Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Implementation of a Model-Checking Component

Deliverable Number: D5.2b

Version: 0.2

Date: 2016

Public Document

http://into-cps.au.dk

Contributors:

Jörg Brauer, VSI
Florian Lapschies, VSI
Oliver Möller, VSI

Editors:

Jörg Brauer, VSI

Reviewers:

Julien Ouy, CLE
Thierry Lecomte, CLE
Etienne Brosse, ST
Martin Peter Christiansen, AI

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	24-10-2016	Jörg Brauer	Initial document version
0.2	30-11-2016	Jörg Brauer	Reworked document after reviews

Abstract

This deliverable documents the implementation of the model checking component that has been developed as an extension to RT-Tester and its model-based test case generator (RTT-MBT) to support the verification of multi-models. The model checker can be used to verify Linear Temporal Logic (LTL) queries using bounded model checking and supports tailored abstractions that allow the analysis of multi-models consisting of both continuous-time and discrete-event models.

Contents

1 Introduction

In model checking (MC), the behavior of a system or program is formally specified as a model that describes how state changes as the system (and time) progresses. All paths through this model are then exhaustively checked against the system requirements, which are typically expressed in some temporal logic. A typical query, which could be passed to a model checker, is "The system never reaches a bad state", and the task of verifying this query is then left to the model checker.

1.1 Model Checking Primer

This approach, however, may lead to state explosion since the number of states in a system is exponential in the number of system variables, the sizes of their domains, and the number of concurrent components. Because of the computational complexity of model, there has been much interest in improving model checking using the following techniques, or combinations thereof:

Symbolic Model Checking represents states and transitions of a system symbolically, for example, as Boolean formulae. This approach enables states that share some commonality to be represented without duplicating their commonality. Classically, binary decision diagrams (BDDs) have been used for this representation, but SAT-based methods have become popular due to the impressive progress in SAT and SMT solving.

Bounded Model Checking (BMC) examines the transition relation of a system only up to a certain path length, the bound, as opposed to traditional unbounded model checking. The technique is therefore often applied for bug hunting rather than system verification. However, it is important to note that BMC can too be applied for verification if an appropriately large bound is chosen.

Abstraction is based on the key idea of abstracting away from the detailed nature of states. A model checker then operates over classes of states which are related in some sense, rather than individual states. If the number of classes is small, then all paths through the system can be examined without incurring the problems of state explosion. However, abstraction inevitably leads to a loss in precision, and false positives may thus occur.

1.2 Model Checking in INTO-CPS

A major goal of the INTO-CPS project is to develop a framework that supports the development and analysis of cyber-physical systems whose behavior is expressed as a multi-model, which provides the behavioral model required to apply model checking. This setting directly leads to two important observations that affect the implementation of model checking in INTO-CPS:

Concurrency The multi-model consists of a possibly large number of independent components operating in parallel.

Domains The variable domains and computations used in the multi-model may be both, discrete event (DE) or continuous time (CT), and both kinds of sub-models may be combined in a single multi-model.

This setting clearly necessitates the combination of symbolic model checking methods with abstraction. The abstractions developed for model checking in INTO-CPS have been described in [BM15]. The key idea of abstraction in INTO-CPS is to abstract a continuous time model $\mathcal{M}_{CT}$ into a discrete event model $\mathcal{M}_{DE}$ such that $\mathcal{M}_{DE}$ is an over-approximation of $\mathcal{M}_{CT}$. This abstraction mechanism is integrated into a SAT-based bounded model checker for SysML state charts. This model checker has become part of the RT-Tester Model-Based Test Case Generator (RTT-MBT).

1.3 Requirements

This deliverable is based on and aims at implementing the following high level requirements from [LPH+15].

0005 and 0032 Model checking shall be applied to a co-simulation configuration, rather than stand-alone state charts, and hence the outputs of this requirement flow into the model checking functionality of RTT-MBT. In particular, this requires that the abstracted CT models need to be integrated into the co-simulation environment as well so as to replace the original behavior.

0032 DE models, which are part of a configured co-simulation environment, shall be supported using BMC techniques.

0033 and 0034 This requirement is the key scope of this deliverable, as it specifies that RTT-MBT must support DE abstractions of CT models. Further, DE abstractions of CT models shall be handled as if they originally were DE models. To do so, the DE abstractions should be

specified in the same formalism as originary DE models, and can thus be supported by RTT-MBT with as little integration effort as possible.

0036 and 0037 The model checker shall be able to represent a counterexample trace if a violation of the property checked was detected. Such a counterexample trace is invaluable for fixing the implementation [CV03].

0107 The model checking functionality shall be configurable and executable via the INTO-CPS Application.

1.4 Outline

The remainder of this document is structured as follows. First, Sect. 2 discusses related techniques that have been described in the literature. This section is followed by a description of the model checking techniques implemented in RTT-MBT in Sect. 3. The practical application of the RTT-MBT model checker, that is, its configuration and invocation via the command-line interface, and the configuration items for model checking in INTO-CPS are described in Sect. 4. Finally, Sect. 5 provides a conclusion.

2 Related Work

This section discusses various important contributions to the areas of (symbolic) model checking, bounded model checking and abstract interpretation, which have to some extent been incorporated in the INTO-CPS model checking framework.

2.1 Temporal Logic & Model Checking

In its general setting, model checking amounts to aswering the question whether a model $\mathcal{M}$ satisfies its specification φ, formally $\mathcal{M} \models \varphi$. Two temporal logics — namely computation tree logic [CES86] (CTL) and linear temporal logic [Pnu77] (LTL) — are supported by virtually any model checker for discrete event systems. RTT-MBT supports LTL rather than CTL, which is justified as follows:

- LTL formulas are interpreted over (infinite) linear execution sequences of a system or model, whereas CTL considers branches of sequences. As RTT-MBT is not only a model checker but also a test case generation

framework, its main goal is reasoning about linear executions, which is why LTL is preferred.

- Further, CTL suffers from the fact that it is a branching time formalism: Reasoning about branching time is unintuitive and thus hard to use for non-experts in temporal logic, which hinders the practical application of this formalism in industry. In fact, it has been observed that "*nontrivial CTL equations are hard to understand and prone to error*" [SBF+97] and "*CTL is difficult to use for most users and requires a new way of thinking*" [BBL98].
- Whereas LTL allows compositional reasoning, and thus allows model checking backends to be improved by integrating compositional techniques, CTL is non-compositional [Var01, NV07].

Initially, the problem $\mathcal{M} \models \varphi$ for LTL was solved by using a construction based on Büchi automata [VW86]. Intuitively, this appproach represents the system and the LTL specification as Büchi-automata, and then algorithmically checks whether the intersection of system and the negation of the specification is non-empty. In this case, a violation of the specification has been detected. Later, symbolic methods have been introduced, which solve the LTL model checking problem by representing both the system and specification as Boolean formulae, see [BCCZ99, BHJ+06]. Comprehensive introductions to temporal logics, model checking and the core algorithms are given by Clarke et al. [CGP99] as well as Baier and Katoen [BK08]. RTT-MBT uses symbolic techniques based on propositional encodings of LTL specifications, as will be discussed in Sect. 3.

2.2 Bounded Model Checking

The key idea of BMC is to exercise the behavior of a system only up to a certain depth of computations [BCCZ99, CBRZ01, CKOS05]. BMC has been established as a valuable bug-hunting framework for hardware and software [CKL04], which is motivated by the observation that bugs can often be found after few computation steps if only the right inputs are chosen. However, it has been observed that bounded model checking can also be applied for formal verification if the unrolling depth k of the transition relation is large enough. Precisely, the the unrolling depth k has to match the completeness threshold c of the system, which can intuitively be described as: If no counterexample of length c or less is found, the specification holds for all (infinite) executions of the model. Hence, BMC with $k \geq c$ suffices for

proving correctness of a system [BCCZ99, Thm. 27]. However, computing the completeness threshold is as least as hard as solving the model checking problem itself [CKOS04, KOS+11]. Consequently, BMC is often used for verification up to a certain bound, without giving an actual correctness guarantee for nonterminating executions of the system.

2.3 Abstraction

The foundations of abstraction have been formalized by Cousot & Cousot [CC77] in the abstract interpretation framework. In principle, the semantics of a program is specified using lattices. Lattices A and C are then used to specify state in the concrete and abstract domains, respectively, and importantly these lattices are connected by an abstraction function $\alpha : A \to C$ and a concretization function $\beta : C \to A$. For $c \in C$ and a correct abstraction function α, the value $\alpha(c)$ then describes c in the sense that it contains c, and possibly more values. This form of imprecision preserves soundness, but may lead to false positive (or spurious) warnings.

Often, abstract systems are sufficient to prove interesting system properties. However, if this is not the case, the abstraction has to be refined into a more precise representation of the concrete system semantics, an approach that has widely been automated using techniques such as counterexample guided abstraction refinement [GS97].

However, of course abstract interpretation techniques have widely been applied to the verification of hybrid systems [Hen96]. For example, Sankaranarayanan et al. [SDI08] have combined symbolic model checking with states encoded on top of template polyhedra, that is, conjunctions of linear inequalities $\sum_{i=0}^{n} c_i \cdot v_i \leq k$ where the c_i are fixed a priori. However, such works target an entirely different setting than our work since it is entirely based on abstracting formally specified hybrid automata, whereas we focus on continuous-time models that may not necessarily have a formal semantics (the outputs may, for example, be computed using a controller that is directly connected to the system). Further, the scalability of complex abstractions such as template polyhedra in a network of components is uncertain. As stated by Sankaranarayanan et al. [SDI08, Sect. 1], "*hybrid systems verification is a challenge even for small systems*", which of course applies to networks of hybrid systems.

3 Model Checking Implementation

The model checking implementation for INTO-CPS uses the standard SAT-based approach to BMC. Formally, let $\mathcal{I}$ denote an encoding of the initial states of the system and let $\mathcal{T}(s_i, s_{i+1})$ denote an encoding of a single transition from pre-state s_i to post-state s_{i+1}. The semantics of the system for k execution steps is then fully described by:

$$\mathcal{I} \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}(s_i, s_{i+1})$$

If the LTL specification is encoded as a formula φ, then the system contains no violation of the specification if the conjoined formula

$$\mathcal{I} \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}(s_i, s_{i+1}) \wedge \neg\varphi$$

is unsatisfiable. The BMC problem can thus simply be solved by deriving a formula of the above form and letting an off-the-shelf SAT or SMT solver search for a solution to this formula.

3.1 Encoding for Co-Models

The implementation of model checking in RTT-MBT is based on the transition relation encoding described in [HPW15]. We refrain from repeating the encoding here and refer the reader to [HPW15, Sect. 4] for a detailed description, and instead describe the approach intuitively.

Initially, RTT-MBT transforms a UML/SysML model so that it consists of a collection of n of blocks $B_1, \ldots, B_n$. All blocks are derived from a set of (possibly hierarchical) state machines, which are executed concurrently. Concurrency is modelled using interleaving semantics, that is, only one operation is executed at a time, which entails that each operation executes atomically. The order of execution, however, is nondeterministic: From any block, any operation whose precondition is satisfied can be executed next. The transition relation can then be encoded as a disjunction over all blocks (which represent state machines) and all operations (which represent transitions).

This approach dovetails with multi-modelling, since each component in the co-model itself can be interpreted as a set of concurrent state machines. If

two more such components are used in a single co-model, then the overall semantics is simply be represented by disjunctively adding the encodings of the state machines that constitute the respective component. Hence, replacing a continuous-time component C by its abstract counterpart $\alpha(C)$ simply amounts to replacing C by an abstract representation [1].

3.2 Encoding the Specification

The LTL encoding algorithm has to provide techniques to encode the following kinds of LTL formulae:

- atomic propositions
- logical connectives ($\vee$, $\wedge$, $\neg$)
- Globally operator (G)
- Finally operator (F)
- Next-state operator (X)
- Until operator (U)

For each of these operations, RTT-MBT uses the propositional encoding discussed in [BHJ+06]. Given that the semantics of an LTL formula is specified over single executions, it is straightforward to implement these encodings.

Example Consider the LTL formula $\varphi = \mathsf{G}(\mathtt{x} = 0 \vee \mathtt{y} \geq 0)$, which states that in any reachable state, at least one of the atomic propositions $\mathtt{x} = 0$ and $\mathtt{y} \geq 0$ must hold. For each unrolling step $0 \leq i \leq k$, let $\mathtt{x}_i$ and $\mathtt{y}_i$ denote the state of $\mathtt{x}$ and $\mathtt{y}$. Then, for each single step, the property is specified as $(\mathtt{x}_i = 0 \vee \mathtt{y}_i \geq 0)$. Since the atomic proposition is required to hold globally, the LTL property φ is simply encoded as:

$$\mathsf{encode}(\varphi, 0, k) = \bigwedge_{i=0}^{k} (\mathtt{x}_i = 0 \vee \mathtt{y}_i \geq 0)$$

[1] The construction of the abstractions is discussed in deliverable [BM15]. This step will be discussed further in Sect. 3.3.

In the general case, the encoding for $\varphi = \mathsf{G}\psi$ where ψ is itself an LTL formula is:

$$\mathsf{encode}(\varphi, 0, k) = \bigwedge_{i=0}^{k} \mathsf{encode}(\psi, i, k)$$

Example As another example, consider the LTL formula $\varphi = \mathsf{F}(\mathtt{x} \leq \mathtt{y} + \mathtt{z})$, which expresses that in any execution of the system $(\mathtt{x} \leq \mathtt{y} + \mathtt{z})$ must eventually holds. As before, this property can be encoded as:

$$\mathsf{encode}(\varphi, 0, k) = \bigvee_{i=0}^{k} (\mathtt{x}_i = \mathtt{y}_i + \mathtt{z}_i)$$

The general case for $\varphi = \mathsf{F}\psi$ is:

$$\mathsf{encode}(\varphi, 0, k) = \bigvee_{i=0}^{k} \mathsf{encode}(\psi, i, k)$$

3.3 Configuration & Abstractions

3.3.1 Basic Configuration

The RTT-MBT model checker currently depends on the following basic configuration items:

- the unrolling depth $k \geq 0$
- the LTL formula φ
- the multi-model $\mathcal{M}$

The unrolling depth and the LTL formula have their obvious meaning. The multi-model is the overall system model that is exported from Modelio via XMI. This multi-model contains a connection diagram, which expresses which ports and signals are used as inputs and outputs of each sub-model. Configuring these items is the only necessity for executing the RTT-MBT model checker. Details regarding the configuration of these parameters via the command-line interface and the INTO-CPS application, respectively, are given in Sect. 4.2 and Sect. 4.3.

3.3.2 Configuration of Abstractions

The Year 1 deliverable [BM15] describes three abstractions (also referred to as model approximations) for model checking of co-models consisting of both, DE and CT systems. Here, we briefly repeat the details of these abstractions and their current status in the RTT-MBT model checker [2].

Interval Abstraction uses the well-known box or interval domain [CC77] to bound the range of continuous variables. RTT-MBT allows to specify upper and lower bounds for all variables of a co-model, and both the model checker and the test case generator allow input variables to hold values only in the provided range. An example of an interval abstraction for a continuous variable is given in Fig. 1. To implement this functionality for a variable $\mathtt{v}$, the component that computes abstracted value has to be removed from the input to RTT-MBT. Suppose that $\mathtt{l_v}$ and $\mathtt{u_v}$, respectively, denote the lower and upper bounds of $\mathtt{v}$. The transition relation is conjoined with an additional constraint:

$$\bigwedge_{i=0}^{k} \mathtt{l_v} \leq \mathtt{v}_i \leq \mathtt{u_v}$$

Let $\mathcal{V}$ denote the set of variables to be abstracted, and assume that each $\mathtt{v} \in \mathcal{V}$ is computed by a component $\mathcal{C}_\mathtt{v}$. Further, let $\mathtt{remove}_{\{\mathcal{C}_1,\ldots,\mathcal{C}_n\}}$ denote the operation that removes a set of components from the transition relation. Then, the overall formula to be solved is given by:

$$\mathcal{I} \wedge \bigwedge_{i=0}^{k-1} \mathtt{remove}_{\{\mathcal{C}_\mathtt{v} | \mathtt{v} \in \mathcal{V}\}}(\mathcal{T}(s_i, s_{i+1})) \wedge \neg\varphi \wedge \bigwedge_{i=0}^{k} \bigwedge_{\mathtt{v} \in \mathcal{V}} (\mathtt{l_v} \leq \mathtt{v}_i \leq \mathtt{u_v})$$

Intuitively, the computation of a variable $\mathtt{v}$ is removed from the transition relation, and in each unrolling step a value that satisfies the bounds constraint may be chosen.

Gradient-Based Interval Abstraction is more constrained than interval abstraction as it too specifies to what extent the value of an input variable may change in a given time unit, which has to be specified manually by the designer of the abstraction. For instance, interval abstraction allows a variable to first equal its lower bound, and then, in the next computation

[2] The current release of RTT-MBT provides stable support for interval-based abstraction.

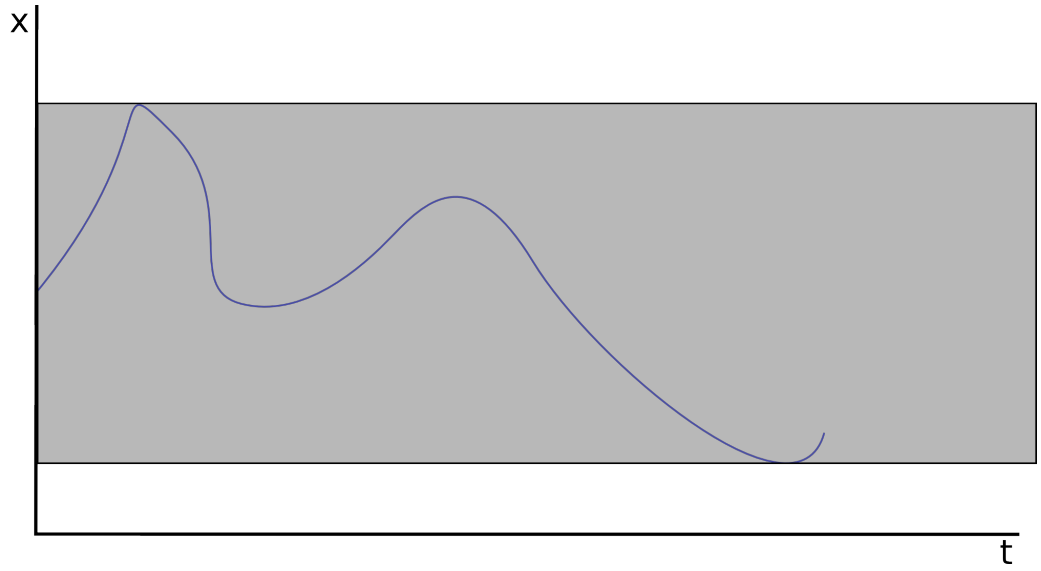

Figure 1: Interval abstraction for a continuous variable.

step, equal its upper bound, which may be unrealistic and lead to spurious results. Gradient-based interval abstraction can be used to specify how a continuous variable may gradually change its value, for example, the value of an input variable may change by at most five units in 10 ms. If $g_{\mathtt{v}}$ denotes the gradient for a variable $\mathtt{v}$, the constraint that shall be satisfied by a variable $\mathtt{v}$ in unrolling step i is given by:

$$(|\mathtt{v}_i - \mathtt{v}_{i-1}| \leq g) \wedge (\mathtt{l}_{\mathtt{v}} \leq \mathtt{v}_i \leq \mathtt{u}_{\mathtt{v}})$$

Intuitively, the difference between to consecutive valuations of $\mathtt{v}$ must not exceed g and $\mathtt{v}$ must always respect its defined range. The system behavior admitted by this abstraction is thus a subset of the behavior admitted by interval abstraction.

Simulation-Based Abstraction Interval abstraction and gradient-based abstraction may, when applied to industrial-scale multi-model co-simulation environments, lead to some frustration since the abstractions may be either too coarse — too many spurious warnings are produced — or too expensive — the tool does not terminate in a reasonable amount of time. We have therefore proposed an additional strategy, which is based on signal values extracted from execution logs of the system as it is running. Suppose that a co-model is executed and the value of a continuous variable $\mathtt{v}$ in this execution

is characterized by a function $f_{\mathtt{v}} : \mathbb{Q} \to \mathbb{Q}$, which yields a concrete value for a given time stamp [3].

A variable $\mathtt{v}$ is then characterized by a finite sequence of intervals:

$$[\mathtt{l}_{\mathtt{v},0}, \mathtt{u}_{\mathtt{v},0}], \ldots, [\mathtt{l}_{\mathtt{v},n}, \mathtt{u}_{\mathtt{v},n}]$$

This sequence is computed as follows: Define a bound c that denotes the maximum size of an interval, that is, $|\mathtt{u}_{\mathtt{v},i}, \mathtt{l}_{\mathtt{v},i}| \leq c$ must hold for each $0 \leq i \leq n$. Given a time stamp t, we then compute the largest time stamp $t' > t$ such that $\forall t'' \in [t, t'] : |f_{\mathtt{v}}(t) - f_{\mathtt{v}}(t'')| \leq c$ based on the execution log. Starting with $t = 0$, this step is performed iteratively until the end of the execution log is reached, where each iteration i induces an interval:

$$[\mathtt{l}_{\mathtt{v},i}, \mathtt{u}_{\mathtt{v},i}] = [\min(\{f_{\mathtt{v}}(t'') \mid t'' \in [t, t']\}), \max(\{f_{\mathtt{v}}(t'') \mid t'' \in [t, t']\})]$$

This behavior can then simply be expressed as a state machine consisting of a single successor chain of constraints.

3.4 Summary of Configuration Items

We conclude this section with a summary of the configuration items required by each abstraction:

Interval Abstraction: A set of input variables with boundaries for each variable.

Gradient-Based Interval Abstraction: A set of input variables to be abstracted, and for each such variable a pair of a gradient value and a time frame.

Simulation-Based Abstraction: An execution log, a set of input variables, and for each input variable an upper bound for the value range covered by a single interval.

4 Application of Model Checking

Model checking functionality is provided by the RTT-MBT toolchain, the installation of which is described in Sect. 4.1. A description of how to use

[3] The execution log has to discretize the progress of time and the value of $\mathtt{v}$, hence the choice of $\mathbb{Q}$ rather than $\mathbb{R}$.

model checking via the command-line interface and with the INTO-CPS application is given in the following sections.

4.1 Installation of the RTT-MBT

For model checking with RTT-MBT, a number of software packages must be installed. These software packages have been bundled into two installers:

- **VSI tools dependencies bundle:**
  This bundles is required on the Windows platform and installs the following third party software:
  - Python 2.7.
  - GCC 4.9 compiler suite, used to compile FMUs.
- **VSI tools – VSI Test Tool Chain:**
  - RT-Tester 6.0, a stripped version of the RT-Tester core test system that contains the necessary functionality for INTO-CPS.
  - RT-Tester MBT 9.0, the model-based testing extension of RT-Tester.
  - RTTUI 3.9, the RT-Tester graphical user interface.
  - Utility scripts to run RTT-MBT.
  - Examples for trying out RTT-MBT.

These bundles can either be downloaded via the download manager of the INTO-CPS Application or can come pre-installed in VirtualBox images.

4.2 Command-Line Interface

The executable responsible for processing model checking queries is called `rtt-mbt-mc` and can be found in the RTT-MBT installation folder. This executable can be used as a command-line tool for checking LTL queries. In order to check an LTL query the following arguments need to be provided as command-line arguments:

- `-model <XMI-FILE>` specifies the model on which model checking should be performed. The model should be in the form of an XMI file exported by Modelio.
- `-spec <LTL-QUERY>` specifies the LTL-query string to check.
- `-bound <NUMBER>` specifies the upper bound k for BMC (see Sect. 3.2).

After checking the query, the tool responds with one of the following verdicts:

- The LTL formula is satisfied up to a bound of k. No counterexample could be found.
- The LTL formula does not hold. A counterexample is also printed.

When specifying the LTL formula string, the operators introduced in Sect. 3.2 have the following textual representation:

- The logical connectives ($\vee$, $\wedge$, $\neg$) are represented by `||`, `&&`, and `!`, respectively.
- Globally operator (G): `Globally ([...])`
- Finally operator (F): `Finally ([...])`
- Next-state operator (X): `Next ([...])`
- Until operator (U): `([...]) Until ([...])`

Model variables found in the model can be used in atomic propositions. Atomic propositions must always be enclosed by brackets ([and]). For instance, the following strings are valid LTL specifications over variables `x` and `y` in the syntax supported by RTT-MBT:

$$\mathsf{F}([x > 0 \;||\; x == y])$$

$$\mathsf{G}([x >= 0 \;\&\&\; y >= 0])$$

$$\mathsf{F}(\mathsf{G}([x >= 1]))$$

In addition, for each location in a state machine a variable with the same name as the originating location is introduced. These variables evaluate to true if and only if the model is residing in the respective control state.

Example Invocation: As an example, we utilize the turn-indicator model. Assume that we want to verify the property that both turn-indicator lamps must always be turned off if neither emergency flashing is active nor left or right flashing is selected using the turn indicator lever. This property can be expressed using the following LTL specification:

$$\mathsf{G}(\text{TurnIndLvr} \neq 0 \vee \text{EmerSwitch} = 1 \vee (\text{LampsLeft} = 0 \wedge \text{LampsRight} = 0))$$

If we want to check this property with a bound of 50 steps, we invoke the model checker as follows:

```
rtt-mbt-mc \
  -model turnInd.xmi \
  -bound 50 \
  -spec 'Globally ([TurnIndLvr != 0 || EmerSwitch || \
                    (LampsLeft == 0 && LampsRight == 0)])'
```

The model checker then reports that the LTL formula does not hold. We further inspect the output of `rtt-mbt-mc` listed below, which reveals that there is the (desired) functionality in model that allows the lamps to flash for three times even after the turn indicator lever has been released.

```
RT-Tester Model-Based Test Generator (Release 9.0-1.5.8-x86_64.fad12af3a290fbf330882201bf1ed1fa10eeaaa3.dev)

[SOLVER] Reached goal after 50 steps within 60.146s.

Found counterexample to [!([true] Until !([(((IMR.TurnIndLvr@0 != 0) || IMR.EmerSwitch@0)
  || ((IMR.LampsLeft@0 == 0) && (IMR.LampsRight@0 == 0)))]))].

IMR.LampsLeft@50 |-> (ConcreteLattice<signed int>, 0)
IMR.LampsRight@50 |-> (ConcreteLattice<signed int>, 1)
IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_OFF@50 |-> (ConcreteLattice<bool>, 1)
IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.EMER_ACTIVE@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.TURN_IND_OVERRIDE@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.initial@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.initial@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.FLASH_CTRL.tilOld@50 |-> (ConcreteLattice<signed int>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.FLASHING.OFF@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.FLASHING.ON@50 |-> (ConcreteLattice<bool>, 1)
IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.FLASHING.initial@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.IDLE@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.OUTPUT_CTRL.initial@50 |-> (ConcreteLattice<bool>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.ctr@50 |-> (ConcreteLattice<signed int>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.lOld@50 |-> (ConcreteLattice<signed int>, 0)
IMR.SystemUnderTest.OUTPUT_CTRL.rOld@50 |-> (ConcreteLattice<signed int>, 1)
IMR.SystemUnderTest.OUTPUT_CTRL.t@50 |-> (ConcreteLattice<unsigned int>, 0)
IMR.SystemUnderTest.left@50 |-> (ConcreteLattice<signed int>, 0)
IMR.SystemUnderTest.right@50 |-> (ConcreteLattice<signed int>, 1)
_timeTick@50 |-> (ConcreteLattice<unsigned int>, 256)
IMR.EmerSwitch@50 |-> (ConcreteLattice<signed int>, 0)
IMR.TurnIndLvr@50 |-> (ConcreteLattice<signed int>, 0)
IMR.voltage@50 |-> (ConcreteLattice<float>, 13.0000029)
```

4.3 User Interface Integration

Once an INTO-CPS project has been created, model checking functionality can be found under the top-level activity *MODEL-CHECKING* in the project browser.

Starting the License Management Process: Before getting started, the RT-Tester license management process has to be launched. To this end, right-click on *MODEL-CHECKING* and select *Start RT-Tester License Dongle* (see Fig. 2).

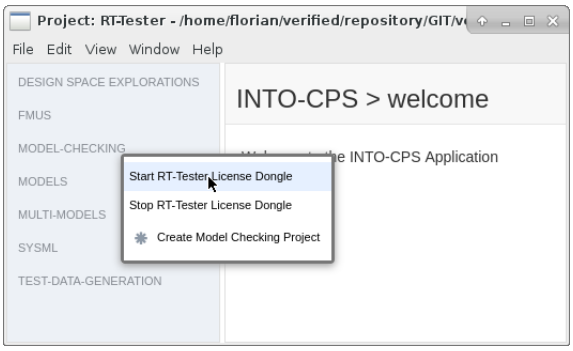

Figure 2: INTO-CPS Application: Start RT-Tester license dongle

Creating a new Model Checking Project: Model checking projects are presented as sub-projects of INTO-CPS application projects. In order to add a new project, right-click on the top-level activity *MODEL-CHECKING* in the project browser and select *Create Model Checking Project* (see Fig. 3). Then, the user has to provide a project name and the model that has been ex-

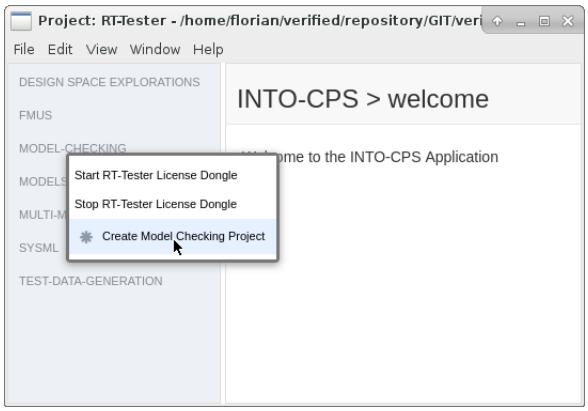

Figure 3: INTO-CPS Application: Creating a model checking project

ported to XMI from Modelio. After pressing *Create Model Checking Project*, a new node representing the model checking project is added to the project browser.

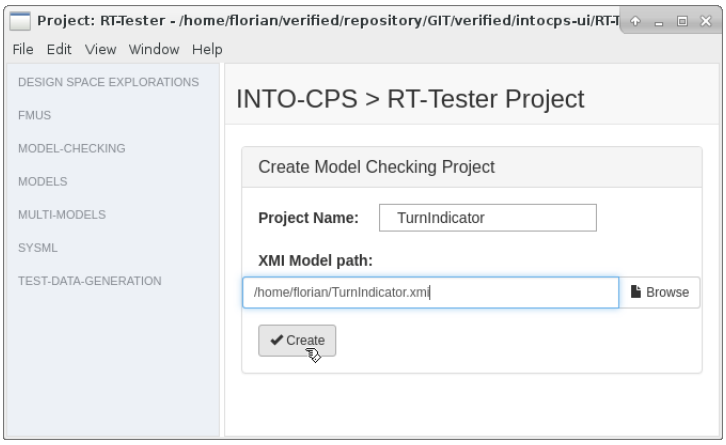

Figure 4: INTO-CPS Application: Model checking creation dialog

Adding, Editing and Checking LTL Queries: The next step is to add LTL queries to the project. To add a query, right click on the project and select *Add LTL Query* (see Fig. 5). Then enter a name for the new query

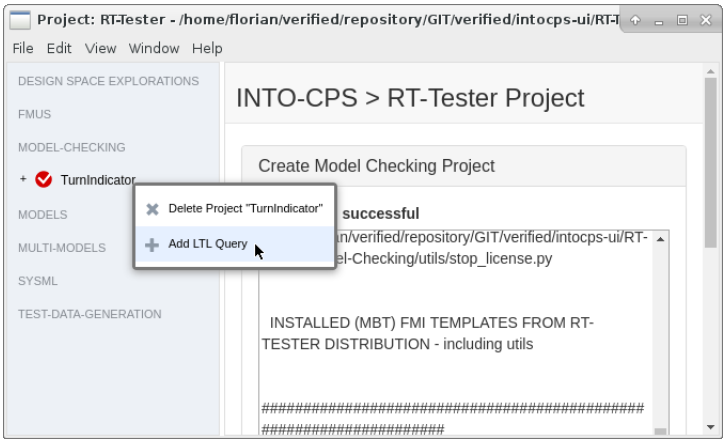

Figure 5: INTO-CPS Application: Adding an LTL formula

(see Fig. 6). To edit the LTL query, double click on the corresponding node in the project browser (see Fig. 7). The LTL formula can then be edited in a text field. Note that the editor supports auto-completion for variable names and LTL operators (see Fig. 8). The user must also provide the upper bound k for the bounded model checking query (see Sect. 3.2). To check the query, press *Save & Check*. A window will open which is being filled with the output of the `rtt-mbt-mc` model checking tool described in Sect. 4.2. The tool either reports that the query holds within the specified number of steps

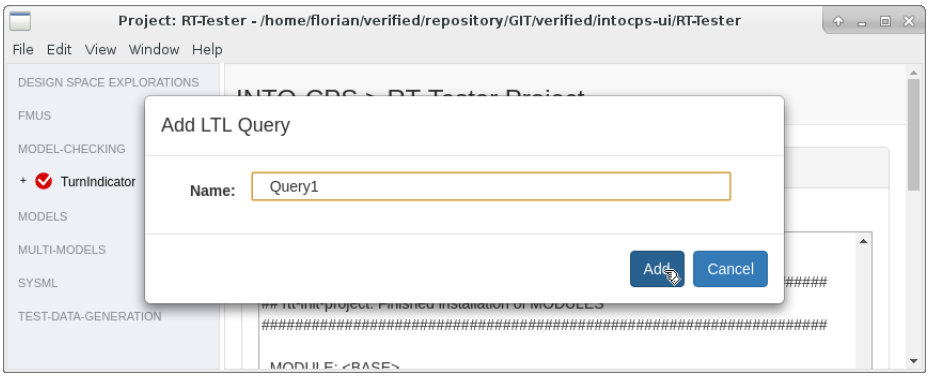

Figure 6: INTO-CPS Application: LTL formula creation dialog

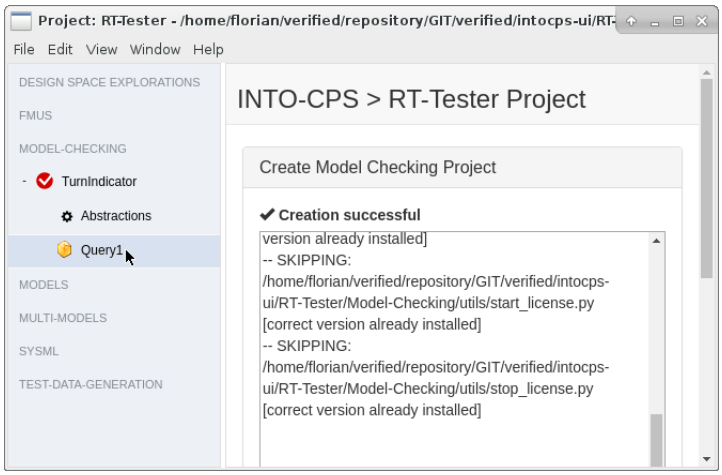

Figure 7: INTO-CPS Application: Open LTL editor

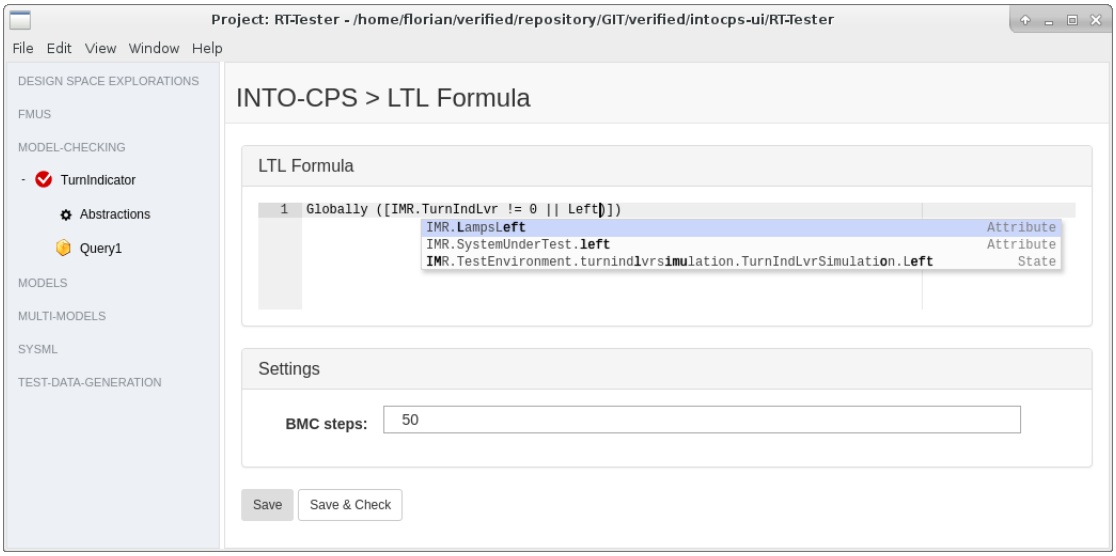

Figure 8: INTO-CPS Application: LTL formula editor

— as depicted in Fig. 9 — or it will print a counter example to demonstrate that the property does not hold.

Configuring Abstractions: To configure abstractions for a particular model checking project, double-click on the corresponding *Abstractions* node below that project in the project browser. It is then possible to choose an abstraction method for each output variable of an environment component along with making the associated setting. In Fig. 10 the interval abstraction has been selected for the output variable *voltage*. This abstraction has further been configured to restrict the variable's value within the interval $[10, 12]$. After pressing *Save*, these abstraction will be applied to all model checking queries in the current model checking project.

5 Conclusion & Outlook

This deliverable provides an overview of the LTL model checker for SysML multi-models that has been developed in the context of the INTO-CPS project. A noteworthy particularity of the project's setting is that it targets heterogeneous systems which combine both, DE and CT models. The implemented model checker, however, provides direct support for DE models, but abstracts from CT models using configurable abstraction mechanisms.

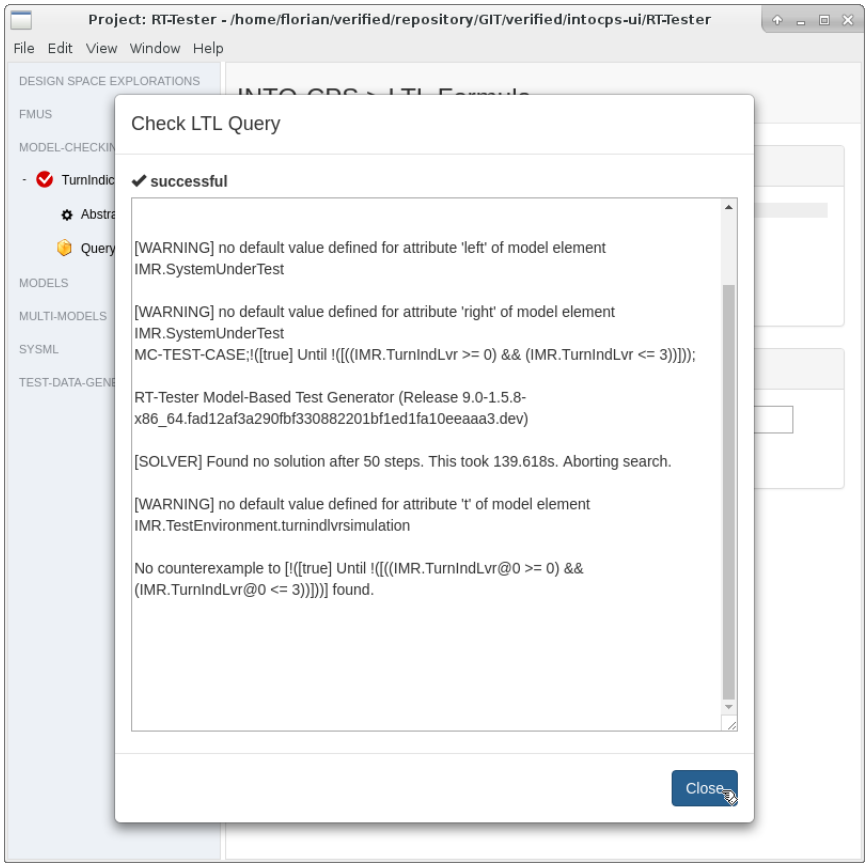

Figure 9: INTO-CPS Application: Model checking result

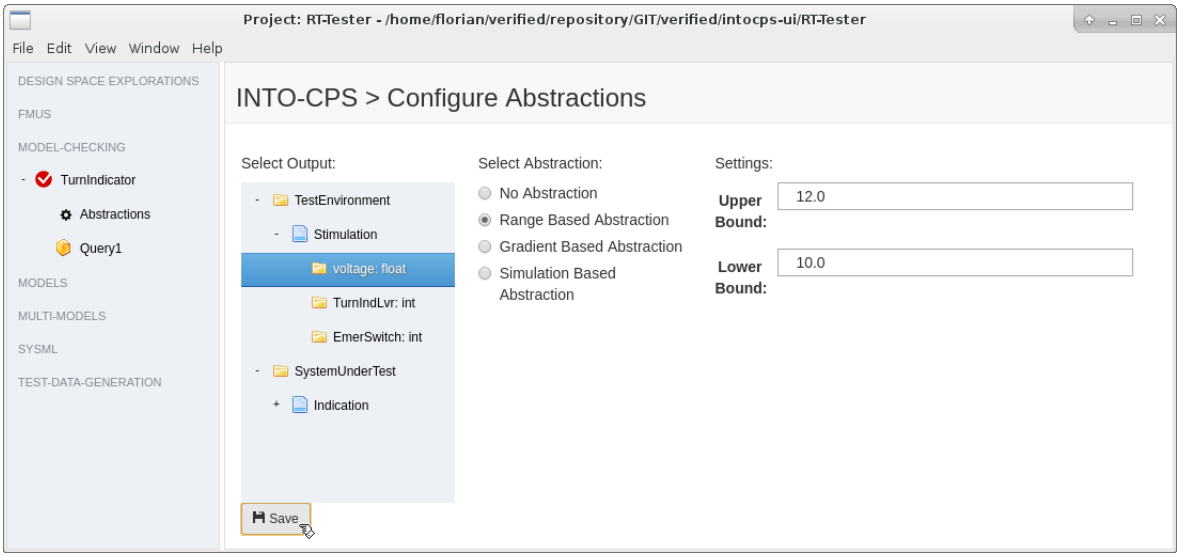

Figure 10: INTO-CPS Application: Configure Abstractions

In the near future, further abstractions — based on the explicit needs from case studies — will be integrated into the model checker, and the integration in the INTO-CPS application will be extended accordingly.

References

[BBL98] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In Alan J. Hu and Moshe Y. Vardi, editors, *10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 1998.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi A. Juntilla, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

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

[BM15] Jörg Brauer and Oliver Möller. Abstraction Techniques from CT to DE. Technical report, INTO-CPS Deliverable, D5.1c, December 2015.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming (POPL 1977)*, pages 238–252. ACM, 1977.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podel-

ski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[CKOS05] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[CV03] E.M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In R.P. Kurshan M.K. Inan, editor, *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HPW15] Christoph Hilken, Jan Peleska, and Robert Wille. A unified formulation of behavioral semantics for sysml models. In Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe, editors, *3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 263–271. SciTePress, 2015.

[KOS+11] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear Completeness Thresholds for Bounded Model Checking. In *23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2011.

[LPH+15] Peter Gorm Larsen, Ken Pierce, Francois Hantry, Joey W. Coleman, Sune Wolff, Kenneth Lausdahl, Marcel Groothuis, Adrian Pop, Miran Hasanagić, Jörg Brauer, Etienne Brosse, Carl Gamble, Simon Foster, and Jim Woodcock. Requirements Report year 1. Technical report, INTO-CPS Deliverable, D7.3, December 2015.

[NV07] Sumit Nain and Moshe Y. Vardi. Branching vs. linear time: Semantical perspective. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2007.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57. ACM, November 1977.

[SBF+97] Thomas Schlipf, Thomas Buechner, Rolf Fritz, Markus M. Helms, and Juergen Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4&5):567–576, 1997.

[SDI08] S. Sankaranarayanan, T. Dang, and F. Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Annual Symposium on Logic in Computer Science (LICS)*, pages 332–334. IEE Computer Society Press, 1986.

A List of Acronyms

20-sim	Software package for modelling and simulation of dynamic systems
ACA	Automatic Co-model Analysis
AST	Abstract Syntax Tree
AU	Aarhus University
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CLE	ClearSy
CLP	Controllab Products B.V.
COE	Co-simulation Orchestration Engine
CPS	Cyber-Physical Systems
CT	Continuous-Time
DE	Discrete Event
DESTECS	Design Support and Tooling for Embedded Control Software
DSE	Design Space Exploration
FMI	Functional Mockup Interface
FMI-Co	Functional Mockup Interface – for Co-simulation
FMI-ME	Functional Mockup Interface – Model Exchange
FMU	Functional Mockup Unit
LTL	Linear Temporal Logic
MC	Model Checking
RTT-MBT	RT-Tester Model Based Test Case Generator
SAT	SATisfiable Boolean formula, a symbolic representation of terms that can/should evaluate to *true*
SMT	Satisfiability Modulo Theories, i.e., a SAT formula interpreted over a logical theory (here, this describes a system design)
ST	Softeam
SysML	Systems Modelling Language
TWT	TWT GmbH Science & Innovation
UML	Unified Modelling Language
UNEW	University of Newcastle upon Tyne
UTRC	United Technologies Research Center
UY	University of York
VSI	Verifdied Systems International
WP	Work Package
XML	Extensible Markup Language