Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Implementation of a Model Checking Component for Global Model Checking

Deliverable Number: D5.3c

Version: 1.0

Date: 2017

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
Etienne Brosse, ST
Frederik Foldager, 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	24-08-2017	Jörg Brauer	Writing Plan
0.2	06-11-2017	Jörg Brauer	Ready for internal review
1.0	08-12-2017	Jörg Brauer	Reworked after internal reviews

Abstract

This deliverable presents the overall state of the INTO-CPS model checking capabilities at the end of year 3 of the project. The particular focus of this document is the representation of continuous-time behavior for integration with model checking, and the configuration of the required abstractions in the INTO-CPS Application. Additionally, this deliverable provides information on tracing of the model checking activity in the context of the development of safety-critical systems, where traceability is a core part of the development lifecycle.

Contents

1 Introduction

This document is concerned with verifying global system properties of multi-models – which are themselves expressed as a collection of continuous-time (CT) and discrete event (DE) models, and the integration of the model checking component in the INTO-CPS project. To address these topics, the model checking component developed for INTO-CPS is discussed from two different perspectives:

- From a technical perspective, it is discussed which formal representations and abstraction mechanisms are implemented to support the verification of system-wide properties of multi-models.
- The integration in the INTO-CPS project focusses on the end-user, and how model checking features can be used during a project. In this regard, particular focus is set to the value of traceability for model checking.

1.1 Model Checking of Mixed Multi-Models

The nature of multi-models as defined in INTO-CPS is not dissimilar to general hybrid systems, which can be seen as systems containing both, physical components that evolve over time and discrete components that may influence the continuous dynamics. Despite the substantial effort that was put into the development of hybrid model checking, there are still open questions that need to be answered in order to increase the applicability and usability of hybrid model checking tools. We therefore deviate from representing multi-models by what they are — hybrid systems — and consider DE abstractions of their continuous behaviors, which can be seen as a response to the scalability issues of hybrid model checking.

A classical approach to representing hybrid systems is to apply hybrid automata [ACH+95, Hen96], which are a formalism that can accurately describe systems composed of a mixture of discrete and continuous behaviors by expressing the behavior of continuous variables using ordinary differential equations. A textbook example of an execution of a hybrid automaton is given in Fig. 1, depicting the behavior of a bouncing ball dropped from some initial height with zero initial velocity. Due to gravity, the ball initially accelerates towards the ground and falls until it hits the ground. It then bounces back whilst losing some of its kinetic energy, and raise again.

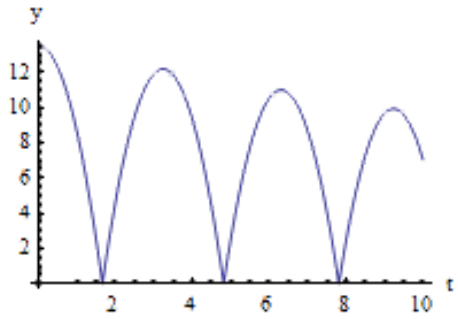

Figure 1: Continuous-time behavior of a bouncing ball.

There is, of course, a multitude of ways of the continuous behavior of a bouncing ball could be represented using a discrete abstraction. In the following, Fig. 2 and Fig. 3 show two abstractions used in the INTO-CPS Application [BM15, BLM16]. Figure 2 abstracts the entire state space of y over t using one tight interval that contains all potential values. Figure 3 shows how the hybrid behavior can automatically be abstracted using a sequence of intervals which are based on a concrete simulation, which yields much more precise results. In this approach, time-discretization is applied and one interval is used to capture all values of y within a fixed-size timeframe.

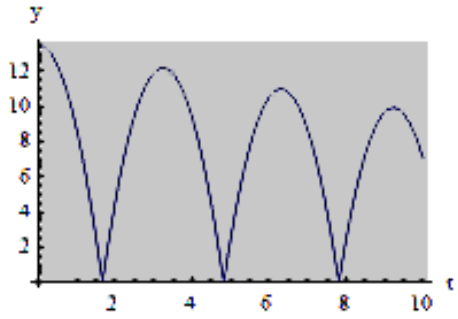

Figure 2: Continuous-time behavior of a bouncing ball abstracted via a single interval.

Approaches for deriving and handling such interval abstractions have been the topic of previous deliverables [BM15, BLM16]. The focus of this deliverable is the combination of multi-models for model checking, and the application in the INTO-CPS project.

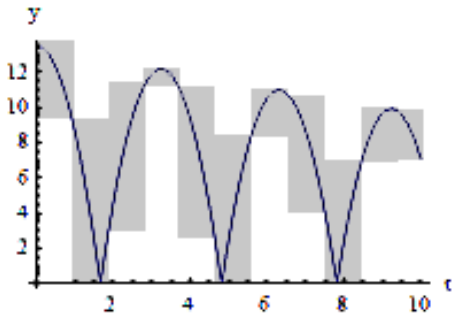

Figure 3: Continuous-time behavior of a bouncing ball via an adaptive sequence of intervals.

1.2 Traceability for Model Checking

Over the past two decades, model checking in its various flavors has made impressive progress in terms of applicability in industry. As one example, the approach has become a standard technique for the verification of hardware circuits. As another example, this author's company applies model checking to verify interlocking system configurations for customers from railway industry, where the configurations are specified in a domain-specific language and then automatically verified for sanity and safety.

Apart from the capability to automatically verify system properties provided in temporal logic, a question that commonly arises in industrial projects in safety-critical domains is that of tracing. For the technique to be of value for certification-related activities, it must be possible to identify and relate the configurations and results of the verification process, which naturally leads to the notion of *traceability*, the purpose of which is to relate certain information and artefacts related to the verification process. Questions to which a traceability implementation should be able to automatically provide answers include:

- Is requirement X satisfied for model Y?
- Which model checking queries exercise requirement X?
- Which is the last model version for which model checking query X failed?

It is no surprise that traceability typically is stored as some form of database which relates entities according to a well-defined scheme. We do not contribute to these fundamentals, but rather present a scheme that appears

well-suited for covering industrial needs related to model checking for safety-critical systems.

1.3 Outline

The remainder of this document is structured as follows. First, Sect. 2 discusses related work with respect to model checking and traceability. Then, the formal foundations of model checking in the INTO-CPS project are discussed in Sect. 3, which is followed by a description of the implementation of model checking and traceability in the INTO-CPS Application. The deliverable concludes with a discussion.

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. Related work on traceability in general is not discussed, since this document only provides details on the specific implementation of traceability for the model checking component. For an overview of the INTO-CPS approach to traceability, we refer the reader to [KLN+17].

2.1 Temporal Logic & Model Checking

In its general setting, model checking amounts to answering 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 approach 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]. Our model checker, RTT-MBT, uses symbolic techniques based on propositional encodings of LTL specifications, as will be discussed in Sect. 4.

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 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.

2.4 Traceability for Verification and Validation

The value of traceability for verification and validation activities as allowing the verification of properties which are spread over different develop-

ment artefacts and versions thereof. Virtually all industrial standards for safety-critical systems, such as the RTCA DO-178B [WG-92], the RTCA DO-178C [WG-11], the CENELEC EN 50128 [CEN99] or the ISO 26262 [ISO11] are aligned to this view and require traceability of both, development and verification artefacts. The model checking component integrated in the INTO-CPS supports this view and allows to both store and retrieve identified artefacts, and relations among them. We refer the read to [LNH+16, KLN+17] for a thorough discussion of related work.

3 Foundations of Model Checking in INTO-CPS

In the context of the INTO-CPS project, multi-models are considered, that is, networks of heterogeneous system components, the combination of which forms the entire system. To the model checker, each system component is a possibly hierarchical state machine, the behavior of which can be represented via its transition relation. Based on this representation of components, a multi-model is represented as a network hierarchical state machines, each of which represents the behavior of some system component. Indeed, this perspective significantly simplifies the interpretation of a multi-model.

This section starts by proving a primer on LTL model checking via SMT-solving based on unrolling the transition relation for a single system component. It follows a description of how the different state charts are formally combined, which leads to a formal representation of multi-models.

3.1 Primer on Model Checking by Unrolling

The model checking implementation uses the standard SAT-based approach to bounded. 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:

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

This formula only describes how the system behaves, when different computations are performed, but it does not relate the behavior to the desired

system properties. Assuming that a formula φ describes the desired system property, the model checking problem amounts to checking whether the combined formula

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

is satisfiable, where $\neg\varphi$ describes all states that *do not satisfy the specification*. If this formula $\mathcal{M}$ is satisfiable, the model represents a counterexample trace which specifies a finite trace through the system leading to the erroneous state. Otherwise, no such trace exists and the desired system property is satisfied. It is well-known that the state chart semantics can be represented by unrolling their operational semantics [LP99], which is likewise true for φ, which we briefly discuss in what follows.

3.2 Unrolling LTL Specifications

LTL is a temporal logic that specifies properties over the future over paths, where the term *temporal* refers to a discrete notion of time. LTL does not in itself include the ability to reason about concrete, dense time, but rather considers timing as a sequence of computation steps. For instance, the logic itself provides the means to express that some property should hold after three computation steps, but not that it holds after three seconds, as opposed to more advanced logics such as timed computation tree logic (TCTL).

Specifications in LTL are built from a finite set of atomic propositions, Boolean operators and the following temporal modalities:

Globally ψ A property ψ shall hold *globally*, denoted $\mathsf{G}\psi$.

Finally ψ *Eventually* property ψ shall be satisfied, denoted $\mathsf{F}\psi$.

Next ψ In the *successor state*, ψ shall be satisfied, denoted $\mathsf{X}\psi$.

ψ_1 **Until** ψ_2 Property ψ_1 shall hold *until* ψ_2 holds, denoted $\psi_1 \mathsf{U} \psi_2$.

These operators can be combined and nested to express complex properties. All LTL formulae are implicitly universally quantified, which means that the properties have to hold for all paths, as opposed to CTL which allows to specify properties over different possible futures by considering the branching structure of the computations[1].

[1] At a first glance, CTL may thus appear to be more expressive than LTL. However, expressiveness of both logics is incomparable, and CTL has certain disadvantages when it comes to its application. See Sect. 2 for a discussion.

The semantics of LTL is specified over infinite paths, which are naturally specified as ω-words $\pi = s_0 s_1 \ldots$ over the alphabet 2^{AP} where AP denotes the set of atomic properties. Let $\pi^i = s_i s_{i+1} \ldots$ denote the suffix of π starting at position i. The satisfaction $\models$ of an ω-word π is then specified as:

- $\pi \models p$ for $p \in \mathsf{AP}$ if and only iff $p \in s_0$
- $\pi \models \neg\psi$ iff $\pi \not\models \psi$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$
- $\pi \models \mathsf{X}\ \psi$ iff $\pi^1 \models \psi$.
- $\pi \models \psi_1\ \mathsf{U}\ \psi_2$ iff there exists $i\mathbb{N}$ such that $\pi^i \models \psi$ and for all $0 \leq k \leq i$, $\pi^k \models \psi_1$.

The semantics of $\mathsf{G}\psi$ and $\mathsf{F}\psi$ follow directly from the following equivalences:

- $\mathsf{G}\ \psi \equiv \neg\mathsf{F}\ \neg\psi$
- $\mathsf{F}\ \psi \equiv \mathsf{true}\ \mathsf{U}\ \psi$

This semantics can naturally be unrolled until a fixed depth k, for example:

$$\begin{aligned} \mathsf{F}\ \psi^i &= \psi^i \vee \psi^{i+1} \vee \ldots \vee \psi^k \\ \mathsf{G}\ \psi^i &= \psi^i \wedge \psi^{i+1} \wedge \ldots \wedge \psi^k \\ \mathsf{G}\ \psi_1^i \wedge \mathsf{F}\ \psi_2^i &= (\psi_1^i \vee \psi_1^{i+1} \vee \ldots \vee \psi_1^k) \wedge (\psi_2^i \wedge \psi_2^{i+1} \wedge \ldots \wedge \psi_2^k) \end{aligned}$$

This approach allows the specification to be passed directly to an SMT solver so that $\mathcal{M}$ can be checked for satisfiability.

3.3 Model Checking for Multi-Models

The behavior of a single (discrete-event) system component depends on its definition of internal actions, its input variables and its internal state. Intuitively, the component resides in some internal state until inputs arrive. It then executes one or more actions, which may potentially alter the internal state and some changes become visible at the output interfaces. This simple understanding of a component's behavior directly leads to a straightforward representation of multi-models, which cannot interactive via shared states, but only through their interfaces.

We define a system component formally as a tuple $\mathsf{C} = (\mathsf{O}, \mathsf{S}, \mathsf{I}_\mathsf{I})$, where:

- O denotes the operational semantics of the system component,
- $\mathsf{S} = \{s_1, \ldots, s_n\}$ denotes the internal state of the system component,
- I_I denotes the set of input variables of the system component, and
- I_O denotes the set of output variables of the system component.

The most import part of the representation of multi-models is the handling of interface variables. Input variables usually can freely be assigned values by the model checker. This is not necessarily the case when the system components are combined, because the outputs of one component may now serve as the inputs of another. Output variables of all system components thus become part of the *internal system state*; the occurrences of these variables as inputs in the operational semantics of a component are likewise replaced.

Figure 4 sketches this approach for a system consisting of two components that model the conversion of temperatures from Fahrenheit to Celsius and a controlling unit, which can send certain control commands to an actuator. The inputs are highlighted in green color, and the outputs are given in orange. When analyzed on its own, each component has two inputs and one output. The temperature conversion component takes as input a voltage and a temperature value in Fahrenheit, and transforms it into Celsius. The heating controller takes as input a voltage and a temperature value in Fahrenheit, and issues a control command.

If these two components are combined, the shared data leads to different interface connections:

- The input *voltage* is shared among the system components, and thus is considered identical by both.
- The heating controller component only has one external input (voltage), and the output of the temperature conversion component becomes the input of the heating controller. The temperature value in Celsius can thus been seen as becoming part of the overall system-wide state, rather than an output.

This transformation dovetails with the specification of multi-models via SysML connection diagrams as performed in INTO-CPS. A particular reason for the combination of independent system components for model checking stems from the fact that abstractions of continuous-time components are integrated with native discrete-event components. The construction of a multi-

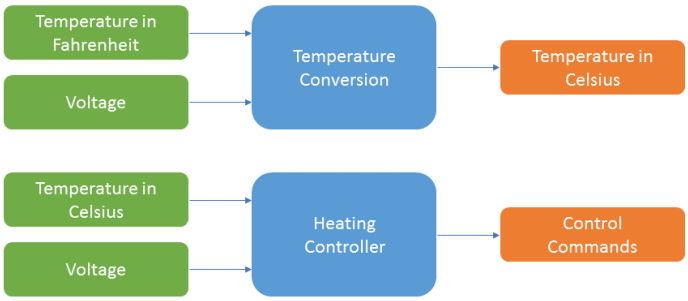

Figure 4: The system consists of two components for temperature conversion and heating. The output *temperature in Celsius* of the temperature conversion component is used as input for the *heating controller* component. The input *voltage* is shared and used by both components. Figure 5 shows the combined representation of these two components, where shared interface signals are represented via shared internal state variables.

model representation based on combining the constituent system components thereby provides a straightforward and configurable integration of the abstraction mechanism. For instance, it becomes trivial to replace an interval abstraction of a continuous-time component with a more precise one generated from simulations [BLM16].

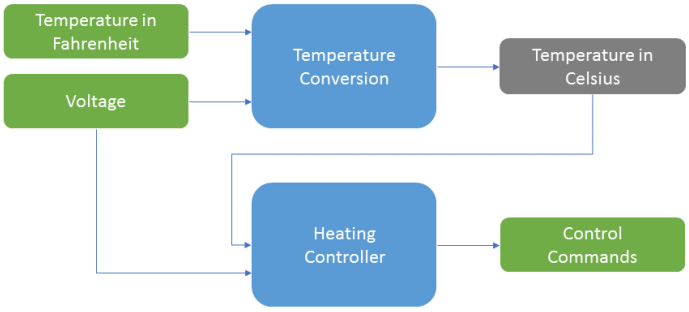

Figure 5: Two separate components, as given in Fig. 4 are combined in a single model. The key idea is to transform output signals of one component, which serve as inputs to another, as shared global variables, so that all changes immediately become visible to the dependent components. Since the modelling formalism of SysML state charts supports parallel composition, it is straightforward to combine two discrete-event models into one model containing two parallel components.

4 Implementation of a Model Checking Component in INTO-CPS

This section focuses on the architecture of the model checking component as implemented in the INTO-CPS project, and how it is connected to the traceability engine.

4.1 Architecture

The overall architecture of the system implementation is depicted in Fig. 6. The model checking component is configured via the INTO-CPS Application, which communicates with the model checking component via an abstraction layer implemented in Python. There is neither direct communication between the model checking component and the INTO-CPS Application, nor is the is the model checking component invoked directly. The rationale for this additional abstraction layer is that the implementation of the model checking algorithm itself is independent of the existence of a traceability implementation. Likewise, since the abstraction layer encapsulates all tool invocations, it is not necessary to alter the INTO-CPS Application in order to extend or modify the tracing.

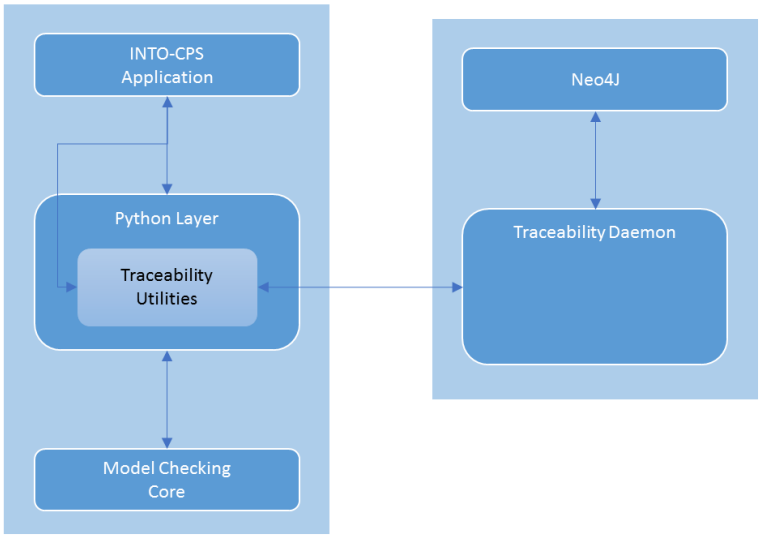

Figure 6: Integration of the model checking component in the INTO-CPS architecture.

4.2 Configuration of Model Checking

In principle, applying model checking to a multi-model requires three important configuration steps:

- The multi-model containing the different components has to be imported into the INTO-CPS Application.

- The abstractions to be applied to the multi-model have to be configured [2].
- The LTL specifications have to be defined.

Figure 7 shows the dialog to configure the desired abstractions. The example is based on the RT-Tester turn indication lever model, which models turn indication functionality based on three inputs (found under *Stimulation* in Fig. 7):

- A continuous input called `voltage` is provided, and the turn indication functionality is expected to operate only if the voltage is within the valid range from `9` to `13` [V].
- An integral input `TurnIndLever` is provided, which indicates the state of the physical turn indication lever. Admissible values range from `0` to `2`.
- A discrete input `EmerSwitch` defines whether emergency flashing is turned on, which may to some extent lead to interactions with the core turn indication functionality.

In this example, the input signal `voltage` of type `float` is to be fed using values from a concrete execution, the log of which is stored in a file called `signals.json`. The abstraction is configured so that values within an interval of 2 units are used for discretization of the signal.

The following listing shows an excerpt of a logfile, which simply denotes the values of all signals at different timestamps. Initially, `voltage` is zero and is set to `8.14` after 8231 ms. After 9000 ms, `voltage` becomes `12.041` and then slightly changes its value until 18016 ms have passed, when the execution ends. The input format is the standard RT-Tester logging format, which can thus be used to combine test executions using RT-Tester with model checking in the INTO-CPS Application. A screenshot from the signal viewer integrated into RT-Tester is given in Fig. 8. It is also straightforward to define execution logs manually. Overall, the simulation-based abstraction represents the signal flow using three intervals for `voltage`.

```
[ ...
  {"name":"voltage", "type":"float",
   "data": [[0,0,0.000000],
```

[2] Here, it is important to note that abstraction can not only be applied to continuous-time components, but also discrete-event components. Given a concrete execution for example, it may be desirable to verify system properties only for this execution; then, simulation-based abstraction could be applied to replace a discrete-event component.

```
                [1,8231,8.14],
                [2,9000,12.041],
                [3,10000,12.241],
                [4,18016,12.241]]},
...
]
```

It is not possible to verify system properties subject to the constraints specified through the execution log, such as: Within the first 9000 ms, the turn indication lever is off, no matter how other system inputs are set. This property is then formalized as:

$$\mathsf{G}((\texttt{\_timeTick} < 9000) \rightarrow ((\texttt{lightLeft} == 0) \wedge (\texttt{lightRight} == 0))$$

The configuration dialog for such queries is shown in Fig. 9.

For practical use, it is of course necessary to present counterexamples to the tool users. The provided functionality is given in Fig. 10. The model checker was configured so that the continuous input signal `voltage` is abstracted by a gradient, that is, it may change its value by `1.00` units in 1000 ms. The model checker was then invoked to verify the specification:

$$\mathsf{G}((\texttt{voltage} < 10 \vee \texttt{voltage} \geq 14) \rightarrow (\neg\texttt{LampsLeft} \wedge \neg\texttt{LampsRight})$$

The additional predicate `_stable` refers to a built-in predicate, which expresses that only stable states shall be considered. Stable states are those

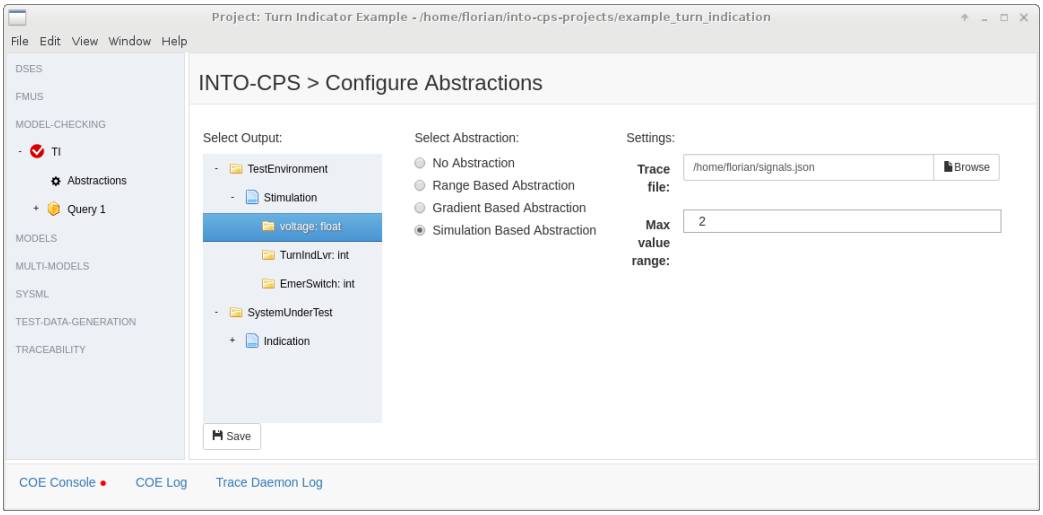

Figure 7: Configuration of simulation-based abstraction for signals.

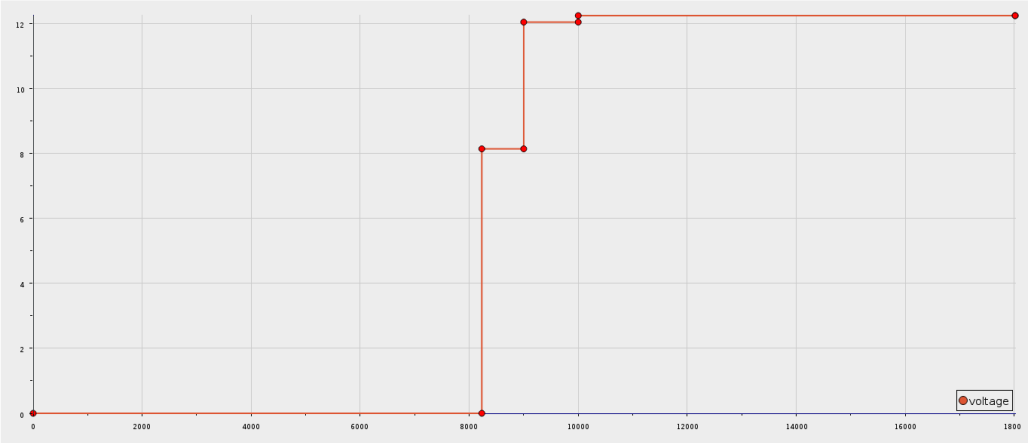

Figure 8: The values of `voltage` extracted from a concrete execution, which is then passed into the abstraction generator for model checking.

states where time passes, so that the specification allows the property to be violated internally. The counterexample view in Fig. 10 then shows a sequence of states and values, depicting how the specification is violated.

4.3 Traceability

The set of entities related to tracing of model checking activities and artefacts is depicted in Fig. 11. The actions in the INTO-CPS Application that lead to a recording of traceability related data are defined as:

ACT-CONFIGURE Configure a model for model checking.

ACT-ABSTRACT Define a discrete-event abstraction of a continuous-time component.

ACT-EXECUTE Run the model checker to verify a query.

Figure 11 intuitively sketches some of the relations among entities that are related to traceability. The syntax of traceability items for model checking is formally defined in [KLN+17]. Some examples of relations given in the diagram are:

- Model checking is always executed by some user (**ACT-EXECUTE**), indicated by an edge from activity *Model Checking* to an entity *Agent*.
- The action of executing the model checker (**ACT-EXECUTE**) produces a *Result*, which is connected to the *Agent* who has executed the

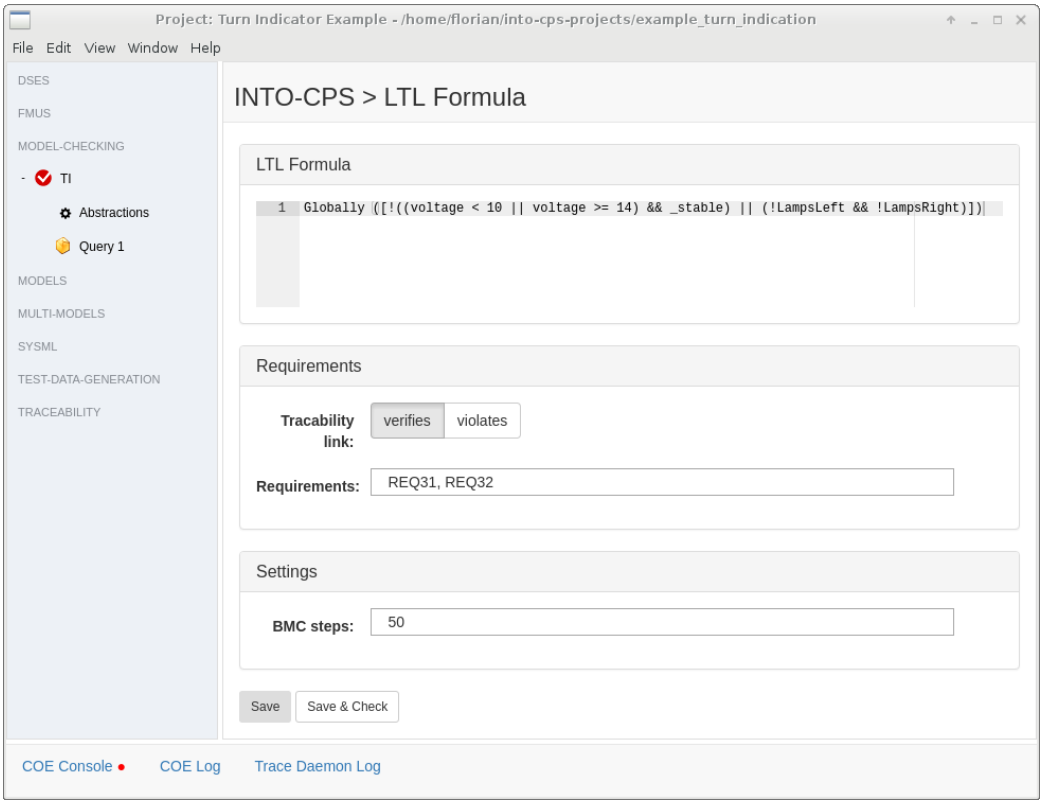

Figure 9: Editing of LTL formulae in the INTO-CPS Application. The screenshot also shows how requirements can be linked to model checking queries. A successful verification of a specification through the model checker can either confirm a requirement (*verifies*) or show that it is violated (*violates*), depending on the specification. After the model checker has terminated, the traceability data is automatically transferred to the traceability server. See Sect. 4.3 for further details.

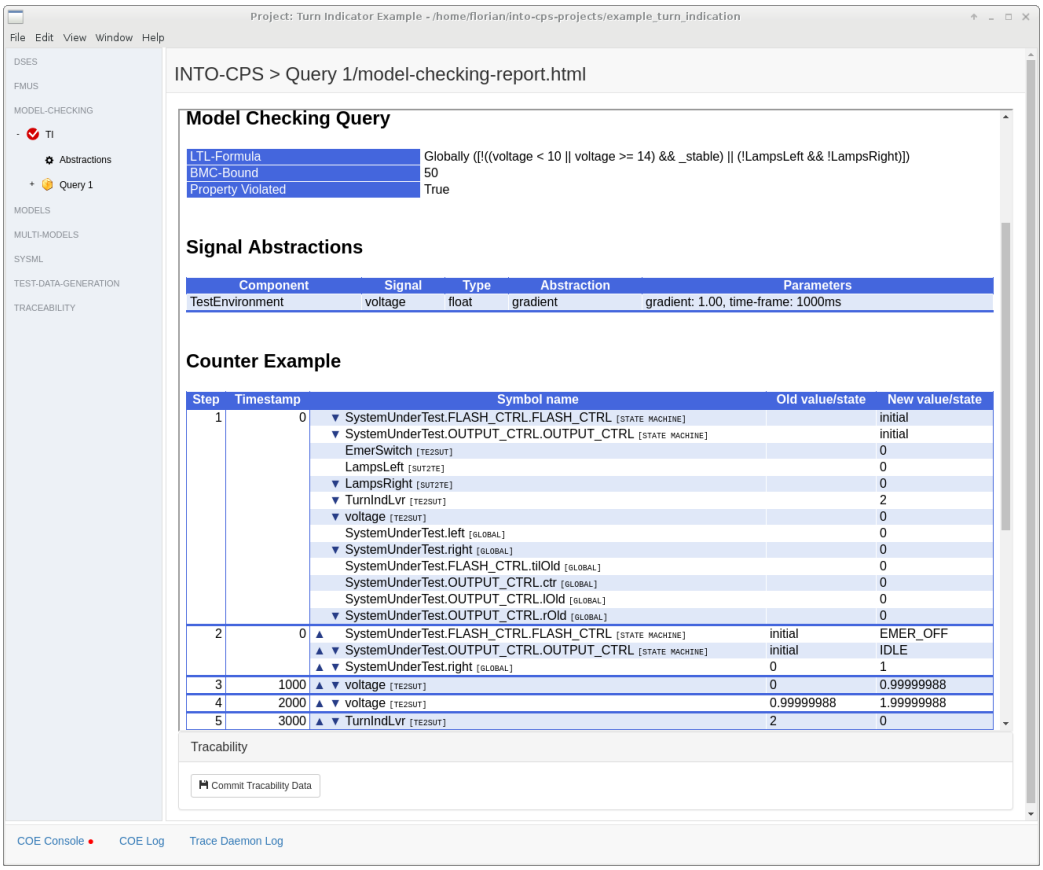

Figure 10: Presentation of counterexample traces in the INTO-CPS Application.

model checker. The result is linked to an arbitrary number of *Requirement* entities, which are either verified or shown to be violated.

No configuration is required at all because valid default settings are used. However, it is necessary to define the traceability OSLC server via environment variables:

- `OSLC_SERVER`: The machine is running the OSLC server, if undefined `localhost` is used.
- `OSLC_PORT`: The corresponding port number, if undefined `808` is used.
- `OSLC_EDOMAIN`: E-mail domain to be used.
- `OSLC_VERBOSE`: If to a non-zero value, then verbose output is generated, which shows all the internal JSON data before posts are generated.

For further details on the format of the traceability data, we refer to [KLN+17].

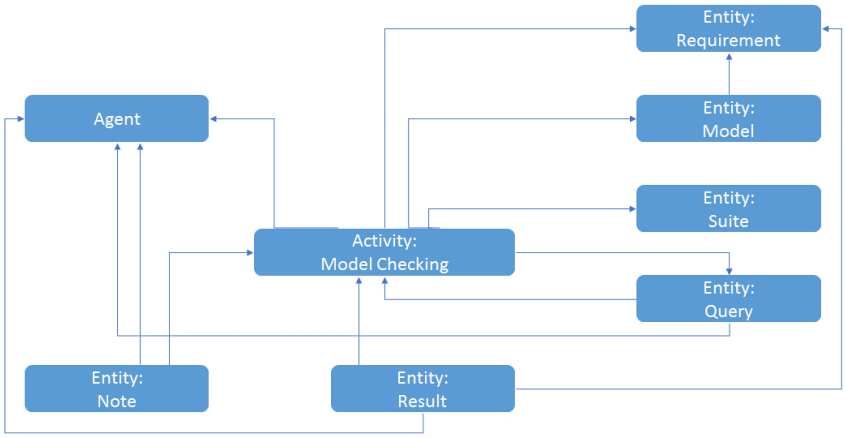

Figure 11: The diagram depicts the data and dependencies which are related to tracing of the activity *model checking*. The corresponding artefacts are automatically collected by the model checking component and transferred to the traceability engine.

5 Conclusion

The scope of this document is the implementation and application of state-of-the-art model checking techniques for cyber-physical systems, whose behavior is described by a network of heterogeneous components. The components are heterogeneous in the sense that they are specified using different formal models, which is not uncommon to cyber-physical systems.

The approach advocated in this deliverable is conceptually simple: Each system component has to be represented as a discrete-event component, which may require abstraction. The resulting network of discrete-event components is then combined and fed into a model checker for LTL. This strategy thereby allows to verify system-wide properties of the multi-model, whilst operating on interchangeable components.

A seemingly easy aspect whose practical aspect must not be underestimated is traceability, which is a core requirement in virtually any standard for the development of safety-critical systems, such as the RTCA DO-178B [WG-92] and RTCA DO-178C [WG-11], the CENELEC EN-50128 [CEN99] or the ISO 26262 [ISO11]. Traceability of development artefacts may be considered an essential milestone towards completeness of the development lifecycle. Only if artefacts from a development phases can safely be associated the underlying artefacts, such as system requirements, development can be considered complete. One contribution of this deliverable is that it provides a general scheme how model checking traceability can be implemented, and which entities may be necessary to thoroughly trace artefacts and actions.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[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.

[BLM16] Jörg Brauer, Florian Lapschies, and Oliver Möller. Implementation of a Model-Checking Component. Technical report, INTO-CPS Deliverable, D5.2b, December 2016.

[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.

[CEN99] Railway Applications: Systematic Allocation of Safety Integrity Requirements. Technical Report prR009-004, CENELEC, March 1999.

[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 Podelski, 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.

[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.

[ISO11] Road Vehicles - Functional Safety. Technical report, International Organization for Standardization, 2011. ICS 43.040.10.

[KLN+17] Christian König, Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Im-

plementation. Technical report, INTO-CPS Deliverable, D4.3d, December 2017.

[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.

[LNH+16] Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Design. Technical report, INTO-CPS Deliverable, D4.2d, December 2016.

[LP99] Johan Lilius and Iván Porres Paltor. *Formalising UML State Machines for Model Checking*, pages 430–444. Springer, 1999.

[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.

[WG-92] RTCA SC-167/EUROCAE WG-12. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178B, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 1992.

[WG-11] RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.

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
CTL	Computation Tree Logic
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