg},m' \rangle$,

and some key pair $kp \leadsto \langle k,m \rangle, \langle k',m' \rangle$ \hspace{1cm} (10)

Verification procedure

The verification procedure relies on the interpretation of the influence graph. The absence of a path from high to low nodes means that there is no possible direct or indirect influence from one to the other. If no high confidentiality node has a path to a low confidentiality node, this implies non-interference.

Theorem 1  A component is non-interfering if the influence graph contains no path from any high confidentiality variable or port to any low confidentiality variable or port.

If the property is violated, the path serves as a witness to the possible information leak.

We can also use the influence graph in a slicing algorithm, where the high confidentiality nodes are given and we infer that every reachable node from a high confidentiality node, must itself be of a high confidentiality. This is a simple reachability analysis.

Finally, the verification procedure can be made more precise by introducing new types of nodes in the influence graph.
3.5 Contract-Based Verification of Functional, Real-Time, Security, and Safety Properties with OCRA and nuXmv

3.5.1 Properties as Temporal Formulas

Temporal Logic  Since the seminal work of Pnueli in 1977 [37], the use of temporal logic for reasoning about the properties of computerized systems has been steadily increased. The idea is that a temporal formula formalizes a functional requirement of the system, i.e., an action or sequence of actions that the system must be able to perform in response to some inputs. Temporal logic is ideal for this purpose because a model of a formula can be seen as a computation of the system, i.e., a sequence of states where each transition from state to state represents a step in the computation. Thus, if the behavior of a system $M$ is represented by a set of computations $L_M$, the property $\phi$ is satisfied by the system (denoted with $M \models \phi$) iff each computation $\sigma \in L$ is a model of the formula $\phi$ (denoted with $\sigma \models \phi$). If timing aspects such as the time between two events or the absolute time of an execution is part of the alphabet describing the states or the transitions, then also some real-time requirements can be captured by temporal logic. Similarly, if security aspects such as security levels can be expressed, then also some security requirements can be formalized using the logic. The same applies to some safety requirements: if the requirement specify the capability of a component to react (for example, by triggering an alarm or some recovery actions) to input conditions that are considered not safe, then the requirement can be expressed in temporal logic.

In D-MILS, the language used to specify properties is the one provided by the OCRA tool [20]. It consists of a textual human-readable version of a Real-time First-Order Linear-time Temporal Logic. The logic extends Linear-time Temporal Logic (LTL) [37] with predicates over data, event and event data ports, with metric operator to constrain the time among events, and with uninterpreted functions, i.e. functional symbols that do not have a specific interpretation but are used to abstract procedures and the related results (such as CRC checksum or encryption), or to label data with user-defined tags (such as “is_high” or “low-level”, etc.).

More specifically, the formulas of the logic are defined in the following table, where on left the concrete syntax of OCRA is used, while on the right the formulas are written in a mathematical notation.
Let a computation $\sigma$ consist of 1) a sequence $s$ alternating assignments to the data variables and events 2) a divergent weakly-monotonic sequence $T$ of time points, and 3) an interpretation $I$ of the uninterpreted functions. The computation satisfies the formulas according to the following rules:
3.5.2 Contract-Based Compositional Approach

We used the property-based composition approach described in D4.1. The compositional verification techniques are typically based on proving that each component \( m_i \) satisfies some local assertion \( \phi_i \) and on checking some proof obligations on the assertions. This amounts essentially to prove the validity of a finite number of implications involving the assertions \( \phi_1, \ldots, \phi_n \) and \( \phi \). The compositional verification rule has therefore the following general form:

\[
\text{for all } i, 1 \leq i \leq n, m_i \models \phi_i \\
\text{compprop}_\gamma(\phi_1, \ldots, \phi_n) \preceq \phi \\
\text{compsys}_\gamma(m_1, \ldots, m_n) \models \phi
\]

where \( \text{compsys}_\gamma \) is the composition operator of the components \( m_1, \ldots, m_n \) given the connections \( \gamma \); \( \text{compprop}_\gamma \) is the related composition operator for the properties; \( \preceq \) is the refinement operator for the properties. In synchronous systems, for example, \( \gamma \) can be seen as a renaming, \( \text{compsys}_\gamma \) is the synchronous composition, and \( \text{compprop}_\gamma \) is the conjunction (after renaming). In asynchronous systems as in the case of MILS-AADL, the operators are more complex and can be reduce to the synchronous case by introducing additional variables (such as stuttering of not active components) and conditions (such as frame conditions).

The \( \preceq \) relation is typically given by the implication. When the properties are structured into contracts, the relation is more complex in order to take into account the assumption of components on the environment. Let \( C = \langle A,G \rangle \) be a contract of \( S \). Let \( I \) and \( E \) be respectively an implementation and an environment of \( S \). We say that \( I \) is an implementation satisfying \( C \) iff \( I \models A \rightarrow G \). We say that
Figure 4: Layers in the reasoning engine underlying OCRA

$E$ is an environment satisfying $C$ iff $E \models A$. We denote with $\mathcal{I}(C)$ and with $\mathcal{E}(C')$, respectively, the implementations and the environments satisfying the contract $C$. We say that a contract $C'$ refines a contract $C$ ($C' \leq C$) iff $\mathcal{I}(C') \subseteq \mathcal{I}(C)$ and $\mathcal{E}(C) \subseteq \mathcal{E}(C')$. This notion has been extended in [23] to consider the refinement along a structural decomposition taking into account the contracts of the sub-components of a component. Still in [23], it is proved that the contract refinement can be verified by the generation of a set of proof obligations, which are temporal formulas that are valid if and only if the refinement is correct.

In the following sections, we describe how we verify the above conditions when properties are expressed in the logic defined in Section 3.5.1.

### 3.5.3 Layered Approach for Contract-Based Reasoning

The logic defined in Section 3.5.1 is very expressive and required the development of effective techniques to reason about it. To this purpose the engine undertakes a layered approach to prove the contract refinement as depicted in Figure 4. The refinement is first translated by OCRA into a set of entailment problems in temporal logic. nuXmv [17] translates this into a liveness model-checking problem with a classic automata-theoretic approach [45]. The resulting problem requires proving that a certain liveness condition can be visited only finitely many times along an (infinite) execution. This problem is in turn reduced to proving an invariant on the reachable states with the K-liveness techniques described in [25]. This has been extended to infinite-state systems and to take into account real-time aspects in [22]. Finally, the invariant is proved with an efficient combination of induction-based reasoning, explicit-state search, and predicate abstraction, extending the IC3 algorithm [15] to the infinite-state case, as described in [21].

**Transition Systems representing the MILS-AADL programs** We use Transition Systems as the semantics of MILS-AADL programs to reason about their temporal properties. Basically, the semantics of MILS-AADL defined in D3.3 can be mapped to Transition Systems so that each MILS-AADL
program $P$ can be mapped to a Transition System $M_P$ that has the same computations of $P$, and therefore $P \models \phi$ iff $M_P \models \phi$ for any property $\phi$.

Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We denote formulas with $\varphi, \psi, I, T, P$, variables with $x, y, v$, and sets of variables with $X, Y, V$. Unless otherwise specified, we work on quantifier-free formulas, and we refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. A literal is an atom or its negation. A clause is a disjunction of literals, whereas a cube is a conjunction of literals. If $s$ is a cube $l_1 \land \ldots \land l_n$, with $\neg s$ we denote the clause $\neg l_1 \lor \ldots \lor \neg l_n$, and vice versa. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, and in disjunctive normal form (DNF) if it is a disjunction of cubes. With a little abuse of notation, we might sometimes denote formulas in CNF $C_1 \land \ldots \land C_n$ as sets of clauses $\{C_1, \ldots, C_n\}$, and vice versa. If $X_1, \ldots, X_n$ are sets of variables and $\varphi$ is a formula, we might write $\varphi(X_1, \ldots, X_n)$ to indicate that all the variables occurring in $\varphi$ are elements of $\bigcup_i X_i$. For each variable $x$, we assume that there exists a corresponding variable $x'$ (the primed version of $x$). If $X$ is a set of variables, $X'$ is the set obtained by replacing each element $x$ with its primed version ($X' = \{x' \mid x \in X\}$).

Given a formula $\varphi$, $\varphi'$ is the formula obtained by adding a prime to each variable occurring in $\varphi$. Given a theory $T$, we write $\varphi \models_T \psi$ (or simply $\varphi \models \psi$) to denote that the formula $\psi$ is a logical consequence of $\varphi$ in the theory $T$.

A transition system $M$ is a tuple $M = \langle V, I, T \rangle$ where $V$ is a set of (state) variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula representing the transitions. For simplicity, we do not distinguish between state variables and transition variables representing the events of the system. We assume that the latter do not appear primed in the transition formula $T$.

A state of $M$ is an assignment to the variables $V$. We denote with $\Sigma_V$ the set of states. A [finite] path of $M$ is an infinite sequence $s_0, s_1, \ldots$ [resp., finite sequence $s_0, s_1, \ldots, s_k$] of states such that $s_0 \models I$ and, for all $i \geq 0$ [resp., $0 \leq i < k$], $s_i, s_{i+1} \models T$. Given two transitions systems $M_1 = \langle V_1, I_1, T_1 \rangle$ and $M_2 = \langle V_2, I_2, T_2 \rangle$, we denote with $M_1 \times M_2$ the synchronous product $\langle V_1 \cup V_2, I_1 \land I_2, T_1 \land T_2 \rangle$.

Given a Boolean combination $\phi$ of predicates, the invariant model checking problem, denoted with $M \models_{fin} \phi$, is the problem to check if, for all finite paths $s_0, s_1, \ldots, s_k$ of $M$, for all $i$, $0 \leq i \leq k$, $s_i \models \phi$.

Given a LTL formula $\phi$, the LTL model checking problem, denoted with $M \models \phi$, is the problem to check if, for all (infinite) paths $\sigma$ of $M$, $\sigma \models \phi$.

The automata-based approach [44] to LTL model checking is to build a transition system $M_\phi$ with a fairness condition $f_\phi$ such that $M \models \phi$ iff $M \times M_\phi \models \text{FG} f_\phi$. This reduces to finding a counterexample as a fair path, i.e., a path of the system that visits the fairness condition $f_\phi$ infinitely many times. In case of finite-state systems, if the property fails there is always a counterexample in a lasso-shape, i.e., formed by a prefix and a loop.

### 3.5.4 Verification of Infinite-State Systems

SAT-based algorithms take in input a propositional (with Boolean variables) transition system and a property, and try to solve the verification problem with a series of satisfiability queries. These algorithms can be naturally lifted to SMT in order to tackle the verification of infinite-state systems.
IC3 [16] is a SAT-based algorithm for the verification of invariant properties of transition systems. It builds an over-approximation of the reachable state space, using clauses obtained by generalization while disproving candidate counterexamples.

We recently presented in [21] a novel approach, referred to as IC3(IA), to lift IC3 to the SMT case, which is able to deal with infinite-state systems by means of a tight integration with predicate abstraction (PA) [31]. The approach leverages Implicit Abstraction (IA) [41], which allows to express abstract transitions without computing explicitly the abstract system, and is fully incremental with respect to the addition of new predicates. When an abstract counterexample is found, as in Counter-Example Guided Abstraction-Refinement (CEGAR), it is simulated in the concrete space and, if spurious, the current abstraction is refined by adding a set of predicates sufficient to rule it out.

K-LIVENESS [25] is an algorithm recently proposed to reduce liveness (and so also LTL verification) to a sequence of invariant checking steps. Differently from other reductions (such as [40]), it lifts naturally to infinite-state systems without requiring counterexamples to be in a lasso-shape form. K-LIVENESS uses a standard approach to reduce LTL verification for proving that a certain signal \( f \) is eventually never visited (FG\( \neg f \)). The key insight of K-LIVENESS is that, for finite-state systems, this is equivalent to find a \( K \) such that \( f \) is visited at most \( K \) times, which in turn can be reduced to invariant checking.

Given a transition system \( M \), a Boolean combination of predicates \( \phi \), and a positive integer \( K \), for every finite path \( \sigma \) of \( M \), let \( \sigma \models_{\text{fin}} \#(\phi) \leq K \) iff the size of the set \( \{i \mid \sigma[i] \models \phi\} \) is less or equal to \( K \). In [25], it is proved that, for finite-state systems, \( M \models \text{FG} \neg f \) iff there exists \( K \) such that \( M \models_{\text{fin}} \#(f) \leq K \). The last check can be reduced to an invariant checking problem. K-LIVENESS is therefore a simple loop that increases \( K \) at every iteration and calls a subroutine SAFE to check the invariant. In particular, the implementation in [25] uses IC3 as SAFE and exploits the incrementality of IC3 to solve the sequence of invariant problems in an efficient way.

We integrated K-LIVENESS with the algorithm IC3(IA) as SAFE procedure. In this way, we can effectively prove many LTL properties of infinite-state transition systems (although the problem is in general undecidable). The algorithm is available in nuXmv for model checking and in OCRA for checking the refinement of contract. Together with the translation described in D3.3, these tools provide a framework to compositionally verifying LTL properties on MILS-AADL programs.

### 3.5.5 Verification of Real-Time and Hybrid Systems

K-LIVENESS is not complete for infinite-state systems, because even if the property holds, the system may visit the fairness condition an unbounded number of times. Consider for example a system with an integer counter and a parameter \( p \) such that the counter is used to count the number of times the condition \( f \) is visited and once the counter reaches the value of \( p \), the condition is no more visited. This system satisfies \( \text{FG} \neg f \) because for any value of \( p \), \( f \) is visited at most \( p \) times. However, K-LIVENESS will obtain a counterexample to the safety property \( \#(f) \leq K \) for every \( K \), by setting \( p \) to \( K \).

Similarly, K-LIVENESS does not work on the transition system representing a Timed Automaton (TA). In particular, a fair Zeno path forbids K-LIVENESS to prove the property: for every \( K \), the fairness is visited more than \( K \) times, but in a finite amount of (real) time. Removing Zeno paths by
adding an automaton to force progress is not sufficient for Parametric Timed Automata (PTA) and in general hybrid systems. In fact, in these systems a finite amount of time can be bounded by a parameter or a variable that is dynamically set. Therefore, in some cases, there is no $K$ to bound the occurrences of the fairness, although there is no fair non-Zeno path.

In the following, we show how we make K-LIVENESS work on hybrid automata. The goal is to provide a method so that K-LIVENESS checks if there is a bound on the number of times the fairness is visited along a diverging sequence of time points. The essential point is to use a symbolic expression $\beta$ based on the automaton structure to force a minimum distance between two fair time points. We use an additional transition system $Z_\beta$, with a condition $f_Z$, to reduce the problem of proving that $H \models \phi$ to proving that $M_H \times M_{\neg \phi} \times Z_\beta \models \FG \neg f_Z$.

**Linking the fairness to time progress** In this section, we define the transition system $Z_\beta$ that is later used to make K-LIVENESS converge. We first define a simpler version $Z_B$ that works only for timed automata.

Consider the fair transition system $M = M_H \times M_{\neg \phi}$ resulting from the product of the encoding of a Parametric Rectangular Hybrid Automata (PRHA) $H$ and of the negation of the property $\phi$. Let $f$ be the fairness condition of $M$. We build a new transition system $Z_B(f, time)$ that filters the occurrences of $f$ along a time sequence where $time$ values are distant more than $B$ time units. $Z_B(f, time)$ is depicted in Figure 5. It has two locations (represented by a Boolean variable $l$) and a local real variable $t_0$. The initial condition is $l = 0$. The fairness condition $f_Z$ is $l = 1$. The system moves or remains in $l = 0$ keeping $t_0$ unchanged. It moves or remains in $l = 1$ if $f$ is true and $time \geq t_0 + B$ and sets $t_0$ to $time$.

We reduce the problem of checking whether $\phi$ holds in $H$ to checking that the fairness condition $f_Z$ cannot be true infinitely often in $M_H \times M_{\neg \phi} \times Z_B$, i.e. $M_H \times M_{\neg \phi} \times Z_B \models \FG \neg f_Z$.

**Theorem 2** If $B > 0$, $H \models \phi$ iff $M_H \times M_{\neg \phi} \times Z_B \models \FG \neg f_Z$.

We generalize the construction of $Z_B$ considering as bound on time a function $\beta$ over some continuous variables of the model. The new monitor is $Z_\beta(f, time, X)$ shown in Figure 6. It has a local variable $x_0$ for every variable $x$ occurring in $\beta$. $X_0$ is the set of such variables. Now, when $t_0$ is set to $time$, we set also $x_0$ to $x$ and this value is kept until moving to $l = 1$. The condition on time is now $time > t_0 + \beta(X_0)$. It is easy to see that we can still prove that if $\beta(X)$ is always positive, then $H \models \phi$ iff $M_H \times M_{\neg \phi} \times Z_\beta \models \FG \neg f_Z$.

We say that the reduction is complete for K-LIVENESS for a certain class $\mathcal{H}$ of automata iff for every $H \in \mathcal{H}$ there exists $\beta_H$ such that $H \models \phi$ iff there exists $K$ such that $M \times M_{\neg \phi} \times Z_{\beta_H} \models \liminf (f_Z) \leq K$. 
Thus, if $H \models \phi$, and the reduction is complete, and the subroutine SAFE terminates at every call, then K-ZENENESS also terminates proving the property.

The K-ZENO algorithm  The K-ZENO algorithm is a simple extension of K-LIVENESS which, given the problem $H \models \phi$, builds $M = M_H \times M_\phi \times Z_\beta$ and calls K-LIVENESS with inputs $M$ and $f_\pi$. As K-LIVENESS, either K-ZENO proves that the property holds or diverges increasing $K$ up to a certain bound. The crucial part is the choice of $\beta$, because the completeness of the reduction depends on $\beta$. Note that the reduction may be complete, but the completeness of K-ZENO still depends on the completeness of the SAFE algorithm.

As for TAs, we take as $\beta$ the maximum among the constants of the model and 1. For example, consider the TA in Figure 7 (it is actually a compact representation of the TA where $loc1$ is split into two locations corresponding to $b = \top$ and $b = \bot$). It represents an unbounded number of switches of $b$ within 1 time unit. The model satisfies the property $FG_{pc} = loc2$. Taking $\beta = 1$, K-ZENO proves the property with $K = 1$. In fact, starting from the location $loc1$, after 1 time unit, the automaton cannot reach $loc1$ anymore. For PTAs, we consider as $\beta$ the maximum among the parameters, the constants of the model and 1.

We generalize the above idea to consider PRHA with bounded non-determinism. We also assume an endpoint of a flow interval is 0, it cannot be open (must me included in the interval). Guards and invariants of PRHA are conjunctions of inequalities of the form $x \triangleleft\triangleright B$ where $\triangleleft\triangleright \in \{\leq, \geq, <, >\}$. Hereafter, we refer to one of such inequalities as a constraint of the PRHA.

For every constraint $g$ in the form $x \leq B$ or $x < B$ (guard or invariant) of HA, we consider the minimum positive lower bound $r_g$ for the derivative of $x$, if exists. For example, if we have three locations with $\dot{x} \in [1, 2]$, $\dot{x} \in [0, 3]$, $\dot{x} \in [-1, 2]$, we take $r_g = 1$ (since 0 and −1 are not positive). We consider the minimum lower bound $v_g$ for the non-deterministic reset of $x$. For example, if we have three transitions with resets $x' \in [1, 2]$, $x' \in [0, 3]$, $x' \in [-1, 2]$, we take $v_g = -1$. In case $g$ is in the form $x \geq B$ or $x > B$, we define $r_g$ and $v_g$ similarly by considering the maximum negative upper bound of the derivative of $x$ and the maximum upper bound of the reset of $x$. We define the bound $\beta_g(x_0)$ as follows: $\beta_g(x_0) = \max((B - x_0)/r_g, (B - v_g)/r_g)$.

Finally, as $\beta$ we take the maximum among the $\beta_g$ for all $g$ in the automaton $H$ for which $r_g$ exists and the constant 1. Note that this coincides with the $\beta$ defined above for TA and Parametric TA, where $r_g$ is always 1 and $v_g$ is always 0 and $x_0$ is always non negative.

Completeness for Rectangular Hybrid Automata  In this section, we restrict the focus to PRHA that are initialized and have bounded non-determinism. Moreover, we restrict the LTL formula to
have the atoms that predicate over $pc$ only. In this settings, we prove that the reduction to K-
LIVENESS defined in the previous section is complete.

**Theorem 3** If $H \models \phi$, then there exists $K$ such that $M_H \times M_{\neg\phi} \times Z_\beta \models_{\text{fin}} \#(f_Z) \leq K$.

If the hybrid automaton falls outside of the class of initialized PRHA with bounded non-determinism, 
K-ZENO is still sound, but no longer guaranteed to be complete. A simple counterexample is shown 
in Figure 8 in which the stopwatch variable $x$ is not reset when its dynamic changes. The automaton 
satisfies the property ($FG_{\text{good}}$), because the invariant on $x$ and the guard on $y$ make sure that the 
total time spent in $\text{bad}$ is at most 1 time unit. However, K-ZENO cannot prove it with any $K$ because 
time can pass indefinitely in $\text{good}$, while $x$ is stopped. Therefore, it is always possible to visit $\text{bad}$ 
and $f_Z$ an unbounded number of times. Finally, note that K-ZENO is able to prove other properties 
such as for example that the stopwatch automaton satisfies the formula $GF_{\text{good}}$.

**Experimental Evaluation** We have implemented the K-ZENO algorithm on top of the SMT exten-
sion of IC3 described in Section 3.5.4. We remark that, although the completeness results hold only 
for initialized PHRA with bounded non-determinism, our implementation supports a more general 
class of HAs with rectangular dynamics. However, it currently can only be used to verify LTL prop-
erties, and not to disprove them. If a property does not hold, our tool does not terminate. Similarly 
to the Boolean case [25], our implementation consists of relatively few (and simple) lines of code on 
top of IC3.

We tried our approach on various kinds of benchmarks and properties.

- Fischer family benchmarks. 4 different versions (UPPAAL Fischer, Fischer Param, Fischer 
  Hybrid).
- Distributed Controller. $n$ sensors with a preemptive scheduler and a controller.
- Nuclear Reactor models the control of a nuclear reactor with $n$ rods.
- Navigation family benchmarks: movement of an object in an $n \times n$ grid of square cells. Two 
  versions NavigationInit and NavigationFree.
- Diesel Generator: three different versions (small, medium, large).
- Bridge: from the UPPAAL distribution with DIVINE properties.
- Counter to force scalable $K$

Note that the benchmarks fall in different classes: some of them are timed automata (Fischer, Diesel 
Generator, Bridge, Counter), some are parametrized timed automata (Fischer Param, Fischer Fair), 
some are initialized rectangular automata (Fischer Hybrid, Nuclear Reactor), while some have rect-
angular dynamics but are not initialized (Distributed Controller, NavigationInit, NavigationFree).
We manually generated several meaningful LTL properties for the benchmarks of the Fischer family, the Distributed Controller and the Nuclear Reactor. The properties match several common patterns for LTL like fairness (GFp), strong fairness (G(Fp → GFq)), and “leads to” (G(p → Fq)). Moreover, in several cases we added additional fairness constraints to the common patterns to generate properties that hold in the model. For the Bridge and Diesel Generator benchmarks we used the properties already specified in the models. For the navigation benchmark we checked that eventually the object will stay forever in the “stability” region. Finally, we used the property (FG good) in the Counter benchmarks.

In order to evaluate the feasibility of our approach, we have run it on a total of 276 verification tasks, consisting of various LTL properties on the benchmark families described above. Our best configuration could solve 205 instances within the resource constraints (900 seconds of CPU time and 3Gb of memory). If instead we consider the “Virtual Best” configuration, obtained by picking the best configuration for each individual task, our implementation could solve 238 problems. We report details about some of the properties we could prove in Table 1. On each row, the table shows the model name, the class of instances it belongs to (timed, parametric, rectangular, non-initialized rectangular), the property proved (with variables pi’s used as placeholders for atomic propositions), the size of the symbolic encoding (number of Boolean and Real variables, and number of nodes in the formula DAG of the transition relation), the value of k reached by K-LIVENESS and the total execution time. We remark that we are not aware of any other tool capable of verifying similar kinds of LTL properties on the full class of instances we support.

We analyze the performance impact of different heuristics and implementation choices along the following dimensions:

- Invariant checking engine. We have two versions of SMT-based IC3, one based on approximated preimage computations with quantifier elimination (called IC3(QE) here), and one based on implicit predicate abstraction (IC3(IA)). The results shown in Table 1 indicate that IC3(IA) is generally superior to IC3(QE) on software verification benchmarks. However, the situation is less clear in the domain of timed and hybrid systems.

1 On most of the instances the value of k reached by K-LIVENESS is small. The explanation is that, on real models, the number of constraints that must be violated inside a loop that contains f→φ before time diverges is usually low. The benchmarks of the Counter family were created on purpose, to show that k can increase arbitrarily.

### Table 1: Selected experimental results.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Class</th>
<th>Property</th>
<th># Bool vars</th>
<th># Real vars</th>
<th>Trans size</th>
<th>k</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer (8 processes)</td>
<td>T</td>
<td>(∩i=1 GFpi) → G(!p18 → Fp18)</td>
<td>132</td>
<td>20</td>
<td>1286</td>
<td>3</td>
<td>6.37</td>
</tr>
<tr>
<td>Fischer Fair (2 processes)</td>
<td>P</td>
<td>(p1 ∧ GFp2) → G(p3 → Fp4)</td>
<td>38</td>
<td>12</td>
<td>622</td>
<td>4</td>
<td>76.14</td>
</tr>
<tr>
<td>Fischer Hybrid (10 processes)</td>
<td>R</td>
<td>(GFp1 ∧ GFp2 ∧ GFp3) → G(p4 → Fp5)</td>
<td>106</td>
<td>64</td>
<td>8759</td>
<td>1</td>
<td>325.03</td>
</tr>
<tr>
<td>Dist Controller (3 sensors)</td>
<td>N</td>
<td>G(p1) → (GFp2)</td>
<td>58</td>
<td>27</td>
<td>1737</td>
<td>1</td>
<td>397.24</td>
</tr>
<tr>
<td>Nuclear Reactor (9 rods)</td>
<td>R</td>
<td>G(p1) → Fp2</td>
<td>82</td>
<td>24</td>
<td>3258</td>
<td>1</td>
<td>530.40</td>
</tr>
<tr>
<td>NavigationInit (3x3)</td>
<td>N</td>
<td>FG(p1 ∨ p2 ∨ p3 ∨ p4)</td>
<td>16</td>
<td>8</td>
<td>808</td>
<td>2</td>
<td>4.37</td>
</tr>
<tr>
<td>NavigationInit (10x10)</td>
<td>N</td>
<td>FG(p1 ∨ p2 ∨ p3 ∨ p4)</td>
<td>22</td>
<td>8</td>
<td>4030</td>
<td>2</td>
<td>453.74</td>
</tr>
<tr>
<td>NavigationFree (3x3)</td>
<td>N</td>
<td>FG(p1 ∨ p2 ∨ p3 ∨ p4)</td>
<td>16</td>
<td>8</td>
<td>808</td>
<td>2</td>
<td>3.37</td>
</tr>
<tr>
<td>NavigationFree (9x9)</td>
<td>N</td>
<td>FG(p1 ∨ p2 ∨ p3 ∨ p4)</td>
<td>22</td>
<td>8</td>
<td>3461</td>
<td>2</td>
<td>872.07</td>
</tr>
<tr>
<td>Counter 10</td>
<td>T</td>
<td>FGp</td>
<td>10</td>
<td>24</td>
<td>294</td>
<td>10</td>
<td>52.74</td>
</tr>
<tr>
<td>Diesel Gen (small)</td>
<td>T</td>
<td>G(p1 → F(¬p2 ∨ p3))</td>
<td>84</td>
<td>24</td>
<td>724</td>
<td>1</td>
<td>16.55</td>
</tr>
<tr>
<td>Diesel Gen (medium)</td>
<td>T</td>
<td>G(p1 → F(¬p2 ∨ p3))</td>
<td>140</td>
<td>30</td>
<td>1184</td>
<td>1</td>
<td>51.24</td>
</tr>
<tr>
<td>Diesel Gen (large)</td>
<td>T</td>
<td>G(p1 → F(¬p2 ∨ p3 ∨ p4))</td>
<td>264</td>
<td>62</td>
<td>2567</td>
<td>1</td>
<td>538.39</td>
</tr>
</tbody>
</table>

Incrementality. We compare our fully-incremental implementation of K-LIVENESS to a non-incremental one, in which IC3 is restarted from scratch every time the K-LIVENESS counter is incremented.

Initial value of the K-LIVENESS counter. We consider the impact of starting the search with a right (or close to) value for the K-LIVENESS counter $k$, instead of always starting from zero, in IC3. For this, we use a simple heuristic that uses BMC to guess a value for the counter: we run BMC for a limited time (20 seconds in our experiments), increasing $k$ every time a violation is detected. We then start IC3 with the $k$ value found.

Overall, we considered six different configurations: IC3(IA) and IC3(QE) are the default, incremental versions of K-LIVENESS with IC3, using either approximate quantifier elimination or implicit abstraction; IC3(IA)-NOINCR and IC3(QE)-NOINCR are the non-incremental versions; BMC+IC3(IA) and BMC+IC3(QE) are the versions using a time-limited initial BMC run for computing an initial value for the K-LIVENESS counter $k$. The six configurations are compared in Figure 9, showing the number of instances solved (y-axis) and the total execution time (x-axis). The figure also includes the “Virtual Best” configuration, constructed by taking the best result for each individual instance.

Figure 9 shows that, differently from the case of software verification, the default version of IC3(QE) performs much better than IC3(IA). Although we currently do not have a clear explanation for this, our conjecture is that this is due to the “bad quality” of the predicates found by IC3(IA) in the process of disproving invariants when the value of $k$ is too small. Since IC3(IA) never discards predicates, and it only tries to add the minimal amount of new predicates when performing refinements, it might simply get lost in computing clauses of poor quality due to the “bad” language of predicates found. This might also be the reason why IC3(IA)-NOINCR performs better than IC3(IA), despite the runtime cost of restarting the search from scratch every time $k$ changes: when restarting, IC3(IA)-NOINCR can also throw away bad predicates. A similar argument can also be applied to BMC+IC3(IA): using BMC to skip the bad values of $k$ allows IC3(IA) to find predicates that are more relevant/useful for proving the property with the good (or close to) value of $k$.

The situation for IC3(QE) is instead completely different. In this case, not only turning off incrementality significantly hurts performance, as we expected, but also using BMC is detrimental. This is consistent with the behavior observed in the finite-state case for the original K-LIVENESS imple-
We conclude our evaluation with a comparison of our implementation with alternative tools and techniques working on similar systems. As already remarked above, we are not aware of any tool that is able to handle arbitrary LTL properties on the class of systems that we support. Therefore, we concentrate our comparison only on Timed Automata, comparing with DiVINE [7]. We use a total of 64 instances from the Fischer, Bridge and Counter families. Unfortunately, we could not include the industrial Diesel Generator model, since it is modeled as a symbolic transition system, whereas DiVINE expects a network of timed automata (in UPPAAL format) as input. However, the Diesel Generator benchmark was reported to be very challenging for explicit-state approaches [33].

The results are shown in Figure 10, where we compare DiVINE with our two best configurations, BMC+IC3(IA) and IC3(QE). We can see that DiVINE is very fast for simple instances, outperforming our tool by orders of magnitude. However, its performance degrades quickly as the size of the instances increases. In contrast, both BMC+IC3(IA) and IC3(QE) scale better to larger instances. This is particularly evident for BMC+IC3(IA): after having found a good initial value for the K-LIVENESS counter with BMC, IC3(IA) can solve almost all the instances in just a few seconds.

### 3.5.6 Safety

In the previous sections, we proposed to specify functional, real-time, security, and safety properties and contracts in temporal logic and described the algorithms that we implemented to verify the contract refinement in a system architecture. However, not all requirements can be expressed in this way. We have already seen in the previous sections how we cope with security properties such as non-interference. As for safety, some requirements specify that other properties must be satisfied also in case of component failures or that there does not exist any single point of failure. In order to ensure that the system satisfies this kind of requirements, typically safety analysis techniques such as fault-tree analysis are employed.

Here, we detail how we can exploit the contract refinement for contract-based fault-tree analysis. The technique is described in detail in [14]. In D-MILS, we adapt it to the logic used to express the properties.

The idea of contract-based fault-tree analysis is that the contract refinement is automatically extended to consider the possibility that some components or the environment may fail in ensuring the respec-
D4.5 Compositional verification techniques and tools for distributed MILS

...utive properties. Automatic techniques analyze the dependencies among such failures and shape them in form of a fault tree.

More specifically, for each component $c$ and for each contract $C$, two failure ports are automatically added: one, named $c.C - FAILURE_O$, representing the failure of the component implementation to satisfy the guarantee, the other, name $c.C - FAILURE_I$ representing the failure of the component environment to satisfy the assumption. The automatic fault-tree generation finds, for every refinement component $c$, which combination of failures of the subcomponents of $c$ and failure of the environment of $c$ may cause $c.C - FAILURE_O$, and similarly for every subcomponent $c'$ of $c$, which combination of failures of the sibling subcomponents of $c'$ and failure of the environment of $c'$ may cause $c.C - FAILURE_I$. The result can be shown in a hierarchical fault tree whose top-level event is the failure of a system contract and the leaves are the failure of the leaf components and the failure of the system environment.

3.5.7 Starlight example

Architecture In this section, we exemplify the approach on an example taken from the literature [3, 19]. The Starlight Interactive Link is a dispatching device developed by the Australian Defense Science and Technology Organization to allow users to establish simultaneous connections to high-level (classified) and low-level networks. The idea is that the device acts as a switch that the user can control to dispatch the keyboard output to either a high-level server or to a low-level server. The user can use the low-level server to browse the external world, send messages, or have data sent to the high-level server for later use.

Figure [1] shows the architecture of the Starlight Interactive Link as formalized in D-MILS. The components $H$ and $L$ represent respectively the high-level and low-level networks. The low-level network can exchange information with the external world. The component $D$ represents the Starlight device, which receives commands from the user and dispatches the commands to $H$ or to $L$ based on an internal state. The state is changed with two switch commands, namely switch_to_low and switch_to_high. The original architecture has only the blue components, with $D$ in place of $E$. We extended this architecture with a safety mechanism to make the system “fail-secure” with respect to failures of the dispatcher: the dispatcher is extended with a monitor $M$; the communication of the dispatcher to $L$ is filtered by $M$ that in case of failure of $D$ blocks the communication. To avoid confusion we refer to the actual device that is filtered by $M$ as the dispatcher ($D$), while to the component consisting of $D$ and $M$ as the extended dispatcher $E$.

System contract The architecture has been enriched with contracts that formalize the functional requirements to ensure that the system responds correctly to the user commands, and the security requirement that there is no leakage of high-level data. Here, we focus on the latter, which says:

Req-Sys-secure: No high-level data shall be sent by $L$ to the external world.

The architecture ensures [Req-Sys-secure] assuming the following requirement on the user:

Req-User-secure: The user shall switch the dispatcher to high before entering high-level data.
Moreover, we consider the following safety requirement:

**Req-Sys-safe**: No single failure shall cause a loss of **Req-Sys-secure**.

We formalized the requirements of the system and of the components using OCRA contracts. In the following, we use the concrete syntax accepted by the tool. We briefly clarify the used notation: “and”, “or”, “not”, “implies” are standard Boolean operators; “always”, “never”, “since”, “in the past” are standard temporal operators of LTL with past also referred to with the mathematical notation $G$, $G\neg$, $S$, $O$; “last_data” is a built function to refer to the last data passed by the event of a event data port; italics names refer to ports or uninterpreted functions declared in the model.

The requirements **Req-Sys-secure** and **Req-User-secure** have been formalized into the FO-LTL formulas:

**Formal-Sys-secure**: never $\text{is\_high}(\text{last\_data(outL)})$

**Formal-User-secure**: always $((\text{is\_high}(\text{last\_data(cmd)})) \implies (\text{not switch\_to\_low} \text{ since switch\_to\_high}))$

Note that the formalization of **Req-User-secure** improves the informal requirement, which is not precise. A literal formalization would be:

**Formal-User-secure-wrong**: always $((\text{is\_secure}(\text{last\_data(cmd)})) \implies (\text{in\ the\ past switch\_to\_high}))$
but this is wrong, because we have to ensure that the last switch was a `switch_to_high`, without a more recent `switch_to_low`. We can actually improve the informal requirement as:

\[\text{Req-User-secure-new: Whenever the user sends commands with high data, she shall previously issue a `switch_to_high` and no `switch_to_low` since the last `switch_to_high`}\]

which is formalized by \[\text{Formal-User-secure}\].

Note that while \[\text{Req-Sys-secure}\] is a requirement on the implementation of the Starlight system, \[\text{Req-User-secure}\] is actually a requirement on its environment (the user). This is reflected by the system contract specification, which sets \[\text{Formal-Sys-secure}\] as the guarantee and \[\text{Formal-User-secure}\] as the assumption of the system contract.

**Component contracts** The dispatcher ensures the system security requirement with the following local requirement:

\[\text{Req-D-low-mode: The dispatcher shall send commands to } L \text{ only if the last switch was a `switch_to_low` and the input command has been received after.}\]

formalized into:

\[\text{Formal-D-low-mode: always (cmdL implies ((not switch_to_high) since switch_to_low) and (not switch_to_low) since cmd))}\]

In order to fulfill requirement \[\text{Req-Sys-safe}\], we also filter the commands to \(L\) by a monitor \(M\), which has a requirement \[\text{Req-M-low-mode}\] identical to \[\text{Req-D-low-mode}\] and formalized in the same way. Thus, \(D\) passes also the switches to the monitor and must ensure the following requirement:

\[\text{Req-D-fw-switch: Whenever the dispatcher receives a `switch_to_high`, it shall pass it to } M \text{ before doing any other actions and it sends a `switch_to_low` to } M \text{ only if the last received switch was a `switch_to_low`}\]

formalized into:

\[\text{Formal-D-fw-switch: always ((switch_to_high implies ((not (cmdH or cmdL or return or monitor_switch_to_low)) until monitor_switch_to_high)) and (monitor_switch_to_low implies ((not switch_to_high) since switch_to_low)))}\]

Finally, in order to make the refinement correct, we must require all components to not invent high data. We express this by requiring that \(D, M,\) and \(L\) only pass the data that they have received. Thus, for \(D\), we require that:

\[\text{Req-D-data: } D \text{ shall pass to } cmdL \text{ only the data that has been received with last } cmd.\]

formalized into:

\[\text{Formal-D-data: always ((cmdL implies ((in the past cmd) and (last_data(cmdL) = last_data(cmd)))}}\]

The requirements \[\text{Req-M-data}\] and \[\text{Req-L-data}\], of \(M\) and \(L\) respectively, are analogous. Note that these formulas are actually guarantees of corresponding contracts, without assumptions (i.e. assumptions equal to \textit{true}).

\[\text{As suggested by one of the reviewers, in an alternative model, we could use only one event data port instead of two switch events and ensure that the last switch was high.}\]
D4.5 Compositional verification techniques and tools for distributed MILS

**Analysis results** Given the above contract specifications, OCRA can prove the system `Req-Sys-secure` assuming `Req-User-secure` is correctly refined by the contracts of `D`, `M`, and `L` (see [24] for more details on the technique). One can also show that by using `Formal-User-secure-wrong` instead of `Formal-User-secure` the refinement is not correct and yields a counterexample trace execution.

In order to prove `Req-Sys-safe`, we use OCRA to produce a fault tree showing the dependency of the system failure on the failure of the components (see [14] for more details on the technique). The generated fault tree is exhibited in Figure 12. It shows that neither `Req-D-low-mode` nor `Req-M-low-mode` are single points of failure. Instead, `Req-D-data` and `Req-M-data` are single points of failure. While the failure of `Req-L-data` does not represent real threats since `L` never receives high data, the failure of `Req-D-data` and `Req-M-data` could result in `D` or `M` sending information that had been temporary stored in a buffer used for handling multiple requests or in a cache for improving performance. This can be solved for example by ensuring that such memories are deleted before every `switch_to_low` is completed.

### 3.6 Compositional Verification for Deadlock Freedom of Timed Systems with RT-DFinder

We recall hereafter the method for compositional generation of invariants for timed systems introduced in [5].

The challenge is to approach the state-space explosion problem when model-checking timed systems with great number of components. Our solution consists in breaking up the verification task by proposing a fully automatic and compositional method. To do this, we exploit system invariants. In contrast to exact reachability analysis, invariants are symbolic approximations of the set of reachable states of the system. We show that rather precise invariants can be computed compositionally, from the separate analysis of the components in the system and from their composition glue. This method is proved to be sound for the verification of safety properties. However, it is not complete.

The starting point is the verification method of D-Finder [13, 11, 12], summarised in Figure 13. The method exploits compositionality as explained next. Consider that a system consists of components `B_i` interacting by means of a set `γ` of multi-party interactions, and let `Ψ` be a system property of interest. Assume that all `B_i` as well as the composition through `γ` can be independently characterised by means of component invariants `CI(B_i)`, respectively interaction invariants `II(γ)`. The connection between the invariants and the system property `Ψ` can be intuitively understood as follows: if `Ψ` can be proved to be a logical consequence of the conjunction of components and interaction invariants, then `Ψ` holds for the system.

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 “\( B \models \Box \ Ψ \)” is to be read as “\( Ψ \) holds in every reachable state of \( B \)”.

The verification rule (VR) has been developed in [13, 12] for untimed systems. Its direct application to timed systems may be weak as interaction invariants do not capture global timings of interactions between components. The key contribution of this method is to improve the invariant generation method so to better track such global timings by means of auxiliary history clocks for actions and interactions. At component level, history clocks expose the local timing constraints relevant to the
D4.5 Compositional verification techniques and tools for distributed MILS

Figure 12: Fault-tree generated from the contract refinement. Events are labeled with the name of the component instance followed by the name of the contract, followed either by FAILURE_O, which represents the failure of the component to satisfy the contract, or by FAILURE_I, which represents the failure of the component environment to satisfy the contract assumption.

\[
\vdash \prod_i CI(B_i) \land II(\gamma) \rightarrow \Psi
\]

\[
\|_\gamma B_i \models \Box \Psi \quad \text{(VR)}
\]

Figure 13: Compositional Verification Rule

interactions of the participating components. At composition level, extra constraints on history clocks are enforced due to the simultaneity of interactions and to the synchrony of time progress.

We concretize the component and interaction invariants for the example shown in Figure 14. This example depicts a “controller” component serving \( n \) “worker” components, one at a time. The interactions between the controller and the workers are defined by the set of synchronisations
Periodically, after every 4 units of time, the controller synchronises its action $a$ with the action $b_i$ of any worker $i$ whose clock shows at least $4n$ units of time. Initially, such a worker exists because the controller waits for $4n$ units of time before interacting with workers. The cycle repeats forever because there is always a worker “willing” to do $b$, that is, the system is deadlock-free. Nonetheless, proving deadlock-freedom of the system requires to establish that when the controller is at location $lc_1$ there is at least one worker such that $y_1 - x \geq 4n - 4$. Unfortunately, this property cannot be shown if we use (VR) as it is demonstrated in [13].

To give a logical characterisation of components and interactions, we use invariants. An invariant $\Phi$ is a state property which holds in every reachable state of a component (or of a system) $B$, in symbols, $< B, s_0 > \models \Phi$. We use $CI(< B_1, s_0 >)$ and $II(\gamma, s_0)$, to denote component, respectively interaction invariants, where $s_0$ is the initial symbolic state of the system. For component invariants, our choice is to work with their reachable set. More precisely, for a component $B$ with initial state $s_0$, $CI(< B, s_0 >)$ is the disjunction of $(l \land \zeta)$ and where $l$ is a location and $\zeta$ is a clock difference constraint (a zone in timed automata terminology). To ease the reading, we abuse of notation and use $l$ as a place holder for a state predicate “at(l)” which holds in any symbolic state with location $l$, that is, the semantics of $at(l)$ is given by $(l, \zeta) \models at(l)$. As an example, the component invariants for the scenario in Figure 14 with one worker are:

$$CI(< Controller, (lc_0, x = 0 ) > ) = (lc_0 \land x \geq 0) \lor (lc_1 \land x \leq 4) \lor (lc_2 \land x \geq 0)$$

$$CI(< Worker_1, (l_1^1, y = 0 ) > ) = (l_1^1 \land y_1 \geq 0) \lor (l_2^1 \land y_1 \geq 4)$$

The interaction invariants we use are computed by the method explained in [13]. Interaction invariants are over-approximations of global state spaces allowing us to disregard certain tuples of local states as unreachable. We want to compute the interaction invariant for the running example when the controller is interacting with one worker. We assume that the controller is initially at $lc_0$ and worker1 is initially at location $l_1^1$. In addition, the system clocks are supposed to be initially equal, thus the initial global state of the system is: $s_0 = ((lc_0, l_1^1), x = y = 0)$. The interaction invariant $II\left(\{(a \mid b_1), (c \mid d_i)\}, s_0\right)$ is: $(l_1^1 \lor l_2^1) \land (lc_2 \lor l_1^1) \land (l_2^1 \lor lc_0 \lor l_1^1) \land (lc_2 \lor lc_0 \lor lc_1)$.

### 3.6.1 Timed Invariant Generation

As explained earlier, a direct application of the rule (VR) may not be useful in itself in the sense that the component and the interaction invariants alone are usually not enough to prove global properties,
especially when the properties involve relations between clocks of different components. More precisely, though component invariants encode timings of local clocks, there is nothing to constrain the bounds on the differences between clocks in different components. To give a concrete illustration, consider the safety property \( \Psi_{\text{Safe}} = (l_{c1} \land l_{c1}^{1} \rightarrow x \leq y_{1}) \) that holds in the running example with one worker. We note that if this property is satisfied, it is guaranteed that the global system is not deadlocked when the controller is at location \( l_{c1} \) and the worker is at location \( l_{c1}^{1} \). It is not difficult to see that \( \Psi_{\text{Safe}} \) cannot be deduced from \( CI(<\text{Controller}, s_{0}^{c}>) \land CI(<\text{Worker}_{1}, s_{1}^{1}>) \land II(\{(a | b_{1}), (c | d_{1})\}, s_{0}) \).

### History Clocks for Actions

In this section, we show how we can, by means of some auxiliary constructions, apply (VR) more successfully. To this end, we “equip” components (and later, interactions) with history clocks, a clock per action; then, at interaction time, the clocks corresponding to the actions participating in the interaction are reset. This basic transformation allows us to automatically compute a new invariant of the system with history clocks. This new invariant, together with the component and interaction invariants, is shown to be, after elimination of history clocks, an invariant of the initial system.

We reconsider the sub-system from Figure [4]. We illustrate how the safety property \( \Psi_{\text{Safe}} \) introduced in the beginning of the section can be shown to hold by using the newly generated invariant. The history clocks are \( h_{a} \) and \( h_{c} \) for the controller, and \( h_{b_{1}} \) and \( h_{d_{1}} \) for the worker. \( h_{\text{init}} \) measures the time elapsed since the start. The invariants for the components with history clocks are:

\[
CI(<\text{Controller}^{h}, s_{0}^{c,h}>) = (l_{c0} \land x = h_{\text{init}} < h_{a} \land h_{c} > h_{\text{init}}) \lor \\
(l_{c1} \land x \leq h_{\text{init}} - 4 \land x \leq 4 \land h_{a} > h_{\text{init}} \land h_{c} > h_{\text{init}}) \lor \\
(l_{c2} \land x \leq 4 \land h_{a} \leq h_{\text{init}} - 8) \lor \\
(l_{c2} \land x > h_{\text{init}} - 8 \land h_{a} = x \land h_{c} > h_{\text{init}})) \\
(l_{c2} \land x = h_{a} \land h_{c} = h_{a} + 4 \leq h_{\text{init}} - 8)
\]

Where, \( s_{0}^{c,h} \) and \( s_{0}^{c,h} \) denote initial symbolic states of \( \text{Controller}^{h} \) and \( \text{Worker}_{1}^{h} \) respectively. \( s_{0}^{c,h} = (l_{c0}, x = h_{\text{init}} = 0 \land h_{a} > h_{\text{init}} \land h_{c} > h_{\text{init}}) \) and \( s_{0}^{h} = (l_{1}^{h}, y_{1} = h_{\text{init}} = 0 \land h_{b_{1}} > h_{\text{init}} \land h_{d_{1}} > h_{\text{init}}) \)

\[
CI(<\text{Worker}_{1}^{h}, s_{0}^{c,h}>) = (l_{1}^{h} \land y_{1} = h_{\text{init}} < h_{d_{1}} \land h_{b_{1}} > h_{\text{init}}) \lor \\
(l_{1}^{h} \land y_{1} = h_{b_{1}} \leq h_{\text{init}} - 4) \lor \\
(l_{1}^{h} \land y_{1} = h_{\text{init}} \geq h_{b_{1}} + 4 \land h_{d_{1}} > h_{\text{init}})) \lor \\
(l_{1}^{h} \land y_{1} = h_{d_{1}} \leq h_{\text{init}} - 4 \land h_{b_{1}} \leq h_{d_{1}} - 4)
\]

By using the interaction invariant and the equality constraints \( \mathcal{E}(\gamma) \), after the elimination of the existential quantifiers in \( (\exists h_{a} \exists h_{b_{1}} \exists h_{c} \exists h_{d_{1}}) CI(<\text{Controller}^{h}, s_{0}^{c,h}>) \land CI(<\text{Worker}_{1}^{h}, s_{0}^{c,h}>) \land II(\{(a | b_{1}), (c | d_{1})\}, s_{0}) \land \mathcal{E}(\gamma)) \), we obtain the following invariant \( \Phi \):

\[
\Phi = (l_{1}^{h} \land l_{c0} \land x = y_{1}) \lor (l_{1}^{h} \land l_{c1} \land (y_{1} = x \lor y_{1} \geq x + 4)) \lor
\]
Here, $s_{0}^{h}$ and $s_{0}^{h}$ denote initial symbolic states of $Controller^{h}$ and $Worker^{h}$ respectively. It can be easily checked that $\Phi \land \neg \Psi_{Safe}$ has no satisfying model and this proves that $\Psi_{Safe}$ holds for the system. We used bold fonts in $\Phi$ to highlight relations between $x$ and $y_{1}$ which are not in $CI(<Controller, s_{0}^{h}>) \land CI(<Worker_{1}, s_{1}^{h}>)$ $\land$ $II(\gamma, s_{0})$.

**History Clocks for Interactions**

The equality constraints on history clocks allow to relate the local constraints obtained individually on components. In the case of non-conflicting interactions, the relation is rather “tight”, that is, expressed as conjunction of equalities on history clocks. In contrast, the presence of conflicts lead to a significantly weaker form. Intuitively, every action in conflict can be potentially used in different interactions. The uncertainty on its exact use leads to a disjunctive expression.

Nonetheless, the presence of conflicts themselves can be additionally exploited for the generation of new invariants. That is, in contrast to equality constraints obtained from interaction, the presence of conflicting actions enforce disequalities (or separation) constraints between all interactions using them. In what follows, we show how to automatically compute such invariants enforcing differences between the timings of the interactions themselves. To effectively implement this, we proceed in a similar manner as in the previous section: we again make use of history clocks and corresponding resets but this time we associate them to interactions, at the system level. We use them to express additional constraints on their timing. The starting point is the observation that when two conflicting interactions compete for the same action $a$, no matter which one is first, the latter must wait until the component which owns $a$ is again able to execute $a$. This is referred to as a “separation constraint” for conflicting interactions.

In our running example the only shared actions are $a$ and $c$ within the controller, and both $k_{a}$ and $k_{c}$ are equal to 4, thus the expression of the separation constraints reduces to:

$S(\gamma) \equiv \bigwedge_{i \neq j} |h_{c|d_{i}} - h_{c|d_{j}}| \geq 4 \land \bigwedge_{i \neq j} |h_{a|i_{b_{i}}} - h_{a|i_{b_{j}}}| \geq 4$.

The invariant $S(\gamma)$ is defined over the history clocks for interactions. Previously, the invariant $E(\gamma)$ has been expressed using history clocks for actions. In order to “glue” them together in a meaningful way, we need some connection between history and interaction clocks. This connection is formally addressed by the constraints $E^{*}$. For our running example, $E^{*}$ is:

$E^{*}(\gamma) \equiv h_{b_{1}} = h_{a|i_{b_{1}}} \land h_{b_{2}} = h_{a|i_{b_{2}}} \land h_{a} = \min_{i=1,2}(h_{a|i_{b_{i}}}) \land h_{d_{1}} = h_{c|d_{1}} \land h_{d_{2}} = h_{c|d_{2}} \land h_{c} = \min_{i=1,2}(h_{c|d_{i}})$

The predicate $\exists H_{A} \exists H_{\gamma}.(\bigwedge_{i} CI(<B_{i}^{h}, s_{0}^{h}>) \land II(\gamma, s_{0}) \land E^{*}(\gamma) \land S(\gamma))$ is an invariant of $<\ll B_{i}, s_{0} >$. This new invariant is in general stronger than $\exists H_{A}.(\bigwedge_{i} CI(<B_{i}^{h}, s_{0}^{h}>) \land II(\gamma, s_{0}) \land E(\gamma))$ and it provides better state space approximations for timed systems with conflicting interactions.
To get some intuition about the invariant generated using separation constraints, let us reconsider the running example with two workers. The subformula which we emphasize here is the conjunction of $E^*$ and $S$. The interaction invariant is:

$$II(\gamma) = (l_1^1 \vee l_2^1) \land (l_1^2 \vee l_2^2) \land (l_0 \land l_1 \land l_2) \land (l_1^1 \land l_1^2 \land l_2^1 \land l_2^2)$$

$$\land (l_2 \land l_1^1 \land l_1^2) \land (l_2^1 \land l_1 \land l_2) \land (l_0^1 \land l_1^1 \land l_2^1)$$

The components invariants are:

$$CI(<\text{Controller}^h, s_0^{i,h}>) = (l_0 \land x = h_{\text{init}} < h_a \land h_c > h_{\text{init}}) \lor$$

$$(l_1 \land x \leq h_{\text{init}} - 8 \land x \leq 4 \land h_a > h_{\text{init}} \land h_c > h_{\text{init}}) \lor$$

$$(l_2 \land x \leq 4 \land x = h_c \leq h_a \leq h_{\text{init}} - 12) \lor$$

$$(l_2 \land x \leq h_{\text{init}} - 12 \land h_a = x \land h_c > h_{\text{init}})) \lor$$

$$(l_2 \land x = h_a \land h_c = h_a + 4 \leq h_{\text{init}} - 12)$$

Where, $s_0^{i,h}$ and $s_i^{i,h}$ denote initial symbolic states of Controller$^h$ and Worker$^h_i$ respectively.

$s_0^{i,h} = (l_0, x = h_{\text{init}} = 0 \land h_a \geq 0 \land h_c \geq 0)$ and $s_i^{i,h} = (l_1^i, y_i = h_{\text{init}} = 0 \land h_b \geq 0 \land h_d \geq 0)$

$$CI(<\text{Worker}^h_i, s_i^{i,h}>) = (l_1^i \land y_i = h_{\text{init}} < h_d \land h_b > h_{\text{init}}) \lor$$

$$(l_1^i \land y_i = h_d \land y_i \leq h_{\text{init}} - 8) \lor$$

$$(l_2^i \land y_i = h_d \land y_i \geq h_b + 8 \land h_d > h_{\text{init}}) \lor$$

$$(l_2^i \land y_i = h_d \land y_i \leq h_{\text{init}} - 8 \land h_b \geq h_d - 8)$$

by recalling the expression of $S(\gamma)$ from the running example we obtain that $\exists H_\gamma. E^* (\gamma) \land S(\gamma) \equiv |h_{b_2} - h_{b_1}| \geq 4 \land |h_{d_2} - h_{d_1}| \geq 4$ and thus, after quantifier elimination in $\exists H_\gamma \exists H_\gamma. (CI(<$ Controller$^h,h,s_0^{i,h}>) \land II(\gamma,s_0)) \land E^* (\gamma) \land S(\gamma))$, we obtain the following invariant $\Phi$:

$$\Phi = (l_1^1 \land l_2^1 \land l_0 \land x = y_1 = y_2) \lor$$

$$(l_1^1 \land l_2^1 \land l_1 \land x \leq 4 \land (y_1 = x \land y_2 - y_1 \geq 4 \lor y_1 \geq x + 8 \lor y_2 \geq 4) \lor$$

$$y_2 = x \land y_1 - y_2 \geq 4 \lor y_2 \geq x + 8) \lor$$

$$(l_1^1 \land l_2^1 \land l_2 \land (y_1 \geq x + 8 \lor (y_2 = x + 4 \land y_1 - y_2 \geq 4)) \lor$$

$$(y_2 \geq x + 4 \land y_2 - y_1 \geq 4) \lor$$

$$(l_2^1 \land l_2^1 \land l_2 \land (y_2 \geq x + 8 \lor (y_1 = x + 4 \land y_2 - y_1 \geq 4)))$$

We emphasized in the expression of $\Phi$ the newly discovered constraints. All in all, $\Phi$ is strong enough to prove that the system is deadlock free.

### 3.6.2 Implementation and Experiments

The method has been implemented in a Scala prototype which is freely available at [http://www-verimag.imag.fr/~lastefan/tas/index.html](http://www-verimag.imag.fr/~lastefan/tas/index.html). The method is currently being...
implemented in RT-DFinder, a tool for the verification of timed systems expressed in Real-Time BIP [1]. It takes as input the components $B_i$, an interaction set $\gamma$ and a global safety property $\Psi$ and checks whether the system satisfies $\Psi$. It generates Z3 [2] Python code to check the satisfiability of the formula $\land_i CI(B_i) \land HI(\gamma) \land \Phi^* \land \neg\Psi$ where $\Phi^*$, depending on whether $\gamma$ is conflicting, stands for $E(\gamma)$ or $E^*(\gamma) \land S(\gamma)$. If the formula is not satisfiable, the prototype returns no solution, that is, the system is guaranteed to satisfy $\Psi$. Otherwise, it returns a substitution for which the formula is satisfiable, that is, the conjunction of invariants is true while $\Psi$ is not. This substitution may correspond to a false positive in the sense that the state represented by the substitution could be unreachable.

We have experimented our prototype on several classical benchmarks; temperature control system, acyclic Fischer protocol and train gate controller system. These use-cases are detailed in [5].

<table>
<thead>
<tr>
<th>Model &amp; Property</th>
<th>Size (N)</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Our Prototype</td>
</tr>
<tr>
<td>Fischer &amp; mutual exclusion</td>
<td>1</td>
<td>0.156</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>2.840</td>
</tr>
<tr>
<td>Train Gate Controller &amp; mutual exclusion</td>
<td>2</td>
<td>0.176</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.356</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.484</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4.416</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>15.765</td>
</tr>
<tr>
<td>Temperature Controller &amp; absence of deadlock</td>
<td>2</td>
<td>0.144</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.132</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.308</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4.056</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>19.22</td>
</tr>
</tbody>
</table>

Table 2: Verification times

In Table 2 we compare the verification time. Except from the train-gate-controller example, where Uppaal verifies the property in less time thanks to the use of some specific reduction techniques, our prototype succeeds in checking all of these systems with big number of components. In particular, it succeeded in checking two of these systems, where Upppal was blocked for great number of components.
3.6.3 Extension for Parametric Timed Systems

The verification method above has been generalized [6] to uniform verification of parametric timed systems (PTS), that is, checking that a safety state property hold for a system containing $n$ identical timed components regardless of the value of $n$. The original method being compositional, it suits quite well parameterised systems in that it presents the advantage of reusing existing local characterisations at the global level of system characterisation. Applying the method in the context of PTSs boils down to giving an effective method of checking the validity of quantified formulae. This is not obvious because, for instance, very effective solvers like Z3 fail while trying to prove/disprove them. At a first thought, one could apply some tactics which make extensive use of transitivity and practically reduce the formula to a tautology. However, the simpler and more inspired solution presented in [6] makes use of the small model theorem from [32]. The advantage of combining such result with the compositional method is twofold. On the one hand it can be the case that the system without replicas is big enough to make the construction of the product infeasible. On the other hand, a direct consequence of the modelling choices adopted in our framework leads to an elegant application of the presented method to parameterised timed systems where interactions are given by various types of topologies which extend the standard binary synchronous communication from [32].

3.7 Monolithic Verification with COMPASS

An alternative way to verify the system properties is to use the monolithic verification provided by COMPASS (see also Section 4.5). The approach relies on the extension of COMPASS to support the MILS-AADL language. Properties can be formalized as invariant or temporal formulas in the form of patterns. The properties can then be verified using model checking in a monolithic way, i.e., without exploiting the architectural decomposition of the system. Similarly, fault-tree analysis can be applied on the system to analyze the dependency of system failures on basic faults introduced in the error model. In fact, COMPASS uses the error models specified in the MILS-AADL language. COMPASS provides a facility for automatic model extension, that is, integration of nominal models with error models. The ability to generate such models automatically is a way to enhance reuse (and thus productivity), and to avoid a repetitive and potentially error-prone activity of modelling the system with failures. This approach is based on the notion of fault injections that specify the effect of the faults on the nominal behavior, the automatic integration of nominal (fault-free) system models with (user-defined) fault models, and the use of formal verification tools to analyze the resulting extended model.
Figure 15: Tools architecture.

4 Tool-set

Figure 15 shows the hierarchy of tools that are used in the verification of MILS-AADL models. Below, we give details of the integration and role of the single tools, and some guidelines on how to run the verification described in Section 3.

4.1 COMPASS Front-End

COMPASS is the front end. The user interacts with the GUI or with command-line scripts to check the syntax of the models, to run the verification, and to see the results. Automatic translations convert the user-level language MILS-AADL in the input language of the back-end tools, i.e., into the languages SMV, OCRA, and BIP. COMPASS includes the translators and the code to interact with the back-end tools.

The COMPASS GUI presents to the user different tabs, one for each main functionality of the tool. In particular, the tab “Model” allows the user to load the model and check its syntax. Syntactic errors are reported in a log frame at the bottom of the “Model” tab.

4.2 Verification with OCRA

4.2.1 Integration

OCRA enables the contract-based verification described in Section 3.5. It is called by COMPASS as a process. It receives in input the description of the architecture and contracts in the OCRA language, the behavioral models of the leaf components in the SMV language, and a script with the directives to run a specific verification tasks. nuXmv, xSAP, and HyCOMP are integrated as libraries in OCRA to implement the layered approach described in Section 3.5.3. In particular, nuXmv is used for most reasoning tasks calling specific algorithms for temporal satisfiability and model checking; xSAP is used to generate fault trees; HyCOMP is used for some subroutines on real-time and hybrid features.
4.2.2 Annotations

The properties for OCRA are specified directly in the system model through the annotation mechanism of MILS-AADL.

OCRA annotations formalizing the properties into OCRA contracts must be specified on the component types. The following is an example of OCRA annotation in a MILS-AADL program:

```miladl
system Simple
  features
    input: in event port;
    output: out event port;
  {OCRA: CONTRACT simple_contract
    assume: in the future {input};
    guarantee: always ({input} implies in the future {output});
  }
end Simple;
```

The annotation is the expression between curly brackets and starts with “OCRA:” to indicate that it is an OCRA annotation. The keyword “CONTRACT” is followed by the contract name. The keywords “assume:” and “guarantee:” are followed by temporal expressions, called respectively assumption and guarantee. These are composed using the temporal operators supported by OCRA. For example, we may use the operators “G”, “F”, “->” or the verbose counterpart “always”, “in the future”, “implies”.

The atomic expression inside assumptions and guarantees are enclosed between curly brackets. They are built by applying the operators described in Section 4.4 of D2.1 on constants and the ports of component type. Moreover, we allow the following additional operators:

- “next(.)”: the argument must be a data port; it has the same type of the argument (e.g., “next(value) = enum:OK”)
- “change(.)”: the argument must be a data port; it is a boolean predicate (e.g., “change(value) or value = enum:OK”)
- “rise(.)”, “fall(.)”: the argument must be a boolean data port; it is a boolean predicate (e.g., “rise(alarm)”)
- “time_until(.)”: the argument must be an event port; its type is real (e.g., “time_until(signal) <= 3”)

The complete grammar is listed below (see Section 3.5.1 for the definition of the semantics):

```plaintext
OCRA_annotation = "{OCRA: CONTRACT" name
                     "assume:" property ";";
                     "guarante:" property ";";
property = 
            "(" atom ")" | 
            "not" constraint | 
            constraint "and" constraint | 
constraint "or" constraint | 
```

Page 44   Version 1.0   1 September 2015
Confidentiality: Public Distribution
In order to apply compositional verification of these contracts, the model has to contain the contracts of all components types and the specification of a contract is refined by the contracts of the subcomponents. This is specified with another annotation in the component implementation. This appears as follows:

```plaintext
system implementation Simple.Impl
    subcomponents
        sub1: system SimpleSub.Impl;
        sub2: system SimpleSub.Impl;
        {OCRA: CONTRACT simple_contract REFINEDBY sub1.simple_subcontract, sub2.simple_subcontract
    }
end Simple.Impl;
```
4.2.3 GUI and Guidelines

The verification of OCRA contracts is enabled in the “OCRA” tab of the COMPASS GUI. The “OCRA” tab presents to the user three main verification tasks:

- “Validation”, which is used to check the consistency of all contracts in the model;
- “Correctness”, which is used to check the refinement and implementation of contracts, as described in Section 3.5.2;
- “Safety”, which is used to analyze the contract refinement in case of component failures generating fault trees as described in Section 3.5.6.

On the left side of the tab, the system-level contracts are listed. They can be selected to run the correctness verification or safety analysis tasks on a specific contract or on all contracts (while validation is applied always on all contracts).

For every check, the user can choose to use the Bounded Model Checker (BMC) algorithm, which is tailored to find traces up to a certain bound, or the K-LIVENESS algorithm described in Section 3.5.4. BMC cannot prove a property and is typically used only for debugging (to quickly find counterexamples). In the case implementations are verified, the GUI allows the user to select also the K-ZENO algorithm described in Section 3.5.5. All algorithms have also a bound that can be selected to stop the verification after having reached a certain limit. The user is not required to know the meaning of the bound but if the bound is reached the verification is not complete, as reported in the results.

After every check, a result is shown on the GUI. In case of consistency, the GUI reports only if the contracts are consistent or not. In case of correctness, the GUI reports if the property passes or fails. In case of failure, the report shows which parts are failing (this can be a refinement step or an implementation of a leaf component or combinations thereof) and counterexample traces that can be inspected to understand what is wrong. In case of safety, a fault tree is shown and the GUI provides the possibility to summarize the minimal cut sets in the fault tree.

4.3 Verification with secureBIP

secureBIP implements information flow security checks for component models annotated with transitive non-interference policies. As explained earlier in Section 3.2, the underlying information flow policy for MILS-AADL is based on intransitive non-interference and guaranteed by construction on the model. Therefore, the analysis implemented within secureBIP are intrinsically different, as they rely on a different security model, use different annotations and provide different security guarantees.

4.3.1 Integration

The integration of secureBIP tools within the D-MILS toolset is realized at modeling language level. Models in MILS-AADL are automatically translated into BIP, using the translator tool developed as part of WP3 activities.
In order to be further processed by secBIP, the generated BIP model need to be annotated i.e. by providing explicit security labels for particular model elements such as data variables, ports and interactions. In the current version, secureBIP handles security labels defined according to the Decentralized Label Model (DLM) [36], a highly-expressive, flexible annotation language for confidentiality and integrity annotations.

Let us notice however that, as the translation from MILS-AADL to BIP is structural, the annotations could have been easily provided at higher level in the MILS-AADL specification and then translated into the generated BIP model.

### 4.3.2 Guidelines

The secureBIP tool can be freely downloaded at [http://www-verimag.imag.fr/~bensaid/secureBIP/securebip_tool.html](http://www-verimag.imag.fr/~bensaid/secureBIP/securebip_tool.html).

As presented earlier in Section 3.3, the tool has mainly two functionalities, namely (1) checking a fully annotated model for event/data non-interference and (2) synthesizing a fully annotated non-interferent model, by extending a partial annotation, whenever such a model exists (or explain why it cannot be constructed).

The tool is invoked trough command line interface, for example, the synthesis runs as follows:

```
$ secbip -a config.xml -p model.bip -d comp-name
```

where `config.xml` is a configuration file, containing information about the underlying DLM used for annotation, as well as the partial annotation of model elements, `model.bip` is the BIP model and `comp-name` is the top-level (system) component under analysis. Further details and examples are available online.

### 4.4 Verification with RT-DFinder

RT-DFinder allows the user to compute system invariants using compositional methods, and to further use them in order to prove system properties as described in Section 3.6. It takes as input a BIP (or Real-time BIP) model and optional, a state property. If no property is specified, the property consider is deadlock-freedom. In a first step, several invariants are generated at component- and system-level, by using different generation methods and specific abstractions. Then, in a second step, a check is performed to validate if the conjunction of all these generated invariants imply the property. If yes, the property is proven. Otherwise, a counterexample system state is further generated to illustrate a concrete situation where all invariants actually hold, but not the property. The Yices solver is used as a backend for the validation and counter-example generation.

#### 4.4.1 Integration

The integration of RT-DFinder within the D-MILS toolset has been realized at modeling language level. That is, models in MILS-AADL are translated automatically into BIP using the milsaadl2bip translator developed as part of WP3 tasks. Let us notice that no specific annotations are used, the
obtained BIP model is fully operational and can be used directly as input to RT-DFinder. In contrast, the properties under verification (if different from deadlock freedom) need to be explicitly expressed in the assertion language of the underlying backend solver.

4.4.2 Guidelines

RT-DFinder can be freely downloaded at [http://www-verimag.imag.fr/RTD-Finder](http://www-verimag.imag.fr/RTD-Finder). Its invocation through command line is as follows:

```
$ rtdfinder -f model.bip -r comp-name [-p property.ys] [-out output.ys]
```

where `model.bip` is the BIP model file, `comp-name` is the name of the top-level (system) component, `property.ys` is the property and `output.ys` is the output file. The output file contains all the generated invariants, as well as, the verification conditions, depending on the property, to be further checked by the backend solver. Further details and examples are available online.

4.5 Verification with COMPASS

4.5.1 Integration

COMPASS allows the user to verify properties in a monolithic way without exploiting the architectural structure of the system. The functionalities of COMPASS are enabled to work on MILS-AADL models by the translation to SMV, which has been extended to consider the extension of the language developed during the project. COMPASS performs the verification tasks calling NuSMV4COMPASS, which integrates the libraries of nuXmv, xSAP, and HyCOMP in single executable binary. As for the verification with OCRA, nuXmv is used for most reasoning tasks calling specific algorithms for temporal satisfiability and model checking; xSAP is used to generate fault trees; HyCOMP is used for some subroutine on real-time and hybrid features. The difference with the verification performed with OCRA is that these functionalities are called on the monolithic system represented in the SMV file.

4.5.2 GUI and Guidelines

The verification requires the specification of properties with the pattern-based supported present in the “Property” tab. Fault injections to consider the error model must be specified in a dedicated frame of the “Model” tab. The other relevant tabs are “Validation” to validate the specification of properties, “Correctness” to verify the properties on the model, and “Safety” to perform safety analysis taking into account the model extended by the fault injection. For a more detailed documentation of these functionalities we refer the reader to the user guide of COMPASS, since these functionalities are not specific for D-MILS.

---

5Flattening of components hierarchy is however mandatory for hierarchical models.
5 Assurance Considerations

The methods described in this document provide evidence for the arguments in the assurance case. In particular, the verification tools provide a report on the results of the analysis. In order to use these results in the assurance case, it is necessary to clarify some assumptions:

- We assume that the MILS-AADL model represents the actual system architecture that the designer has in mind. This can be somehow supported by the inspection of the model with COMPASS simulator and by the inspection of the resulting configuration generated by the MPCC.
- We assume that the requirements are faithfully represented by the formal properties and contracts. This can be somehow supported by the validation functions of the tool-set.
- We assume that the translations from the user-level language to the input language of the back-end tools are correct. The translations have been implemented during the project as proof-of-concept. Using them in a real certification process requires to increase their readiness level.
- We assume that the results reported by the back-end tools are correct. This is somehow supported by the readiness level of the back-end tools, the maturity of the development process, and the user base. In fact, these tools have been used in different contexts and by many industrial partners. Some of the tools such as nuXmv, OCRA, xSAP and HyCOMP are publicly available and receive continuous feedback from the users.
Glossary

**local policies**
properties of a component; in the standard MILS literature, the term is primarily used to refer to security properties; in D-MILS, the term is used in a broader sense to encompass any kind of properties (including functional, real-time, safety and security).

**MILS platform**
the abstract machine that results from the composition of a set of MILS foundational components consisting at least of a separation kernel (processor(s) and software), that exports a set of resources and provides mechanisms to support the controlled flow of information among those resources.
Acronyms

**LTL**  Linear-time Temporal Logic  \[20\]

**MPCC**  MILS platform configuration compiler  \[3\]
References


