Project Number 318772

D4.4 Compositional verification techniques and tools for distributed MILS – Part 1

Version 1.0
31 July 2014
Final

Public Distribution

Fondazione Bruno Kessler, Université Joseph Fourier, fortiss

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

Project Partner Contact Information

Fondazione Bruno Kessler Alessandro Ciamatti Via Sommarive 18 38123 Trento, Italy Tel: +39 0461 314320 Fax: +39 0461 314591 E-mail: cimatti@fbk.eu	**fortiss** Harald Ruess Guerickestrasse 25 80805 Munich, Germany Tel: +49 89 36035 22 0 Fax: +49 89 36035 22 50 E-mail: ruess@fortiss.org
Frequentis Wolfgang Kampichler Innovationsstrasse 1 1100 Vienna, Austria Tel: +43 664 60 850 2775 Fax: +43 1 811 50 77 2775 E-mail: wolfgang.kampichler@frequentis.com	**LynuxWorks** Yuri Bakalov Rue Pierre Curie 38 78210 Saint-Cyr-l'Ecole, France Tel: +33 1 30 85 06 00 Fax: +33 1 30 85 06 06 E-mail: ybakalov@lnxw.com
RWTH Aachen University Joost-Pieter Katoen Ahornstrasse 55 D-52074 Aachen, Germany Tel: +49 241 8021200 Fax: +49 241 8022217 E-mail: katoen@cs.rwth-aachen.de	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 894 5845 E-mail: s.hansen@opengroup.org
TTTech Wilfried Steiner Schonbrunner Strasse 7 1040 Vienna, Austria Tel: +43 1 5853434 983 Fax: +43 1 585 65 38 5090 E-mail: wilfried.steiner@tttech.com	**Université Joseph Fourier** Saddek Bensalem Avenue de Vignate 2 38610 Gieres, France Tel: +33 4 56 52 03 71 Fax: +33 4 56 03 44 E-mail: saddek.bensalem@imag.fr
University of York Tim Kelly Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325477 Fax: +44 7976 889 545 E-mail: tim.kelly@cs.york.ac.uk	

Contents

Document Control

Version	Status	Date
0.1	Document started	16 May 2014
1.0	Final version	31 July 2014

Executive Summary

This document describes the principles behind the different tools used to analyse MILS-AADL models in the frame of the D-MILS project. The tools are to be called by a common interface in charge of proposing the right backend with respect to the analysis to perform. The toolset covers various types of compositional verification, including functional, real-time, security and safety properties. Methods and tools proposed here provide evidences for assurance cases that aim to assess the safety or security level of the system.

1 Introduction

This document describes the verification framework developed in the D-MILS project. Designing a D-MILS system implies, in particular, to develop a MILS-AADL model of the system. This model describes the system as a composition of different categories of components. The present document focuses on the verification of the components describing the software part of the system. The goal of the verification is to provide evidences that the system meet a given property. These evidences are to be used in a compositional assurance case [18] to assert that the system is as safe and secure as required.

In term of activities associated to the design and the deployment of a D-MILS system, the verification takes place at the end of the development of the model. The verification shall take place before calling the configuration compiler in order to configure the platform, and then deploy the system. Note that the verification addressed in this document verifies the model of the system. To complete the assurance case, one needs to check that the actual deployed components implement their counterpart in the model. This check is out of the scope of the D-MILS project.

The verification framework is made available to the system designer through a unified interface, based on the one developed for the COMPASS project [15]. This interface enables loading MILS-AADL files describing the system in order to perform the different analyses provided by the framework. Depending on the property to check, several analyses are proposed. Whenever one of them is chosen, the relevant MILS-AADL code is transformed as described in [17] to a language suitable for the backend performing the analysis. The present document describes the analyses provided by these different backends.

In Section 2, we present some verification problems from the Smart Grid case study [16]. The Section 3 presents how the BIP toolset, OCRA and NuSMV can be used to perform compositional analysis for security properties. Verifying real-time properties by using OCRA is presented in Section 4 Security property are addressed in Section 5. Finally, Section 6 presents how compositional techniques allow reusing the previous analyses to establish safety properties.

2 Challenge Problems from Cases Studies

In this Section, we present some verification problems that arise from the requirements for the smart micro grid case study [16]. The verification problems are expressed on the prosumer presented in [17, Sect. 3]. The Figure 1 depicts how the prosumer is decomposed into subjects.

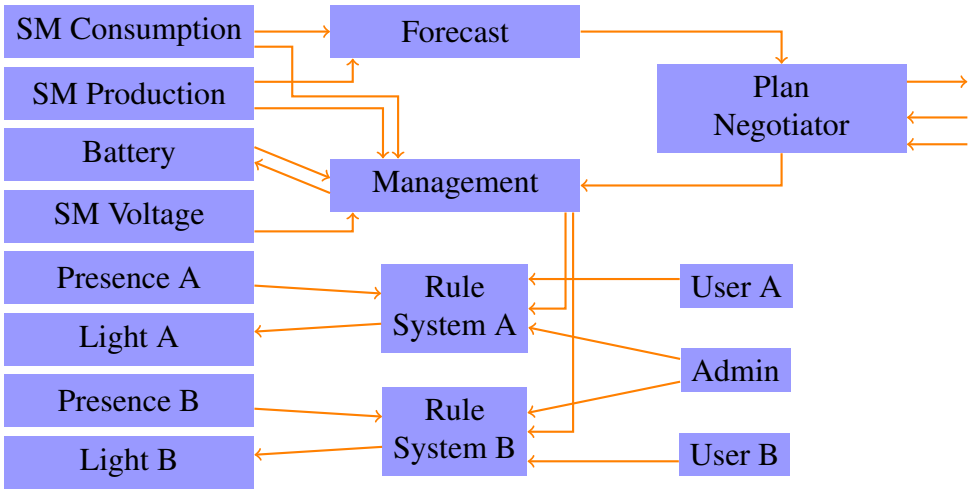


Figure 1: Decomposition of the prosumer into subjects

Functional Properties. We now focus on the following requirements from [16]:

SMG_SA.1 : The highest-level safety priority shall be grid stability. Indicators of instability are: deviation from the frequency 50 Hz and deviation from the nominal voltage level. In the case that the frequency deviates more than 1 Hz, the micro grid shall switch to island mode.

SMG_SA.2 : In case of a power outage, the micro grid shall switch to an island mode.

The prosumer model presented in [17] correspond to a nominal behavior, in which power outage and grid instability do not occur. The latter can be modeled as errors, who affect the voltage and/or the frequency meter. In our case, we consider only the voltage meter. The nominal voltage of 400V is modified whenever one of the error events of the following error model occur:

```
error model VoltageError
end VoltageError;

error model implementation VoltageError.impl
  states
    stable: initial state;
    unstable: error state;
    down: error state;
  events
```

```
    variation: error event;
    blackout: error event;
  transitions
    stable -[ variation] -> unstable;
    stable -[ blackout] -> down;
    unstable -[ blackout] -> down;
    unstable -[reset]-> stable;
    down -[reset] -> stable ;
end VoltageError.impl;
```

Whenever the `variation` error event occurs, the voltage detected by the smart meter is set to 359V. Similarly, upon the `blackout` event, the voltage detected by the smart meter is set to 0V.

In response, the `management` component is in charge of detecting such events (through the voltage value sent by the smart meter) and triggering the island mode. The latter is done by emitting an `island_mode` event, triggered by the following transition of the `management` component implementation:

```
polling -[island_mode when not island
                          and (voltage < 360 or voltage > 440)
                      then island:=true] -> polling ;
```

In order to check that the two requirements above are met, one need to check that whenever the `blackout` or `variation` event occurs, the `island_mode` event is eventually triggered.

Real-Time Properties. The above requirements are completed by a timing specification:

SMG_SA.3 : Switching to island mode shall to be accomplished in less than 20 ms.

This property involves:

- the reaction delay of the `Voltage SM` component,
- the transmission delay between the `Voltage SM` and `management` components, and
- the reaction time of the `management` component.

At the modeling level considered here, the timing constraints are expressed in hours to expressed the consumption and production plans of the prosumer. Therefore, all delays considered above should be instantaneous at this modelling level.

In order to check that this requirement is met, at this level, one should ensure that the `island_mode` event is triggered at the same *model time* as the error causing the prosumer to switch to island mode.

Then a more refined model has to be built, including the aforementioned delays. These delays can either be ensured by the hardware (i.e. latency for the communication between component is ensured by the separation kernel or the time triggered network) or by the components. In the latter case, reaction delays of each component may depend on the resources allocated to it. In particular, assumptions on these resources have to be given to the configuration compiler.

Security Properties We focus on the following requirement:

SMG_SO.1 : Communication / Information Flow between components shall be according to the policy defined in [16, Figure 9] (Prosumer Information Flow) [and [16, Figure 10] (Micro Grid Information Flow)]. No other information flow shall occur.

This requirement can be checked by grouping subjects from Figure 1 so that they match the domains defined in [16, Figure 9]. This correspond to the information flow refinement from Van der Meyden [29].

The last requirement that we consider is defined over the global system comprising the prosumers and the smart grid coordinating them. Such a system is depicted in Figure 2. As a lot of information can be obtained through the consumption of a particular household, the consumption value has to be protected. In our system, a given prosumer should not be able to see or deduce the consumption of another prosumer. However, as the grid gives some information about the total consumption in the

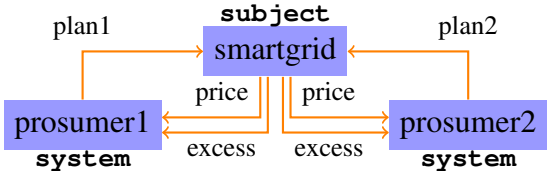


Figure 2: Global view of the smart grid.

grid, each prosumer might get some information about consumption of other prosumers. We would like to ensure that the value returned by the smart grid does not convey more information than what's needed for the grid to function properly. This requires specifying which amount of information is allowed from the grid to the prosumers and ensure that no more information is transmitted.

3 Invariant-based Verification

3.1 Compositional verification for deadlock freedom of timed systems

We recall hereafter the method for compositional generation of invariants for timed systems introduced in [3].

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 [9, 7, 8], summarised in Figure 3. 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 $II(\gamma)$. 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.

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

Figure 3: Compositional Verification Rule

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 "$B \models \Box\ \Psi$" is to be read as "Ψ holds in every reachable state of B".

The verification rule (VR) has been developed in [9, 8] 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 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.

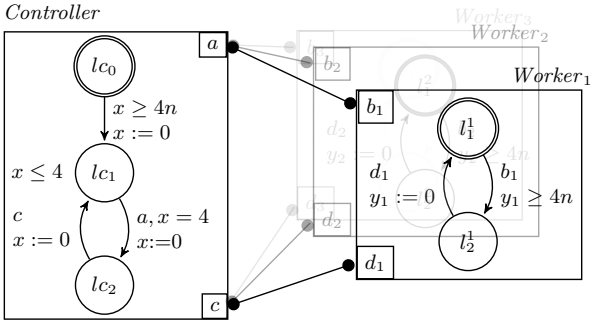


Figure 4: A Timed System Example

We concretize the component and interaction invariants for the example shown in Figure 4. 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 $\{(a \mid b_i), (c \mid d_i) \mid i \leq n\}$. 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 $y_i - x \geq 4n - 4$. Unfortunately, this property cannot be shown if we use (VR) as it is in [9].

To give a logical characterisation of components and interactions, we use invariants. An invariant Φ is a state property which holds in every reachable state of a component (or of a system) B, in symbols, $< B, s_0 > \models \Box\Phi$. We use $CI(< B_i, s_0^i >)$ 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 $(l \wedge \zeta)$ and where, 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 4 with one worker are:

$$CI(< Controller, (lc_0, x = 0) >) = (lc_0 \wedge x \geq 0) \vee (lc_1 \wedge x \leq 4) \vee (lc_2 \wedge x \geq 0)$$
$$CI(< Worker_i, (l_1^i, y = 0) >) = (l_1^i \wedge y_i \geq 0) \vee (l_2^i \wedge y_i \geq 4).$$

The interaction invariants we use are computed by the method explained in [9]. 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 $worker_1$ is intially at location l_1^1. In addition, the system clocks are supposed to be initally equal, thus the intial global state of the system is: $s_0 = ((lc_0, l_1^1), x = y = 0)$. The interaction invariant $II\big(\{(a \mid b_1), (c \mid d_1)\}, s_0\big)$ is; $(l_1^1 \vee l_2^1) \wedge (lc_2 \vee l_1^1) \wedge (l_2^1 \vee lc_0 \vee lc_1) \wedge (lc_2 \vee lc_0 \vee lc_1)$.

3.1.1 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 $\Psi_{Safe} = (lc_1 \wedge l_1^1 \rightarrow x \leq y_1)$ 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 lc_1 and the worker is at location l_1^1. It is not difficult to see that Ψ_{Safe} cannot be deduced from $CI(< Controller, s_0^c >) \wedge CI(< Worker_1, s_0^1 >) \wedge II\big(\{(a \mid b_1), (c \mid d_1)\}, s_0\big)$.

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 projection of history clocks, an invariant of the initial system.

We reconsider the sub-system from Figure 4. We illustrate how the safety property ψ_{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_{init} measures the time elapsed since the start. The invariants for the components with history clocks are:

$$\begin{aligned} CI(< Controller^h, s_0^{ch} >) = (&lc_0 \wedge x = h_{init} < h_a \wedge h_c > h_{init}) \vee \\ &(lc_1 \wedge x \leq h_{init} - 4 \wedge x \leq 4 \wedge h_a > h_{init} \wedge h_c > h_{init}) \vee \\ &(lc_1 \wedge x \leq 4 \wedge x = h_c \leq h_a \leq h_{init} - 8) \vee \\ &(lc_2 \wedge x \leq h_{init} - 8 \wedge h_a = x \wedge h_c > h_{init})) \\ &(lc_2 \wedge x = h_a \wedge h_c = h_a + 4 \leq h_{init} - 8) \end{aligned}$$

Where, s_0^{ch} and $s_0^{1^h}$ denote initial symbolic states of $Controller^h$ and $Worker_1^h$ respectively, $s_0^{ch} = (lc_0, x = h_{init} = 0 \wedge h_a > h_{init} \wedge h_c > h_{init})$ and $s_0^{1^h} = (l_1^1, y_1 = h_{init} = 0 \wedge h_{b1} > h_{init} \wedge h_{d1} > h_{init})$

$$\begin{aligned} CI(< Worker_1^h, s_0^{1^h} >) = (&l_1^1 \wedge y_1 = h_{init} < h_{d_1} \wedge h_{b_1} > h_{init}) \vee \\ &(l_1^1 \wedge y_1 = h_{d_1} \leq h_{b_1} \leq h_{init} - 4) \vee \\ &(l_2^1 \wedge y_1 = h_{init} \geq h_{b_1} + 4 \wedge h_{d_1} > h_{init})) \vee \\ &(l_2^1 \wedge y_1 = h_{d_1} \leq h_{init} - 4 \wedge h_{b_1} \leq h_{d_1} - 4) \end{aligned}$$

By using the interaction invariant and the equality constraints $\mathcal{E}(\gamma)$, after the elimination of the existential quantifiers in $\big(\exists h_a.\exists h_{b_1}.\exists h_c.\exists h_{d_1}\big) CI(< Controller^h, s_0^{ch} >) \wedge CI(< Worker_1^h, s_0^{1^h} >) \wedge II(\gamma, s_0) \wedge \mathcal{E}(\gamma)\big)$, we obtain the following invariant Φ :

$$\begin{aligned} \Phi = (&l_1^1 \wedge lc_0 \wedge \boldsymbol{x = y_1}) \vee \big(l_1^1 \wedge lc_1 \wedge (\boldsymbol{y_1 = x \vee y_1 \geq x + 4})\big) \vee \\ &\big(l_2^1 \wedge lc_2 \wedge \boldsymbol{(y_1 = x + 4 \vee y_1 \geq x + 8)}\big). \end{aligned}$$

Here, s_0^{ch} and $s_0^{1^h}$ denote initial symbolic states of $Controller^h$ and $Worker_1^h$ respectively. It can be easily checked that $\Phi \wedge \neg\Psi_{Safe}$ has no satisfying model and this proves that Ψ_{Safe} holds for the system. We used bold fonts in Φ to highlight relations between x and y_1 which are not in $CI(< Controller, s_0^c >) \wedge CI(< Worker_1, s_0^1 >)) \wedge II(\gamma, s_0)$.

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:

$$\mathcal{S}(\gamma) \equiv \bigwedge_{i \neq j} |h_{c|d_i} - h_{c|d_j}| \geq 4 \ \wedge \bigwedge_{i \neq j} |h_{a|b_i} - h_{a|b_j}| \geq 4.$$

The invariant $\mathcal{S}(\gamma)$ is defined over the history clocks for interactions. Previously, the invariant $\mathcal{E}(\gamma)$ 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 $\mathcal{E}^*$. For our running example, $\mathcal{E}^*$ is:

$$\mathcal{E}^*(\gamma) \equiv \ h_{b_1} = h_{a|b_1} \wedge h_{b_2} = h_{a|b_2} \wedge \ h_a = \min_{i=1,2}(h_{a|b_i}) \wedge h_{d_1} = h_{c|d_1} \wedge h_{d_2} = h_{c|d_2} \wedge \ h_c = \min_{i=1,2}(h_{c|d_i})$$

The predicate $\exists \mathcal{H}_A \exists \mathcal{H}_\gamma . (\bigwedge_i CI(< \ B_i^h, s_0^{i^h} \ >) \wedge II(\gamma, s_0) \wedge \mathcal{E}^*(\gamma) \wedge \mathcal{S}(\gamma))$ is an invariant of $< \|_\gamma B_i, s_0 >$. This new invariant is in general stronger than $\exists \mathcal{H}_A . (\bigwedge_i CI(< B_i^h, s_0^{i^h} >) \wedge II(\gamma, s_0) \wedge \mathcal{E}(\gamma))$ 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 $\mathcal{E}^*$ and $\mathcal{S}$. The interaction invariant is:

$$\begin{aligned} II(\gamma) \ &= \ (l_1^1 \vee l_2^1) \wedge (l_1^2 \vee l_2^2) \wedge (lc_2 \vee l_1^1 \vee l_1^2) \wedge (l_1^1 \vee lc_1 \vee lc_2) \wedge (l_1^2 \vee lc_1 \vee lc_2) \\ &\wedge \ (lc_2 \vee l_1^1 \vee l_1^2) \wedge (lc_0 \vee lc_1 \vee lc_2) \wedge (lc_0 \vee lc_1 \vee l_2^1 \vee l_2^2) \end{aligned}$$

The components invariants are:

$$\begin{aligned} CI(< Controller^h, s_0^{ch} >) = \ & (lc_0 \wedge x = h_{init} < h_a \wedge h_c > h_{init}) \vee \\ & (lc_1 \wedge x \leq h_{init} - 8 \wedge x \leq 4 \wedge h_a > h_{init} \wedge h_c > h_{init}) \vee \\ & (lc_1 \wedge x \leq 4 \wedge x = h_c \leq h_a \leq h_{init} - 12) \vee \\ & (lc_2 \wedge x \leq h_{init} - 12 \wedge h_a = x \wedge h_c > h_{init})) \\ & (lc_2 \wedge x = h_a \wedge h_c = h_a + 4 \leq h_{init} - 12) \end{aligned}$$

Where, s_0^{ch} and $s_0^{i\,h}$ denote initial symbolic states of $Controller^h$ and $Worker_i^h$ respectively.
$s_0^{ch} = (lc_0, x = h_{init} = 0 \wedge h_a \geq 0 \wedge h_c \geq 0)$ and $s_0^{i\,h} = (l_i^1, y_i = h_{init} = 0 \wedge h_{bi} \geq 0 \wedge h_{di} \geq 0)$

$$CI(< Worker_i^h, s_0^{i\,h} >) = (l_1^i \wedge y_i = h_{init} < h_{d_i} \wedge h_{b_i} > h_{init}) \vee$$

$$(l_1^i \wedge y_i = h_{d_i} \leq h_{b_i} \leq h_{init} - 8) \vee$$
$$(l_2^i \wedge y_i = h_{init} \geq h_{b_i} + 8 \wedge h_{d_i} > h_{init})) \vee$$
$$(l_2^i \wedge y_i = h_{d_i} \leq h_{init} - 8 \wedge h_{b_i} \leq h_{d_i} - 8)$$

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

$$\begin{aligned}
\Phi = &\big(l_1^1 \wedge l_1^2 \wedge lc_0 \wedge x = y_1 = y_2\big) \vee \\
&\big(l_1^1 \wedge l_1^2 \wedge lc_1 \wedge x \leq 4 \wedge (y_1 = x \wedge \boldsymbol{y_2 - y_1 \geq 4} \vee y_1 \geq x + 8 \vee \\
&\qquad\qquad y_2 = x \wedge \boldsymbol{y_1 - y_2 \geq 4} \vee y_2 \geq x + 8)\big) \vee \\
&\big(l_2^1 \wedge l_1^2 \wedge lc_2 \wedge (y_1 \geq x + 8 \vee (y_2 = x + 4 \wedge \boldsymbol{y_1 - y_2 \geq 4}))\big) \vee \\
&\big(l_1^2 \wedge l_2^1 \wedge lc_2 \wedge (y_2 \geq x + 8 \vee (y_1 = x + 4 \wedge \boldsymbol{y_2 - y_1 \geq 4}))\big)
\end{aligned}$$

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

3.1.2 Implementation and Experiments

The method has been implemented in a Scala[1] prototype which is freely available at `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 γ and a global safety property Ψ and checks whether the system satisfies Ψ. It generates Z3[2] Python code to check the satisfiability of the formula $\wedge_i CI(B_i) \wedge II(\gamma) \wedge \Phi^* \wedge \neg\Psi$ where Φ^*, depending on whether γ is conflicting, stands for $\mathcal{E}(\gamma)$ or $\mathcal{E}^*(\gamma) \wedge \mathcal{S}(\gamma)$. If the formula is not satisfiable, the prototype returns `no solution`, that is, the system is guaranteed to satisfy Ψ. Otherwise, it returns a substitution for which the formula is satisfiable, that is, the conjunction of invariants is true while Ψ 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 [3].

In table 3.1.2, 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.

[1] `http://www.scala-lang.org/`
[2] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

Model & Property	Size (N)	Time (in seconds)	
		Our Prototype	Uppaal
Fischer & mutual exclusion	1	0.156	0
	4	0.22	0.012
	6	0.36	0.03
	14	2.840	no result in 4 hours
Train Gate Controller & mutual exclusion	2	0.176	0
	8	0.356	0.01
	16	0.7	0.01
	32	1.484	0.03
	64	4.416	0.2
	124	15.765	1.6
Temperature Controller & absence of deadlock	2	0.144	0.008
	8	0.5	0.01
	16	1.132	0.11
	32	1.308	8.83
	64	4.056	649.37
	124	19.22	no results in 6 hours

3.2 Compositional verification of temporal formulas

3.2.1 Functional Properties as Temporal Formulas

Since the seminal work of Pnueli in 1977 [23], 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 tick in the computation. Thus, if the behavior of a system M is represented by a set of computations L_M, the property ϕ is satisfied by the system (denoted with $M \models \phi$) iff each computation $\sigma \in L$ is a model of the formula ϕ (denoted with $\sigma \models \phi$).

More specifically, we use Linear-time Temporal Logic (LTL) [23] to specify properties on a system. The atomic formulas *Atoms* are predicates over the variables V of the system. Besides the Boolean connectives, LTL uses the temporal operators $\mathbf{X}$ ("next") and $\mathbf{U}$ ("until"). Formally,

- a predicate $a \in \mathit{Atoms}$ is an LTL formula,
- if ϕ_1 and ϕ_2 are LTL formulas, then $\neg\phi_1$, and $\phi_1 \wedge \phi_2$ are LTL formulas,
- if ϕ_1 and ϕ_2 are LTL formulas, then $\mathbf{X}\phi_1$ and $\phi_1 \mathbf{U} \phi_2$ are LTL formulas.

We use the standard abbreviations: $\top := p \vee \neg p$, $\bot := p \wedge \neg p$, $\mathbf{F}\phi := \top \mathbf{U} \phi$, $\mathbf{G}\phi := \neg\mathbf{F}\neg\phi$, and $\phi_1 \mathbf{R} \phi_2 := \neg(\neg\phi_1 \mathbf{U} \neg\phi_2)$.

Let a computation be a sequence of assignment over V. We denote with $\sigma[i]$ the $i+1$-the element of the sequence and with σ^i the suffix starting from $\sigma[i]$. Given an LTL formula ϕ and a computation σ, we define $\sigma \models \phi$, i.e., that the path σ satisfies the formula ϕ, as follows:

- $\sigma \models a$ iff $\sigma[0] \models a$
- $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$
- $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$
- $\sigma \models \mathbf{X}\phi$ iff $\sigma^1 \models \phi$
- $\sigma \models \phi \mathbf{U} \psi$ iff for some $j \geq 0$, $\sigma^j \models \psi$ and for all $0 \leq k < j$, $\sigma^k \models \phi$.

3.2.2 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 ϕ.

Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We denote formulas with φ, ψ, I, T, P, variables with x, y, v, and sets of variables with X, Y, and 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 $l_1 \wedge \ldots \wedge l_n$, with $\neg s$ we denote the clause $\neg l_1 \vee \ldots \vee \neg l_n$, 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 $C_1 \wedge \ldots \wedge C_n$ as sets of clauses $\{C_1, \ldots, C_n\}$, and vice versa. If $X_1, \ldots, X_n$ are a sets of variables and φ is a formula, we might write $\varphi(X_1, \ldots, X_n)$ to indicate that all the variables occurring in φ 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 φ, φ' is the formula obtained by adding a prime to each variable occurring in φ. Given a theory $\mathcal{T}$, we write $\varphi \models_{\mathcal{T}} \psi$ (or simply $\varphi \models \psi$) to denote that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$.

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 Σ_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 $\langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$.

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

The automata-based approach [30] to LTL model checking is to build a transition system $M_{\neg\phi}$ with a fairness condition $f_{\neg\phi}$ such that $M \models \phi$ iff $M \times M_{\neg\phi} \models \mathbf{FG}\neg f_{\neg\phi}$. This reduces to finding a counterexample as a fair path, i.e., a path of the system that visits the fairness condition $f_{\neg\phi}$ 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.

3.2.3 Property-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 ϕ_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 ϕ. The compositional verification rule has therefore the following general form:

$$\frac{\begin{array}{c}\text{for all } i, 1 \leq i \leq n, m_i \models \phi_i \\ compprop_\gamma(\phi_1, \ldots, \phi_n) \preceq \phi\end{array}}{compsys_\gamma(m_1, \ldots, m_n) \models \phi}$$

where $compsys_\gamma$ is the composition operator of the components $m_1, \ldots, m_n$ given the connections γ; $compprop_\gamma$ is the related composition operator for the properties; $\preceq$ is the refinement operator for the properties. In synchronous systems, for example, γ can be seen as a renaming, $compsys_\gamma$ is the synchronous composition, and $compprop_\gamma$ 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. This is the case for example for temporal properties such as LTL formulas. 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 a implementation satisfying C iff $I \models A \rightarrow G$. We say that 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 [13] to consider the refinement along a structural decomposition taking into account the contracts of the sub-components of a component. Still in [13], 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.

Therefore, the second condition of the compositional verification rule showed above can be reduced to satisfiability of LTL. The first condition consists of a set of model checking problems. Since the satisfiability of LTL can be reduced on turn to model checking, we focus in the following on an efficient algorithm for model checking of LTL (on infinite-state transition systems).

3.2.4 Model Checking with IC3 and K-LIVENESS

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 [10] 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 [12] 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) [21]. The approach leverages *Implicit Abstraction* (IA) [28], 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 [14] is an algorithm recently proposed to reduce liveness (and so also LTL verification) to a sequence of invariant checking. Differently from other reductions (such as [27]), 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 ($\mathbf{FG}\neg f$). 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 ϕ, and a positive integer K, for every finite path σ of M, let $\sigma \models_{fin} \sharp(\phi) \leq K$ iff the size of the set $\{i \mid \sigma[i] \models \phi\}$ is less or equal to K. In [14], it is proved that, for finite-state systems, $M \models \mathbf{FG}\neg f$ iff there exists K such that $M \models_{fin} \sharp(f) \leq K$. 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 [14] 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.

4 Compositional Verification of Real-Time Properties

4.1 Real-Time Properties as Temporal Formulas

When the MILS-AADL program contains clock or continuous data, then the its semantics can be mapped to hybrid automata. In this section, we focus on the compositional verification of temporal

properties in the real-time case, when we consider system computations with a dense model of time. As in section 3.2.1, we use LTL to specify temporal properties and we used a property-based compositional approach. In the following, we focused on how to solve the model checking problem in the case of Parametric Rectangular Hybrid Automata (PRHA), which include the continuous dynamics allowed by MILS-AADL.

4.2 K-LIVENESS for Hybrid Automata

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 $\mathbf{FG}\neg f$ because for any value of p, f is visited at most p times. However, K-LIVENESS will obtain a counterexample to the safety property $\sharp(f) \leq K$ 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 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 β based on the automaton structure to force a minimum distance between two fair time points. We use an additional transition system Z_β, with a condition f_Z, to reduce the problem of proving that $H \models \phi$ to proving that $M_H \times M_{\neg\phi} \times Z_\beta \models \mathbf{FG}\neg f_Z$.

4.3 Linking the fairness to time progress

In this section, we define the transition system Z_β 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 an PRHA H and of the negation of the property ϕ. 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 ϕ 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. $M_H \times M_{\neg\phi} \times Z_B \models \mathbf{FG}\neg f_Z$.

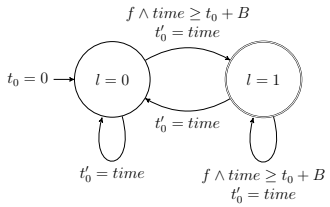


Figure 5: Monitor $Z_B(f, time)$.

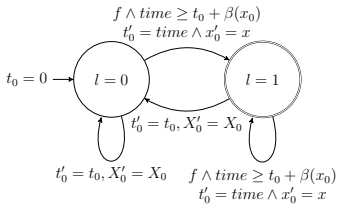


Figure 6: Monitor $Z_\beta(f, time, X)$.

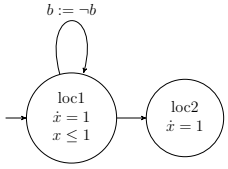


Figure 7: Example of TA.

Theorem 1 If $B > 0$, $H \models \phi$ iff $M_H \times M_{\neg\phi} \times Z_B \models \mathbf{FG}\neg f_Z$.

We generalize the construction of Z_B considering as bound on time a function β 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 β. 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 $M_H \times M_{\neg\phi} \times Z_\beta \models \mathbf{FG}\neg f_Z$.

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 β_H such that $H \models \phi$ iff there exists K such that $M \times M_{\neg\phi} \times Z_{\beta_H} \models_{fin} \sharp(f_Z) \leq K$. Thus, if $H \models \phi$, and the reduction is complete, and the subroutine SAFE terminates at every call, then K-LIVENESS also terminates proving the property.

4.4 The K-ZENO algorithm

The K-ZENO algorithm is a simple extension of K-LIVENESS which, given the problem $H \models \phi$, builds $M = M_H \times M_{\neg\phi} \times Z_\beta$ and calls K-LIVENESS with inputs M and f_Z. 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 β, because the completeness of the reduction depends on β. 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 β 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 $\mathbf{FG}pc = loc2$. Taking $\beta = 1$, K-ZENO proves the property with $K = 1$. In fact, starting from the location $loc1$, after 1 time unit, the automaton

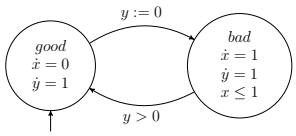


Figure 8: Stopwatch automaton

cannot reach $loc1$ anymore. For PTAs, we consider as β 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 $x \bowtie B$ where $\bowtie \in \{\leq, \geq, <, >\}$. 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 β we take the maximum among the β_g for all g in the automaton H for which r_g exists and the constant 1. Note that this coincides with the β 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.

4.5 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 2 If $H \models \phi$, then there exists K such that $M_H \times M_{\neg\phi} \times Z_\beta \models_{fin} \sharp(f_Z) \leq K$.

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 $(\mathbf{FG}good)$, because the invariant on x and the guard on y make sure that the total time spent in bad is at most 1 time unit. However, K-ZENO cannot prove it with any K because time can pass indefinitely in $good$, while x is stopped. Therefore, it is always possible to visit 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 $\mathbf{GF}good$.

4.6 Experimental Evaluation

4.6.1 Implementation

We have implemented the K-ZENO algorithm on top of the SMT extension of IC3 described in Section 3.2.1. 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 properties, and not to disprove them. If a property does not hold, our tool does not terminate. Similarly to the Boolean case [14], our implementation consists of relatively few (and simple) lines of code on top of IC3.

4.6.2 Benchmarks

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 nxn 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 rectangular 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 ($\mathbf{GF}p$), strong fairness ($\mathbf{GF}p \rightarrow \mathbf{GF}q$), and "leads to" ($\mathbf{G}(p \rightarrow \mathbf{F}q)$). 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 ($\mathbf{FG}$ *good*) in the *Counter* benchmarks.

4.6.3 Evaluation

Effectiveness. 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

Table 1: Selected experimental results.

Instance	Class	Property	# Bool vars	# Real vars	Trans size	k	Time
Fischer (8 processes)	T	$(\bigwedge_{i=1}^{17} \mathbf{GF}p_i) \rightarrow \mathbf{G}(\neg p_{18} \rightarrow \mathbf{F}p_{18})$	132	20	1286	3	6.37
Fischer Fair (2 processes)	P	$(p_1 \wedge \mathbf{GF}p_2) \rightarrow \mathbf{G}(p_3 \rightarrow \mathbf{F}p_4)$	38	12	622	4	76.14
Fischer Hybrid (10 procs)	R	$(\mathbf{GF}p_1 \wedge \mathbf{GF}p_2 \wedge \mathbf{FG}p_3) \rightarrow \mathbf{G}(p_4 \rightarrow \mathbf{F}p_5)$	106	64	8759	1	325.03
Dist Controller (3 sensors)	N	$(\mathbf{GF}p_1) \rightarrow (\mathbf{GF}p_2)$	58	27	1737	1	397.24
Nuclear Reactor (9 rods)	R	$\mathbf{G}(p_1 \rightarrow \mathbf{F}p_2)$	82	24	3258	1	530.40
NavigationInit (3x3)	N	$\mathbf{FG}(p_1 \vee p_2 \vee p_3 \vee p_4)$	16	8	808	2	4.37
NavigationInit (10x10)	N	$\mathbf{FG}(p_1 \vee p_2 \vee p_3 \vee p_4)$	22	8	4030	2	453.74
NavigationFree (3x3)	N	$\mathbf{FG}(p_1 \vee p_2 \vee p_3 \vee p_4)$	16	8	808	2	3.37
NavigationFree (9x9)	N	$\mathbf{FG}(p_1 \vee p_2 \vee p_3 \vee p_4)$	22	8	3461	2	872.07
Counter 10	T	$\mathbf{FG}p$	10	24	294	10	52.74
Diesel Gen (small)	T	$\mathbf{G}(p_1 \rightarrow \mathbf{F}(\neg p_2 \vee p_3))$	84	24	724	1	16.55
Diesel Gen (medium)	T	$\mathbf{G}(p_1 \rightarrow \mathbf{F}(\neg p_2 \vee p_3))$	140	30	1184	1	51.24
Diesel Gen (large)	T	$\mathbf{G}(p_1 \rightarrow \mathbf{F}(\neg p_2 \vee p_3 \vee p_4))$	264	62	2567	1	538.39

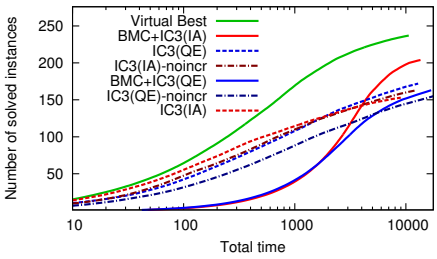
Classes: **T:** timed, **P:** parametric timed, **R:** rectangular, **N:** non-initialized rectangular.

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 p_i'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[3], 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.

Heuristics and Implementation Choices. 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 [12] 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.
- 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.

[3] 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 $f_{\neg\phi}$ before time diverges is usually low. The benchmarks of the *Counter* family were created on purpose, to show that k can increase arbitrarily.



Configuration	# solved	Tot time
Virtual Best	238	10603
BMC+IC3(IA)	205	13525
IC3(QE)	173	12895
IC3(IA)-NOINCR	164	12476
BMC+IC3(QE)	164	16888
IC3(QE)-NOINCR	156	17643
IC3(IA)	154	8493

Figure 9: Experimental comparison of various configuration options.

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 Fig. 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.

Fig. 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 implementation [14]. However, as the authors of [14], also in this case we do not have a clear explanation for this behavior.

Comparison with Other Tools. 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 [4]. 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

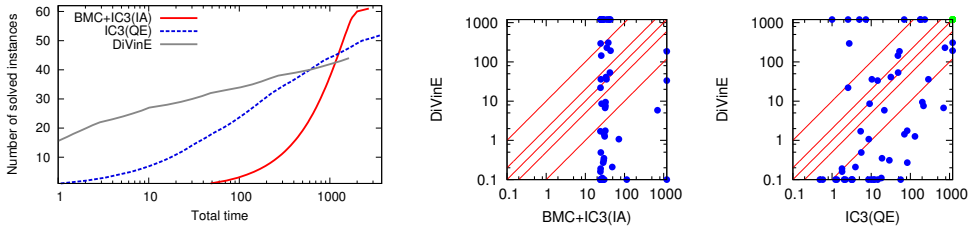


Figure 10: Comparison with DIVINE.

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 [22].

The results are shown in Fig. 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.

5 Compositional Verification for Security Properties

5.1 Information Flow Security for Component-Based Systems

In this section we summarize our recent work on information flow security for components based systems [6]. We have extended the BIP framework [5] with security features, leading to the *secureBIP* framework. *secureBIP* allows for building secure, complex and hierarchically structured systems from atomic components behavior and interactions, annotated with security information. For *secureBIP* we study two types of non-interference: *event non-interference*, tracking information flow about occurences 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 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 [6] and an extended version can be found as the technical report [26].

5.1.1 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 [17]. 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 $\sigma : X \cup P \cup \gamma \rightarrow S$ 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.

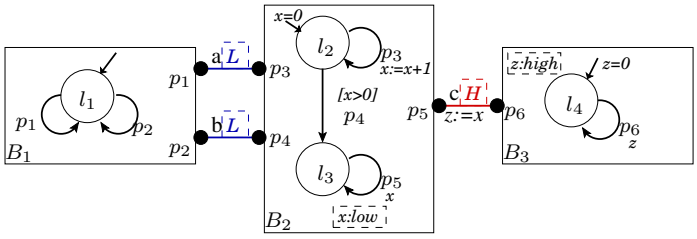


Figure 11: BIP Composite component annotated

Figure 11 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 σ be a security assignment for C, fixed. For a security level $s \in S$, we define $\gamma \downarrow_s^\sigma$ the restriction of γ to interactions with security level at most s that is formally, $\gamma \downarrow_s^\sigma = \{a \in \gamma \mid \sigma(a) \subseteq s\}$.

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

For a security level $s \in S$, we define the equivalence $\approx_s^\sigma$ on states of C. Two states q_1, q_2 are equivalent, denoted by $q_1 \approx_s^\sigma q_2$ 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 σ ensures **event non-interference** of $\gamma(B_1, \ldots, B_n)$ at security level s iff,

$$\forall q_0 \in Q_C^0 : \quad \text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma = \text{TRACES}((\gamma{\downarrow_s^\sigma})(B_1, \ldots, B_n), q_0)$$

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 [24] definition for transitive non-interference. But, let us remark that event non-interference is not concerned about the protection of data.

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

$$\begin{array}{l} \forall q_1, q_2 \in Q_C^0 : \; q_1 \approx_s^\sigma q_2 \Rightarrow \\ \quad \forall w_1 \in \text{TRACES}(C, q_1), w_2 \in \text{TRACES}(C, q_2) : \; w_1|_s^\sigma = w_2|_s^\sigma \Rightarrow \\ \quad\quad \forall q_1', q_2' \in Q_C : \; q_1 \xrightarrow[C]{w_1} q_1' \wedge q_2 \xrightarrow[C]{w_2} q_2' \Rightarrow q_1' \approx_s^\sigma q_2' \end{array}$$

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 σ 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 $s \in S$.

5.1.2 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 σ be a security assignment. We say that C satisfies the **security conditions** for security assignment σ 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 sequential 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,

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

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

(iv) all atomic components B_i are port deterministic:

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p} \ell_2, \tau_2 = \ell_1 \xrightarrow{p} \ell_3 \ \Rightarrow \ (g_{\tau_1} \wedge g_{\tau_2}) \textit{ is unsatisfiable}$$

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 [19, 20] 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 [6] that the above security conditions are sufficient to ensure both event and data non-interference. Formally, whenever the security conditions hold, the security assignment σ is secure for the composite component C.

As an illustration, consider the composite component in Figure 11. It can be relatively easily checked that the security conditions hold, henceforth, the composite component is secure.

5.1.3 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 [11]. To maintain energy consumption, "Temperature" and "Presence" Web services (WS) that encapsulate the sensors and the actuators embedded in the building, collect information 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 aadl-mils language in deliverable d2.2.

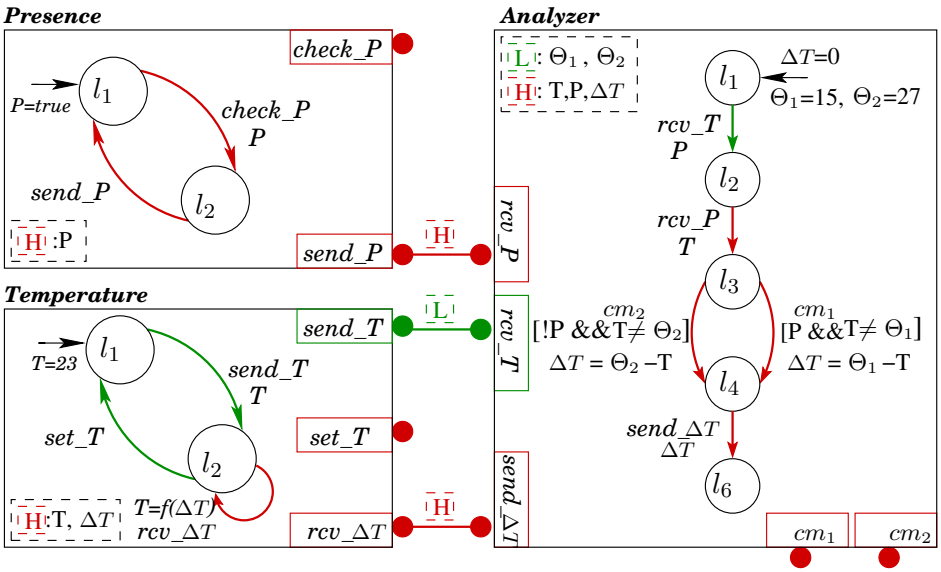


Figure 12: Home Gateway system

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 12, 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 ΔT from some Θ_1 and Θ_2 variables, temperatures threshold depending on P. Then, ΔT is send 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 ΔT value is sent to *Temperature* component to update the T value.

The system can be proven non-interferent iff 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 ΔT 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 ΔT then, it has also to be confidential and assigned with $high$ security level.

5.2 Declassification

In some cases, classified information has to be transmitted to a low security agent for the system to function properly. In the smartgrid example, the consumption of a particular prosumer is classified in the sense that other prosumers should not be able to access or deduce it. However, the grid informs all prosumers about the global consumption. This is necessary, as they need to adapt their plans in case of over or under consumption.

In such cases, the classified information is declassified [25] as it is transmitted to a lower security agent. A declassification operation should specify what information is allowed to be transmitted from the high security to the low security domain. In our case, only the sum of the consumption of all prosumers is needed and thus only this sum should be transmitted to the prosumers. A particular prosumer should not be able to distinguish between two situations where consumption of other prosumers differs but the sum is the same.

We are currently investigating cases where an expression involving classified variables is declassified, but not individual variables. The low security agent should not be able to distinguish two situations in which the declassified expression has a constant value, even if the values of individual variable change.

6 Compositional Verification for Safety Properties

In order to formalize safety requirements, we use the error models specified in the MILS-AADL language: after extending the nominal model with faulty behaviours the safety requirement will be formalized as a property of the extended model with one of the property types discussed in the previous sections (functional, real-time, security). The approach relies on the extension of COMPASS to support the MILS-AADL language. The COMPASS platform 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. The verification can be performed in a compositional way by applying the techniques described above on the extended model.

References

[1] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *Proceedings of EMSOFT*, 2010.

[2] Rafael Accorsi and Andreas Lehmann. Automatic information flow analysis of business process models. In *Proceedings of the 10th international conference on Business Process Management*, BPM'12, pages 172–187. Springer-Verlag, 2012.

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

[4] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, pages 863–868, 2013.

[5] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Software Engineering and Formal Methods SEFM'06 Proceedings*, pages 3–12. IEEE Computer Society Press, 2006.

[6] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven information flow security for component-based systems. In *From Programs to Systems: The Systems Perspective in Computing, Workshop at ETAPS'14*, volume 8415 of *LNCS*, 2014.

[7] S. Bensalem, M. Bozga, T-H. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 614–619, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, 2010.

[9] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 64–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] A.R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87, 2011.

[11] Stéphanie Chollet. *Orchestration de services hétérogènes et sécurisés*. These, Université Joseph-Fourier - Grenoble I, December 2009.

[12] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.

[13] Alessandro Cimatti and Stefano Tonetta. A Property-Based Proof System for Contract-Based Design. In *EUROMICRO-SEAA*, pages 21–28, 2012.

[14] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59, 2012.

[15] The COMPASS project web site. `http://compass.informatik.rwth-aachen.de/`.

[16] Safety and security requirements for fortiss Smart Microgrid demonstrator. Technical Report D1.1, Version 1.4, D-MILS Project, March 2014.

[17] Intermediate languages and semantics transformations for distributed mils – part 2. Technical Report D3.3, Version 1.0, D-MILS Project, July 2014.

[18] Compositional assurance cases and arguments for distributed mils. Technical Report D4.2, Version 1.0, D-MILS Project, April 2014.

[19] Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *In Proc. of Foundations of Software Science and Computation Structures (FOSSACS'05)*, volume 3441 of *LNCS*, pages 299–315. Springer-Verlag, 2005.

[20] Simone Frau, Roberto Gorrieri, and Carlo Ferigato. Petri net security checker: Structural non-interference at work. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2008.

[21] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV*, pages 72–83, 1997.

[22] Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. Smt-based induction methods for timed systems. In *FORMATS*, pages 171–187, 2012.

[23] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[24] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1992.

[25] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[26] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven information flow security for component-based systems. Technical Report TR-2013-7, VERIMAG, 2013. `http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2013-7/`.

[27] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004.

[28] Stefano Tonetta. Abstract model checking without computing the abstraction. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2009.

[29] Ron van der Meyden. Architectural refinement and notions of intransitive noninterference. In Fabio Massacci, Jr. Redwine, SamuelT., and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 5429 of *Lecture Notes in Computer Science*, pages 60–74. Springer Berlin Heidelberg, 2009.

[30] M. Y. Vardi. On the complexity of modular model checking. In *LICS*, pages 101–111. IEEE Computer Society, 1995.