Automated Model-Based Testing with RT-Tester
Technical Report
2011-05-25

Jan Peleska, Elena Vorobev and Florian Lapschies
Universität Bremen
Department of Mathematics and Computer Science
28209 Bremen, Germany
{jp,elenav,florian}@informatik.uni-bremen.de

Cornelia Zahlten
Verified Systems International GmbH
28209 Bremen, Germany
cmz@verified.de

Abstract

In this technical report the theoretical foundations of the model-based testing tool RT-Tester are presented. RT-Tester is an industrial strength tool which automatically generates test cases and the associated test data from models specifying concurrent reactive systems in different concrete formalisms which may also involve clocks and dense time. RT-Tester is applied in industrial test campaigns for embedded systems testing in the automotive, railway and avionic domains. The paper is written from the tool builders' perspective and explains the typical components that should be present in model-based testing tools: it is described how various modeling formalisms can be supported by means of different parser front-ends transforming concrete models into a uniform internal abstract syntax representation. Test cases are derived automatically from this representation; additionally user-defined test objectives can be supplied using a subset of LTL specifications. The initial test case description is symbolic in the sense that only formal specifications of computations suitable for testing an objective have to be provided. Symbolic test cases represent constraint satisfaction problems (CSPs), and their solutions result in concrete test cases with explicit timed sequences of input vectors. An SMT solver is used for solving CSPs which is complemented by a simulator and techniques for model reduction and abstract interpretation, in order to speed up the solution process.

1. Introduction

Model-Based Testing. Automated *Model-based testing (MBT)* has received much attention in recent years, both in academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities, and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [15] as one example where quantitative evaluation results have been given. In this report the term model-based testing is used in the following, most comprehensive, sense: the behavior of the *system under test (SUT)* is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A *symbolic test case generator* analyzes the model and specifies *symbolic test cases* as logical formulae identifying model computations suitable for a certain test purpose. Constrained by the transition relations of SUT and environment model, a *solver* computes concrete model computations which are *witnesses* of the symbolic test cases. The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behavior observed during the test execution is compared against the *expected* SUT behavior specified in the original model. Both stimulation sequences and *test oracles*, i. e., checkers of SUT behavior, are automatically transformed into *test procedures* executing the concrete test cases in a model-in-the-loop, software-in-the-loop, or hardware-in-the-loop configuration.

Observe that this notion of MBT differs from "weaker" ones where MBT is just associated with some technique of graphical test case descriptions. According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modeling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically and debugging deviations of observed against expected behavior is considerably facilitated because the observed test executions can be "replayed" against the model. Moreover, V&V processes and certification are facilitated because test cases can be automatically traced against the model which in turn reflects the complete set of system requirements.

RT-Tester. In this paper the formal foundations of the model-based test case and test data generation component of the RT-Tester test automation tool are described. RT-Tester supports all test levels from module testing to system integration testing and provides different tool components for manual test procedure development, automated test case, test data and test procedure generation (this is the focus of this paper), as well as management functions for large test campaigns. The typical application scope covers (potentially safety-critical) embedded real-time systems involving concurrency, time constraints, discrete control decisions as well as integer and floating point data and calculations. While the tool has been used in industry for over 10 years and has been qualified for avionic systems under test according to the standard [25], the results presented here refer to new functionality that has been validated during the last 2 years in various projects from the transportation domains and are now made available to the public.

The presentation is structured according to the tool builders' perspective: we describe the ingredients that, according to our experience, should be present in industrial-strength test automation tools, in order to cope with test models of the sizes typically encountered when testing embedded real-time systems in the automotive, avionic or railway domains.

Tool Components and Their Interaction. Our starting point is a concrete *test model* describing the expected behaviour of the system under test (SUT) and, optionally, the behaviour of the operational environment to be simulated in test executions by the testing environment (TE) (see Fig. 1). The goal is to derive a set of model computations, that is, sequences $\langle \sigma_0, \sigma_1, \dots \rangle$ of time stamps, interface and internal state valuations σ_i that are concrete instances of test cases specifying which computations are suitable for testing various behavioural aspects of the SUT. Our concept of models also comprises computer programs, typically represented by per-function/method control flow graphs annotated by statements and conditional expressions.

It is our expectation that the ongoing discussions about suitable modelling formalisms for reactive systems – from UML via process algebras to synchronous languages – will not converge to a single preferred formalism in the near future. As a consequence it is important to separate the test case and test data generation algorithms from the concrete formalism. This is achieved for the RT-Tester tool as follows: (1) Models developed in a specific formalism are transformed into some textual representation supported by the CASE tool (usually XMI format). (2) A *parser front end* reads the model text and creates an *intermediate model representation (IMR)* of the abstract syntax. (3) A *transition relation generator* creates the initial state and transition relation of the model as first order logic predicates referring to pre-and post-states. (4) *Model transformers* create additional reduced, abstracted or equivalent model representations which are useful to speed up the test case and test data generation process. (5) A *constraint generator* creates first order formulas representing test cases built according to a given strategy. (6) A *satisfiability modulo theory (SMT) solver* calculates solutions of the test case constraints in compliance with the transition relation. This results in concrete computation fragments yielding the time stamps and inputs to the SUT to be used in the test procedure implementing the test case (and possibly other test cases as well). (7) An *interpreter* simulating the model in compliance with the transition relation is used to investigate concrete model executions continuing the computation fragments calculated by the SMT solver or, alternatively, creating new computations based on environment simulation and random data selection. (8) Finally, the *test procedure generator* creates executable test procedures as required by the test execution environment by mapping the computations derived before into time-controlled commands sending input data to the SUT and by creating test oracles from the SUT model portion checking SUT reactions on the fly, in dependency of the stimuli received before from the TE.

Related Work of the Authors. As of today, RT-Tester supports high-level modeling formalisms Timed CSP (Communicating Sequential Processes [26, 18]), UML 2.0 with composite structures, interfaces, classes and state machines extended by clocks (timers), interpreted according to Harel's Statecharts in the semantics presented in [10], Matlab-Simulink/Stateflow [16] and Timed Moore Automata [15]. These formalisms are currently applied in industrial projects. For testing on program code level research on testing C programs with full data type support (pointers, arrays, structures, unions, type casts) is performed [21, 22, 19]. The combination of SMT solving and abstract interpretation has been published in [23].

Related Work of Other Research Groups. Our UML 2.0-based modelling formalism follows closely Harel's Statecharts in the semantics presented in [10] with synchronous execution of enabled transitions in parallel components, but does not support the event concept of Statecharts and does not allow AND states. The timer concept used is that of timed automata [5, pp. 265].

The problem of deciding the satisfiability of logical (first order) formulas where propositions may be constraints of

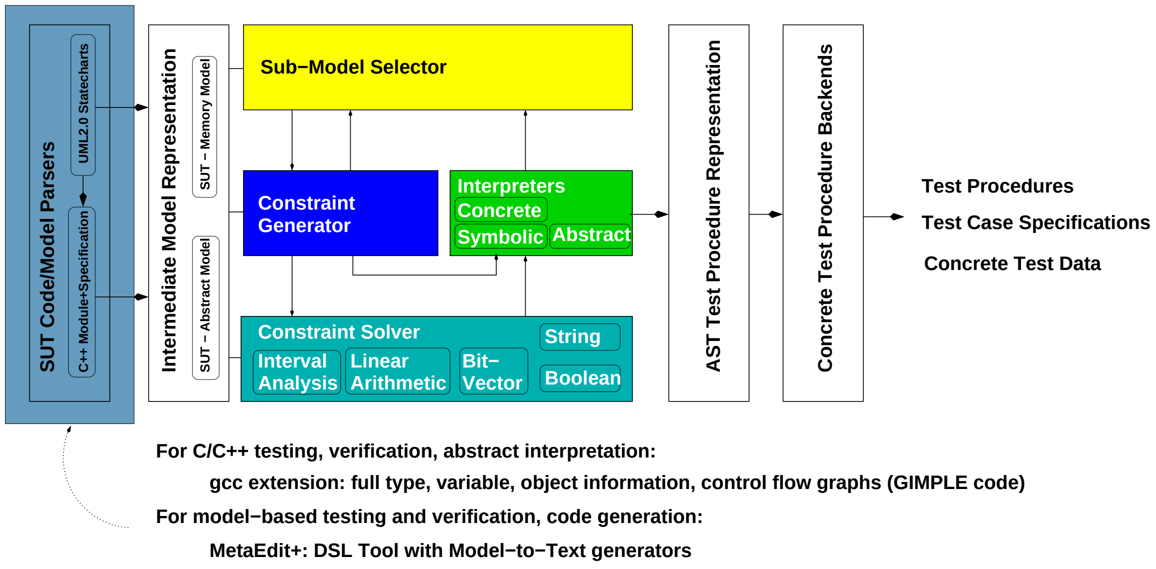
SUT Code/Model Parsers
UML2.0 Statecharts
C++ Module+Specification
Intermediate Model Representation
SUT – Memory Model
SUT – Abstract Model
Sub−Model Selector
Constraint Generator
Interpreters
Concrete
Symbolic
Abstract
Constraint Solver
Interval Analysis
Linear Arithmetic
Bit−Vector
String
Boolean
AST Test Procedure Representation
Concrete Test Procedure Backends
Test Procedures
Test Case Specifications
Concrete Test Data
For C/C++ testing, verification, abstract interpretation:
gcc extension: full type, variable, object information, control flow graphs (GIMPLE code)
For model−based testing and verification, code generation:
MetaEdit+: DSL Tool with Model−to−Text generators

Figure 1. Components of the RT-Tester test case/test data generator.

certain background theories is commonly referred to as the *Satisfiability Modulo Theories (SMT)* problem. SMT solvers have been developed for numerous theories and combinations thereof. In recent years SMT solvers have become important tools for software verification [24].

To find the right stimuli for covering new parts of our model we incorporate an SMT solver to solve constraint formulas that may contain linear and non-linear terms including modular integer arithmetic as well as bit-vector operations. Like most other state-of-the-art SMT solvers [13, 3, 14] solving these kind of formulas our SMT solver, SONOLAR, is based on the bit-blasting approach that translates an SMT formula to a purely propositional formula and lets an SAT solver decide the satisfiability. Various extensions to pure bit blasting have been proposed. [4] explores the generation of over- and under-approximations of bit vector formulas to speed up the decision procedure and [2] extends this concept to the domain of floating-point arithmetic. [30] does not bitblast the whole formula, instead so called modules are used for complex arithmetic operations to reason on the word-level. In addition, sophisticated word- and bit-level rewritings have been developed to simplify formulas [13].

Although our solver doesn't directly use approximations and modules it was ranked second in the division for solving closed quantifier-free formulas over fixed-size bitvectors (QF_BV) at the Satisfiability Modulo Theories Competition (SMT COMP 2010).

Our abstract interpretation approach is inspired by Cousot's work [7, 6] and uses facts from interval analysis [12] which have been abstracted to more general lattices in Section 5.

While our test generation approach relies on constraint solvers to find test-input-data, *search-based testing* techniques use randomized methods guided by optimization goals. In [1] the use of random testing, adaptive random testing and genetic algorithms for use in model-based black-box testing of real-time systems is investigated. To this end, the test-environment is modeled in UML/MARTE while the design of the SUT is not modeled at all, since all test data are derived from the possible environment behavior. An environment simulator is derived from the model that interacts with the SUT and provides the inputs selected by one of the strategies. The environment model also serves as a test oracle that reports errors as soon as unexpected reactions from the SUT are observed. This and similar approaches are easier to implement than the methods described in this paper, because there is no need to encode the transition relation of the model and to provide a constraint solver, since concrete test data is found by randomized model interpretations. We suspect, however, that the methods described in [1] do not scale up to handle industrial-scale systems, where the concurrent nature of the SUT requires to consider the time-dependent interaction between several components, and the construction of a single large product automaton from the many smaller ones describing component behavior is infeasible. To our best knowledge there is no work on using search based testing on synchronous parallel real-time systems in order to achieve a high degree of SUT coverage, let alone to find test input data to symbolic test-cases.

The solutions presented here have been implemented in the RT-Tester test automation tool which provides an alternative to TRON [17, 8] which supports timed automata test models and is also fit for industrial-strength application. RT-Tester also competes with the Conformiq Tool Suite [28], but focuses stronger on embedded systems testing with hard real-time constraints.

2. Abstract Syntax Representation and Transition Relation

Abstract Syntax. RT-Tester supports formalisms whose models consist of hierarchic components using shared interfaces, and where behavior is described by means of hierarchic state machines, together with an expression and action syntax supporting conditional statements and assignments. Any concrete modeling formalism consisting of these ingredients can be incorporated into the tool by adding a parser front end to transform concrete models into the abstract syntax representation described in the following paragraphs.

Model components $c \in C$ are arranged in hierarchic manner, so that a partial function $p_C : C \not\rightarrow C$ mapping each component but the root c_r to its parent is defined. The domain of the function is dom $p_C = C - \{c_r\}$. Each component may declare variables, and hierarchic scope rules are applied in name resolution. Interfaces between test environment and system under test as well as global model variables are declared on the level of c_r. All variables are typed. When parsing the model the scope rules are applied to all expressions and unique variable symbol names are used from then on. Therefore we can assume for the remainder of this section that all variable names are unique and taken from a symbol set V with pairwise disjoint subsets $I, O, T \subset V$ denoting TE $\rightarrow$ SUT inputs, SUT $\rightarrow$ TE outputs and timers, respectively. Function

$$d_C : C \rightarrow \mathbb{P}(C);\ d_C(c) \mapsto \{c' \in C \mid p_C(c') = c\}$$

defines the direct descendants of a component. Leaf components c satisfy $d_C(c) = \varnothing$. Each leaf is associated with a state machine $s \in SM$, where SM denotes the set of all state machines which are part of the model. Function $sc : C \not\rightarrow SM$; dom $sc = \{c \in C \mid d_C(c) = \varnothing\}$ associates component leaves with state machines.

State machines $s \in SM$ are composed of *locations* (also called *control states*) $\ell \in L(s)$ and *transitions*

$$\tau = (\ell, p, g, \alpha, \ell') \in \Sigma(s) \subseteq L(s) \times P \times G \times A \times L(s)$$

connecting source and target locations ℓ and ℓ', respectively. Value $p \in P = \mathbb{N}_0$ denotes the priority of the transition (0 is the best priority) and is used to enforce determinism for state machines specifying SUT behaviour. Transition component $g \in \text{Bexpr}(V)$ denotes the guard condition of τ which is a Boolean expression over symbols from V. For timer symbols $t \in T$ occurring in g we only allow Boolean conditions elapsed(t, c) with constants c. Intuitively speaking, elapsed(t, c) evaluates to `true` if at least c time units have passed since t's most recent reset.

Transition component $\alpha \in A = \mathbb{P}(V \times \text{Expr}(V))$ denotes a set of value assignments to variables in V, according to expressions from Expr(V). For a pair $a = (v, e) \in \alpha$, $var(a) =_{\text{def}} v$ and $expr(a) =_{\text{def}} e$ denote the projections on variable and expression, respectively. For timer symbols $t \in T$ only resets (t, reset) are allowed. A transition without accompanying assignments is associated with an empty set $\alpha = \varnothing$. Function

$$\omega_s : L(s) \rightarrow \mathbb{P}(\Sigma(s));\ \ell \mapsto \{(\bar{\ell}, p, g, \alpha, \ell') \in \Sigma(s) \mid \bar{\ell} = \ell\}$$

maps locations to their outgoing transitions. Locations are associated with (possibly empty) do actions, entry and exit actions; these are captured by the mappings

$$do_s : L(s) \rightarrow A,\ en_s : L(s) \rightarrow A,\ ex_s : L(s) \rightarrow A,$$

For the top-level location s of state machine s we require $en_s = \varnothing, ex_s = \varnothing$, but do-actions on the level of s are allowed.

Control states can be decomposed hierarchically as *OR-states*.The state machine s itself is identified with the top-level OR-state containing all other locations as subordinate control states. Function $p_s : (L(s) - \{s\}) \rightarrow L(s)$ maps lower-level control states to their parent locations. $p_s^n(\ell)$ denotes the n-fold application of p_s to ℓ, with $p_s^0(\ell) = \ell$.

$$d_s : L(s) \rightarrow \mathbb{P}(L(s));\ d_s(\ell) \mapsto \{\ell' \in L(s) - \{s\} \mid p_s(\ell') = \ell\}$$

defines the direct descendants of a location ℓ. Leaf locations are called *basic control states*; $BCS(s) =_{\text{def}} \{\ell \in L(s) \mid d_s(\ell) = \varnothing\}$ denotes the set of leaves belonging to state machine s. To identify a hierarchy of locations, we introduce a recursive set definition

$$\begin{aligned}&\zeta_s(\ell, \ell') =_{\text{def}} \{\ell\} \cup \\ &\qquad (\textbf{if } \ell = \ell' \vee \ell = s \textbf{ then } \varnothing \textbf{ else } \zeta_s(p_s(\ell), \ell') \textbf{ endif})\end{aligned}$$

and use notation $\ell..\ell' =_{\text{def}} \zeta_s(\ell, \ell')$.

On each control state decomposition into sub-ordinate locations exactly one *start location* has to be identified. Start locations are basic control states with no incoming and exactly one outgoing transition which is unguarded. Function start : $(L(s) - BCS(s)) \rightarrow L(s)$ maps higher-level control states to their direct descendants' start locations. The target control state of the transition leaving a start location is called the *initial location*.

Given two locations $\ell_1, \ell_2 \in L(s)$, the *least common ancestor* is the "closest" location containing both ℓ_1 and ℓ_2 as descendants. Function

$$\begin{aligned}&lca : L(s) \times L(s) \rightarrow L(s);\\ &(\ell_1, \ell_2) \mapsto p_s^n(\ell_1) \ \text{ where}\\ &n = \min\{m \in \mathbb{N}_0 \mid p_s^m(\ell_1) \in (\ell_1..s) \cap (\ell_2..s)\}\end{aligned}$$

defines this in a formal way.

Model State and State Transitions. Using the abstract syntax introduced above, consider a model $M = (C, SM, p_C, sc)$ with components $c \in C$ in hierarchic order specified by p_C whose leaves are associated with state machines $s \in SM = \{s_1, \dots, s_n\}$ as specified by function sc. The *state* of a model execution is specified by (1) the *active basic configuration*, that is, the vector $(\ell_1, \dots, \ell_n), \ell_i \in BCS(s_i)$ of basic locations where the state machines which are part of the model currently reside in, (2) the current valuation of all variables from V, and (3) the current time $\hat{t}$ of the execution. We consider basic control states as Boolean variables $\ell : \mathbb{B}$, with exactly one location per state machine s evaluating to `true`, indicating that s currently resides in this location. Following [5] we describe transition relations relating pre- and post-states by means of first order predicates over unprimed and primed symbols from $BCS \cup V \cup \{\hat{t}\}$, where $BCS =_{\text{def}} \bigcup_{s \in SM} BCS(s)$. The unprimed symbols refer to the symbol value in the pre-state, and the primed symbols to post-state values.

State Invariant. Invariant

$$\text{Inv} \equiv_{\text{def}} (\forall s \in SM : \text{XOR}_{\ell \in BCS(s)} \ell) \land (\forall v \in V : v \in D_v)$$

states that each state machine must be in exactly one basic control state during each step of a model execution. Moreover, all variables v assume values in their domain D_v which is usually expressed as a range of values from a given super-type. This invariant will be added to all transition specifications below.

Initial State. Each model execution starts in a state where (1) all inputs to SUT have arbitrary values within their defined range, (2) all other variables are initialised by their default values, and (3) all state machines s are in their topmost start locations whose parent is the state machines s itself. Formally,

$$\begin{aligned}&\text{Init} \equiv_{\text{def}} \text{Inv} \land \\ &\quad (\forall v \in V - I : v = \text{default}(v)) \land (\forall s \in SM : \text{start}(s))\end{aligned}$$

Effect of Value Assignments. Given a specification of value assignments $\alpha = \{a_1, \dots, a_k\} \in A$, the *effect* of α is characterised by predicate

$$\begin{aligned}&\epsilon(\alpha) \equiv_{\text{def}} \\ &\quad (\forall a \in \alpha \land var(a) \in V - T : var(a)' = expr(a)) \land \\ &\quad (\forall a \in \alpha \land var(a) \in T : var(a)' = \hat{t}\ )\end{aligned}$$

In predicate $\epsilon(\alpha)$ every variable symbol occurring on the left-hand side of an assignment in α gets a new post-state $var(a)'$. With the exception of timer variables, the new value is defined by the assignment expression $expr(a)$ evaluated in the pre-state. Timer variables $t \in T$ store the current value of the model execution time, so that the valuation of a guard condition elapsed(t, c) can be performed by evaluation of $\hat{t} - t \geq c$. If $\alpha = \varnothing$ then $\epsilon(\alpha) = $ `true` by construction. The *write set* of a value assignment α is defined as the set of variables that are written to by α, that is, $W(\alpha) =_{\text{def}} \{var(a) \mid a \in \alpha\}$.

If a variable v is addressed in more than one assignment, say, (v, e_1) and (v, e_2) and e_1, e_2 have different valuations in the pre-state, a *racing condition* occurrs which is reflected by the fact that $\epsilon(\alpha) = $ `false`, that is, the value assignments are infeasible. Therefore models with racing conditions are illegal.

Trigger Conditions for State Machine Transitions. Given state machine s, a transition $\tau = (\ell_0, p, g, \alpha, \ell_1) \in \Sigma(s)$ will be *triggered* if (1) its source location or one of its subordinate control states is part of the basic configuration currently active, (2) its guard condition evaluates to `true` in the current model state, (3) no higher-priority transition emanating from the same location is enabled and (4) no higher-level transition of the location under consideration is enabled. This is captured formally by

$$\begin{aligned}&\text{trigger}_s(\ell_0, p, g, \alpha, \ell_1) \equiv_{\text{def}} \\ &\quad (\exists \overline{\ell_0} \in BCS(s) : \overline{\ell_0} \land \ell_0 \in \overline{\ell_0}..s) \land \\ &\quad g \land (\forall (\ell_0, \overline{p}, \overline{g}, \overline{\alpha}, \overline{\ell_1}) \in \omega_s(\ell_0) : \overline{p} \geq p \lor \neg \overline{g}) \land \\ &\quad (\forall \overline{\ell} \in p_s(\ell_0)..s : \forall (\overline{\ell}, \overline{p}, \overline{g}, \overline{\alpha}, \overline{\ell_1}) \in \omega_s(\overline{\ell}) : \neg \overline{g})\end{aligned}$$

Effect of State Machine Transitions. Suppose transition $\tau = (\ell_0, p, g, \alpha, \ell_1)$ between state machine locations will be triggered. Assume further that $\overline{\ell_0}$ is the active basic control state equal to or subordinate to ℓ_0. Formally,

$$\overline{\ell_0} \in BCS(s) \land \overline{\ell_0} \land \ell_0 \in \overline{\ell_0}..s$$

The following predicate specifies the effect of τ's execution.

$$\begin{aligned}&\epsilon(\ell_0, p, g, \alpha, \ell_1) \equiv_{\text{def}} \\ &\quad (\forall \ell \in (\overline{\ell_0}..lca(\ell_0, \ell_1)) - lca(\ell_0, \ell_1) : \epsilon(ex_s(\ell))) \land \\ &\quad (\forall \ell \in lca(\ell_0, \ell_1)..s : \epsilon(do_s(\ell))) \land \epsilon(\alpha) \land \\ &\quad (\forall \ell \in (\ell_1..lca(\ell_0, \ell_1)) - lca(\ell_0, \ell_1) : \epsilon(en_s(\ell))) \land \\ &\quad ((d_s(\ell_1) = \varnothing \land \ell_1') \lor (d_s(\ell_1) \neq \varnothing \land \iota_s(\ell_1)))\end{aligned}$$

Informally speaking, the effect is as follows: (1) All exit actions from $\overline{\ell_0}$ up to, but excluding the least common ancestor (see definition of lca in Section 2) of ℓ_0 and ℓ_1 are executed. (2) All do actions of the locations starting with least common ancestor of ℓ_0 and ℓ_1 and ending at s are executed. (3) The action α associated with τ is executed. (4) The entry actions associated with ℓ_1 up to, but excluding the least common ancestor of ℓ_0 and ℓ_1 are executed. (5) If ℓ_1 is a basic control state then it becomes part of the new basic configuration, otherwise the initial locations of ℓ_1's subordinate control states are visited and associated actions are executed in the way specified by recursive predicate $\iota_s(\ell_1)$:

$$
\begin{array}{l}
\iota_s(\ell) \equiv_{\text{def}} \\
\quad \textbf{let } \ell_0 = \text{start}(\ell), (\ell_0, p_0, g_0, \alpha_0, \ell_1) \in \Sigma(s) \textbf{ in} \\
\qquad \epsilon(\alpha_0) \wedge \epsilon(en_s(\ell_1)) \wedge ((d_s(\ell_1) = \varnothing \wedge \ell_1') \vee \\
\qquad\qquad (d_s(\ell_1) \neq \varnothing \wedge \iota_s(\ell_1))) \\
\quad \textbf{endlet}
\end{array}
$$

The *write set* $W(\tau)$ associated with transition τ is derived from the effect $\epsilon(\tau)$ of the transition as follows:

$$
\begin{array}{l}
W(\tau) =_{\text{def}} (\bigcup_{\ell \in (\overline{\ell_0}..lca(\ell_0,\ell_1)) - lca(\ell_0,\ell_1)} W(ex_s(\ell))) \cup \\
\quad (\bigcup_{\ell \in lca(\ell_0,\ell_1)..s} W(do_s(\ell))) \cup W(\alpha) \cup W(en_s(\ell_1)) \cup \\
\quad (\textbf{if } d_s(\ell_1) = \varnothing \textbf{ then } \varnothing \textbf{ else } \nu_s(\ell_1) \textbf{ endif})
\end{array}
$$

with recursive set definition

$$
\begin{array}{l}
\nu_s(\ell) =_{\text{def}} \\
\quad \textbf{let } \ell_0 = \text{start}(\ell), (\ell_0, p_0, g_0, \alpha_0, \ell_1) \in \Sigma(s) \textbf{ in} \\
\qquad W(\alpha_0) \cup W(en_s(\ell_1)) \cup \\
\qquad\quad (\textbf{if } d_s(\ell_1) = \varnothing \textbf{ then } \varnothing \textbf{ else } \nu_s(\ell_1)) \\
\quad \textbf{endlet}
\end{array}
$$

Transition Relation. For generating test cases with associated test data, the behaviour of a model is encoded by means of a transition relation Φ associating pre-states of locations, variables and current time with post states. The transition relation distinguishes between *discrete transitions* Φ_D and *timed transitions* (also called *delay transitions*) Φ_T allowing the model execution time $\hat{t}$ to advance and inputs to change while the basic configuration, internal and output variables remain frozen. Discrete transitions take place whenever at least one state machine has an enabled transition or a do action needs to be executed. This condition is captured as predicate trigger$_D$. Timed transitions occur when $\neg$trigger$_D$ holds in the pre-state. After each transition the invariant shall still hold, that is, each state machine is in a well-defined basic control state and all inputs have values in their admissible range. These considerations induce the following structure for the transition relation:

$$
\Phi \equiv_{\text{def}} ((\text{trigger}_D \wedge \Phi_D) \vee (\neg\text{trigger}_D \wedge \Phi_T)) \wedge \text{Inv}'
$$

Predicate trigger$_D$ is defined as follows:

$$
\begin{array}{l}
\text{trigger}_D \equiv_{\text{def}} (\exists s \in SM, \tau \in \Sigma(s) : \text{trigger}_s(\tau)) \vee \\
\quad (\exists s \in SM, \ell_0 \in BCS(s), \ell \in \ell_0..s, \\
\qquad (v, e) \in do_s(\ell) : \ell_0 \wedge v \neq e)
\end{array}
$$

To understand the second term in this disjunction, consider a do action $\{(v_1, e_1), \ldots, (v_k, e_k)\}$ which is associated with active basic control state ℓ_0 or with one of ℓ_0's ancestors. This do action only leads to a discrete transition if at least one of the current values of left-hand side variables v_i differs from the current valuation of the associated right-hand side expression e_i, that is, if the precondition $v_i \neq e_i$ holds.

A variable v *is written to* during a discrete state transition if a state machine s taking part in the transition performs an action writing to v. This happens if either (1) s fires a transition accompanied by actions writing to v, or (2) from a certain level of the control state hierarchy on, no transitions on these levels can fire, but one of the do-actions executed in that case writes to v. Formally, this is expressed by

$$
\begin{array}{l}
\text{written}(v) \equiv_{\text{def}} \\
\quad (\exists s \in SM, \tau \in \Sigma(s) : \text{trigger}(\tau) \wedge v \in W(\tau)) \vee \\
\quad (\exists s \in SM, \ell_0 \in BCS(s), \ell_1 \in \ell_0..s : \\
\qquad \ell_0 \wedge (\forall \ell \in \ell_1..s, \tau \in \omega_s(\ell) : \neg\text{trigger}(\tau)) \wedge \\
\qquad\qquad v \in \bigcup_{\ell \in \ell_1..s} W(do_s(\ell)))
\end{array}
$$

Now, if a discrete transition is enabled its effects may be described as follows:

$$
\begin{array}{l}
\Phi_D \equiv_{\text{def}} (\hat{t}' = \hat{t}) \wedge (\forall v \in I : v' = v) \wedge \\
\quad (\forall s \in SM, \tau \in \Sigma(s) : \text{trigger}(\tau) \Rightarrow \epsilon(\tau)) \wedge \\
\qquad (\forall s \in SM, \ell_0 \in BCS(s) : \\
\qquad (\ell_0 \wedge \forall \ell \in \ell_0..s, \tau \in \omega_s(\ell) : \neg\text{trigger}(\tau)) \Rightarrow \\
\qquad\quad (\forall \ell \in \ell_0..s : \epsilon(do_s(\ell))))) \wedge \\
\qquad (\forall v \in V - I : \text{written}(v) \vee v' = v)
\end{array}
$$

(1) The current model execution time $\hat{t}$ remains unchanged. (2) All input variable values remain unchanged. (3) For every state machine possessing an enabled transition τ, the transition's effect as specified by $\epsilon(\tau)$ becomes visible in the post-state. (4) If none of the transitions emanating from the active basic control state ℓ_0 or any of its ancestor locations can fire, all do actions associated with any location in $\ell_0..s$ are executed. (5) All variables which are not in the write set of any executed transition or do action retain their old values (case $\neg$written(v)).

Delay transitions Φ_T are formally characterized as follows:

$$
\begin{array}{l}
\Phi_T \equiv_{\text{def}} (\hat{t}' > \hat{t}) \wedge (\forall s \in SM, \ell \in BCS(s) : \ell' \Leftrightarrow \ell) \wedge \\
\quad (\forall v \in V - I : v' = v) \wedge \\
\quad (\forall s \in SM, (\ell_0, p, g, \alpha, \ell_1) \in \Sigma(s) : \\
\qquad (\exists \overline{g} \in \text{Bexpr}, t \in T, c \in \mathbb{N} : g \equiv \overline{g} \wedge \text{elapsed}(t, c)) \Rightarrow \\
\qquad\qquad (\hat{t}' \leq c + t \vee \hat{t} \geq c + t))
\end{array}
$$

(1) The model execution time is advanced. (2) Inputs may change for the post-state of the delay transition, but all other variables and basic control states remain unchanged. (3) The admissible time shift is limited by the point in time when the next timer will elapse. More precisely, (a) whenever a timer t is still running (so elapsed(t, c) = `false` or, equivalently, $\hat{t} < c + t$) the time may advance at most as far as the point in time where t will elapse, that is, $c + t$. Equivalently, the new model execution time value $\hat{t}'$ shall satisfy

$\hat{t}' \leq c + t$. (b) Alternatively, t may have already elapsed before the delay transition is executed, that is, before current time $\hat{t}$. This is characterised by condition $\hat{t} \geq c + t$. In that case, timer t does not restrict the amount of time $\hat{t}'$ may be advanced.

3. Symbolic Test Cases and Concrete Test Data

In MBT test cases may be expressed as logical constraints identifying model computations which are suitable to investigate a given test objective. We use the term *symbolic test cases* for these constraints to emphasize that at this stage no concrete test data to stimulate a model computation satisfying them exists. As external representation of these constraints we use LTL formulas [5] of the type $\mathbf{F}\phi$, where the free variables in ϕ are model variables, basic state machine control states (interpreted as Booleans, `true` indicating that the machines currently resides in this location), and model execution time. The utilization of the finally operator $\mathbf{F}$ is motivated by the fact that to test a given objective, a computation prefix may have to be executed in order to reach a model state from where ϕ can be fulfilled. Typical model coverage criteria (see [29] for a comprehensive overview of these criteria) may easily be expressed as LTL formulas.

Example 1. To cover a state machine transition τ leaving a hierarchic control state ℓ, constraints can be expressed in the form $\mathbf{F}((\bigvee_i(\ell_i \wedge \psi_i)) \wedge \phi_1)$, where ϕ_1 denotes the guard condition of τ, each ℓ_i is a basic control state sub-ordinate to ℓ, so that the system resides in a basic control state from where τ can fire, and each ψ_i specifies the conditions such that – if the system resides in ℓ_i – no higher-priority transition will fire instead of τ. $\square$

More complex test cases involve formulas ϕ referring to control states of more than one component and/or using temporal operators.

Example 2. The following example refers to the model of turn indication functionality in vehicles which is available under [20]. In order to ensure that the turn indicator lever (in_TurnIndLvr) has priority over the additional turn indication and emergency flashing interfaces in_TurnIndLvrSPV, in_EmSwitchSPV available in special purpose vehicles, formula

$$\begin{aligned}\mathbf{F}(&\mathsf{pr\_Decision} = 4 \wedge \mathsf{lre\_FlashCmd} \in \{1,2\} \wedge \\ &\quad \mathsf{lres\_FlashCmd} = 0 \wedge \\ &\quad ((\mathsf{pr\_Decision} = 4 \wedge \mathsf{lre\_FlashCmd} \in \{1,2\})\ \mathbf{U} \\ &\quad (\mathsf{pr\_Decision} = 4 \wedge \mathsf{lre\_FlashCmd} \in \{1,2\} \wedge \\ &\qquad \mathsf{lres\_FlashCmd} > 0 \wedge \\ &\qquad\qquad \mathsf{lres\_FlashCmd} \neq \mathsf{lre\_FlashCmd})))\end{aligned}$$

represents a suitable test case: pr_Decision is an output of the priority handling function, and its value 4 indicates that left/right or emergency flashing have priority. Variable lre_FlashCmd is an output of the normal and emergency flashing function handling the standard interfaces, and its value 1 or 2 indicates that left or right flashing is active. Variable lres_FlashCmd is an output of the corresponding function handling the SPV interfaces. The formula specifies computations reaching a system state where left or right flashing is active when the SPV interfaces are still passive, and later – while left or right flashing is still performed – the SPV interface is brought into a contradictory state. $\square$

Since test cases need to be realized by finite model computation fragments, symbolic test cases are internally represented as so-called *bounded model checking instances*

$$tc(c, G) \quad \equiv_{\text{def}} \quad \bigwedge_{i=0}^{c-1} \Phi(\sigma_i, \sigma_{i+1}) \wedge G(\sigma_0, \ldots, \sigma_c) \quad (1)$$

In this formula σ_0 represents the current model state and Φ the transition relation, so any solution of 1 is a valid model computation fragment of length c. The test objective ϕ is encoded in $G(\sigma_0, \ldots, \sigma_c)$. For Example 1, $G(\sigma_0, \ldots, \sigma_c) = G(\sigma_c) = ((\bigvee_i(\ell_i(\sigma_c) \wedge \psi_i(\sigma_c))) \wedge \phi_1(\sigma_c))$. Intuitively speaking, $tc(c, G)$ tries to solve $\mathbf{F}\phi$ within c computation steps, starting in model pre-state σ_0.

To solve constraints of type 1 a solver is used which is described in the next section.

4. SMT Solver

Since our SMT solver follows the bit blasting approach, variables are encoded as fixed-width bit vectors, where the bit widths are given by the associated data types. Arithmetic and logical operations on these variables are transformed to Boolean constraints that encode the exact relationship of input and output bits. This allows us to have bit-precise results in the presence of modular arithmetic.

To this end the SMT formula is first transformed to a directed acyclic formula graph, where each single arithmetic and logical operation is represented as a single node. Structural hashing ensures that structurally identical terms are shared among expressions. On this formula graph a series of word-level simplifications like the evaluation of constant expressions, normalizations and term rewriting is performed. This word-level formula graph is then transformed to a bit-level, purely propositional *And-Inverter Graph (AIG)*. AIGs are commonly used among recent bit vector SMT solvers for synthesising propositional formulas [13, 3, 14]. AIGs represent propositional formulas as directed acyclic graphs (DAGs), where nodes are propositional variables or two-input AND-gates and edges may be optionally inverted. These AIG nodes are structurally hashed, too, and allow us to perform simplifications on bit level.

Although a number of competitive SAT solvers accept AIGs as input [27, 11], most SAT solvers require the input to be in CNF. To generate the CNF, for each node of the AIG a boolean variable is introduced. Each node with possibly inverted inputs $n \Leftrightarrow in_1 \wedge in_2$ is then translated to $(\neg n \vee in_1) \wedge (\neg n \vee in_2) \wedge (n \vee \neg in_1 \vee \neg in_2)$. For each root of the AIG an additional unit clause containing the associated variable asserts the corresponding boolean formula to be either true or false, respectively. See [9] for more information on logic synthesis using AIGs.

Many modern SAT solvers have the capability to be called incrementally. This technique allows us to add clauses between solver runs and to add unit clauses that are only valid for one run (so-called *assumptions*). The SAT solver can then re-use conflict clauses learned in previous runs to speed up the following ones.

As described in Section 3 the SMT solver needs to find a solution to the formula $tc \equiv (\bigwedge_{i=0}^{c-1} \Phi(\sigma_i, \sigma_{i+1})) \wedge G(\sigma_c)$, consisting of the c-fold unrolled transition relation and a disjunction of goals of version c. As the solver tries to find a solution with successively larger c, each try with the next larger c adds a transition constraint $\Phi(\sigma_c, \sigma_{c+1})$ to tc and replaces $G(\sigma_c)$ with $G(\sigma_{c+1})$. The additional transition constraint can simply be added to the SAT solver in an incremental fashion. However, with most SAT solvers it is not possible to remove all clauses making up $G(\sigma_c)$ before the next run. Therefore, the root of the AIG representing $G(\sigma_c)$ is added as an assumption when solving $\Phi(\sigma_{c-1}, \sigma_c)$ and is replaced by a new assumption enabling $G(\sigma_{c+1})$ when solving $\Phi(\sigma_c, \sigma_{c+1})$. The clauses of previous goals are still in the SAT solver's clause database, but since they are relatively few compared to the ones stemming from the transition relation, this does not lead to a noticeable slowdown.

5. Abstract Interpretation

5.1 Lattices and Galois Connections

Recall that a binary relation $\sqsubseteq$ on a set L is called a *(partial) order* if $\sqsubseteq$ is reflexive, transitive and anti-symmetric. An element $y \in L$ is called an *upper bound of* $X \subseteq L$ if $x \sqsubseteq y$ holds for all $x \in X$. The lower bound of a set is defined dually. An upper bound y' of X is called a *least upper bound of* X and denoted by $\sqcup X$ if $y' \sqsubseteq y$ holds for all upper bounds y of X. Dually, the *greatest lower bound* $\sqcap X$ of a set X is defined.

An ordered set $(L, \sqsubseteq)$ is called a *complete lattice*, if $\sqcap X$ and $\sqcup X$ exist for all subsets $X \subseteq L$. Lattice L has a *largest element* (or *top*) denoted by $\top =_{\text{def}} \sqcup L$ and a *smallest element* (or *bottom*) denoted by $\bot =_{\text{def}} \sqcap L$. Least upper bounds and greatest lower bounds induce binary operations $\sqcup, \sqcap : L \times L \to L$ by defining $x \sqcup y =_{\text{def}} \sqcup\{x, y\}$ (the *join* of x and y) and $x \sqcap y =_{\text{def}} \sqcap\{x, y\}$ (the *meet* of x and y), respectively. If the join and meet are well-defined for an ordered set $(L, \sqsubseteq)$ but $\sqcup X, \sqcap X$ do not exist for all $X \subseteq L$ then $(L, \sqsubseteq)$ is called an *(incomplete) lattice*.

Given any set M, the *power set lattice* $L_P(M)$ over M is defined by $L_P(M) = (\mathbb{P}(M), \subseteq)$ with meet $\cap$ and join $\cup$.

From the collection of canonic ways to construct new lattices from existing ones $(L, \sqsubseteq), (L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$, we need (1) cross products $(L_1 \times L_2, \sqsubseteq')$ where the partial order is defined by $(x_1, x_2) \sqsubseteq' (y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$ and (2) partial function spaces $(V \not\rightarrow L, \sqsubseteq')$ where $f \sqsubseteq' g$ for $f, g \in V \not\rightarrow L$ if and only if $\text{dom } f \subseteq \text{dom } g \wedge (\forall x \in \text{dom } f : f(x) \sqsubseteq g(x))$. Mappings $\phi : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ between ordered sets are called *monotone* if $x \sqsubseteq_1 y$ implies $\phi(x) \sqsubseteq_2 \phi(y)$ for all $x, y \in L$.

A *Galois connection (GC)* between lattices $(L_1, \sqsubseteq_1), (L_2, \sqsubseteq_2)$ is a tuple of mappings $\_^{\rhd} : (L_1, \sqsubseteq_1) \to (L_2, \sqsubseteq_2)$ (called *right*) and $\_^{\lhd} : (L_2, \sqsubseteq_2) \to (L_1, \sqsubseteq_1)$ (called *left*) such that $a^{\rhd} \sqsubseteq_2 b \Leftrightarrow a \sqsubseteq_1 b^{\lhd}$ for all $a \in L_1, b \in L_2$. This defining property implies that Galois connections are monotone in both directions.

Given data types $D_0, \ldots, D_n$, and abstracting lattices $L(D_i), i = 0, \ldots, n$ such that Galois connections $L_P(D_i) \underset{\rhd}{\overset{\lhd}{\leftrightarrows}} L(D_i)$ exist for all i, there is a natural way to lift n-ary functions $f : D_1 \times \ldots D_n \to D_0$ to operations on the abstracting lattices by setting

$$[f] : L(D_1) \times \ldots \times L(D_n) \to L(D_0);$$
$$(a_1, \ldots, a_n) \mapsto \{f(x_1, \ldots, x_n) \mid x_i \in {a_i}^{\lhd}, i = 1, \ldots, n\}^{\rhd}$$

Apart from the power set lattice introduced above we will utilise in the descriptions below a Boolean lattice $L(\mathbb{B}) = (\{\bot, 0, 1, \top\})$ with $\bot \sqsubseteq 0, 1 \sqsubseteq \top$ and $0, 1$ incomparable. This lattice allows to lift predicates regarded as Boolean functions $\phi(x_1, \ldots, x_n) \in \mathbb{B}$ with free variables $x_i \in D_i$ to three-valued logic predicates over abstracted variables $a_i \in L(D_i)$ using the above lifting procedure for n-ary functions: $[\phi](a_1, \ldots, a_n) \in L(\mathbb{B})$ evaluates to 1, if $\phi(x_1, \ldots, x_n)$ holds for all $(x_1, \ldots, x_n) \in {a_1}^{\lhd} \times \ldots \times {a_n}^{\lhd}$, to 0, if $\phi(x_1, \ldots, x_n)$ is always false and to $\top$ if $\phi(x_1, \ldots, x_n) = 1$ for some $(x_1, \ldots, x_n)$ and = 0 for others.

Finally we use interval lattices for abstracting integral and floating point variables: $L(\mathbb{Z}) = (\mathbb{IZ}, \subseteq)$ specifies the set of integer intervals with subset relation as partial order, meet defined by $\cap$ and join by the *interval hull* $[\underline{x}, \overline{x}] \sqcup [\underline{y}, \overline{y}] =_{\text{def}} [\min(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$. The bottom element is $\bot = \varnothing$, and top is $\top = [-\infty, \infty]$. For interval lattices the general lifting procedure specified above specialises to simple representation for arithmetic operations [12]; as, for example

$$[\underline{x}, \overline{x}][+][\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$[\underline{x}, \overline{x}][-][\underline{y}, \overline{y}] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$$

Interval lattices over floating point types are defined analogously. The problems of modular arithmetic are not considered in this paper, though our interval arithmetic library supports detection of over- and underflows. In the models referenced in Section **??**, however, these mechanisms are not relevant, because the units utilised for the respective variables ensure that over- and underflows cannot occur. Therefore the idealised lattice view on integral and floating point data types described here is appropriate for these models.

Consider a constraint satisfaction problem (CSP) $\phi(x_1, \ldots, x_n) = 1$ with boundary conditions $x_i \in X_i \subseteq D_i, i = 1, \ldots, n$, and denote its solution set by $\mathbb{S} \subseteq X_1 \times \ldots \times X_n$. If the CSP is not trivially solved by every $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$, then $[\phi]({X_1}^{\triangleright}, \ldots {X_n}^{\triangleright})$ will evaluate to $\top$. A *contractor* for this CSP is an operator $C(\phi; {X_1}^{\triangleright}, \ldots {X_n}^{\triangleright})$ returning a vector $a_1, \ldots, a_n$ of lattice elements such that $a_i \sqsubseteq {X_i}^{\triangleright}, i = 1, \ldots, n$ and $\mathbb{S} \subset {a_1}^{\triangleleft} \times \ldots \times {a_n}^{\triangleleft} \subseteq X_1 \times \ldots \times X_n$. This means that C yields a potentially better, that is, smaller approximation of the solution set $\mathbb{S}$ than the original Cartesian product $X_1 \times \ldots \times X_n$. For interval lattices we have natural contractors for arithmetic constraints, for example in $L(\mathbb{Z})$,

$$\begin{aligned}&C_<(x < y; [\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) =_{\text{def}} \\ &\qquad([\underline{x}, \min(\overline{x}, \overline{y} - 1)], [\max(\underline{x} + 1, \underline{y}), \overline{y}])\end{aligned}$$

Analogous contractors can be defined for atomic constraints involving $\leq, >, \geq$ and conjunctions or disjunctions thereof.

5.2 Abstract Interpretation Goals

We use abstract interpretation to investigate possible solutions of the CSPs specifying test cases as introduced in Section 3, that is, logical formulas

$$tc \equiv_{\text{def}} \bigwedge_{i=0}^{c-1} \Phi(\sigma_i, \sigma_{i+1}) \wedge G(\sigma_c)$$

where Φ denotes the model transition relation, σ_0 is the current system state from where the model exploration should start and c is an unknown integer which should be minimised. $G(\sigma_c)$ specifies the firing condition of the test case, or, more practically, the disjunction of all test case firing conditions still to be covered. The abstract interpretation has three main goals: (1) indicate lower bounds $c_0 > 0$ so that no solution of tc exists for $c < c_0$, (2) indicate necessary conditions $\phi(\sigma_0, \ldots, \sigma_c)$ to be fulfilled by every possible solution of tc and (3) execute significantly faster than the SMT solver, so that the execution time of the abstract interpretation plus that of the SMT solver operating with the knowledge $c \geq c_0 \wedge \phi(\sigma_0, \ldots, \sigma_c)$ is smaller than the execution time required by the SMT solver without this additional knowledge.

More formally, the SMT solver investigates sets of states

$$U_0 = \{\sigma_0\}, U_{i+1} = \{\sigma_{i+1} \mid \exists \sigma_i \in U_i : \Phi(\sigma_i, \sigma_{i+1})\}$$

and checks the goal $\exists \sigma_i \in U_i : G(\sigma_i), i = 1, 2, \ldots$. Observe that the U_i are elements of the power set lattice $L_P(S)$ over the concrete state space S. Since S is too large to be investigated in a speedy manner we define an abstraction of S as a cross product of lattices resulting in another lattice

$$\begin{aligned}&L(S) =_{\text{def}} L(D_{x_1}) \times \ldots \times L(D_{x_n}) \times \\ &L(D_{v_1}) \times \ldots \times L(D_{v_m}) \times \\ &L_P(Loc_{s_1}) \times \ldots \times L_P(Loc_{s_p}) \times \mathbb{IR}_+\end{aligned}$$

where $x_i \in I$ denote the input variables, $v_j \in L \cup O$ the internal model variables and outputs, $L_P(Loc_{s_q})$ denotes the power set lattice over basic control states of state machine s_q and $\mathbb{IR}_+$ is the interval lattice over non-negative reals. We assume that Galois connections are available for the data type abstractions $L_P(D_w) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} L(D_w)$. Then a GC $L_P(S) \underset{\triangleright}{\overset{\triangleleft}{\leftrightarrows}} L(S)$ is readily defined by setting for any subset $S_0 \subseteq S$

$$\begin{aligned}&{S_0}^{\triangleright} =_{\text{def}} (\{\sigma(x_1) \mid \sigma \in S_0\}^{\triangleright}, \ldots, \{\sigma(x_n) \mid \sigma \in S_0\}^{\triangleright}, \\ &\quad \{\sigma(v_1) \mid \sigma \in S_0\}^{\triangleright}, \ldots, \{\sigma(v_m) \mid \sigma \in S_0\}^{\triangleright}, \\ &\quad \{\ell_1 \in BCS(s_1) \mid \exists \sigma \in S_0 : \sigma(\ell_1)\}, \ldots, \\ &\qquad \{\ell_p \in BCS(s_p) \mid \exists \sigma \in S_0 : \sigma(\ell_p)\}, \\ &\quad [\min\{\sigma(\hat{t}) \mid \sigma \in S_0\}, \max\{\sigma(\hat{t}) \mid \sigma \in S_0\}])\end{aligned}$$

and for any $a \in L(S)$

$$\begin{aligned}&(a_1, \ldots, a_n, b_1, \ldots, b_m, e_1, \ldots, e_p, [\underline{t}, \overline{t}])^{\triangleleft} =_{\text{def}} \\ &\{\sigma \in S \mid \textstyle\bigwedge_{i=1}^{n} \sigma(x_i) \in {a_i}^{\triangleleft} \wedge \bigwedge_{i=1}^{m} \sigma(v_i) \in {b_i}^{\triangleleft} \wedge \\ &\quad (\exists (\ell_1, \ldots, \ell_p) \in e_1 \times \ldots \times e_p : \textstyle\bigwedge_{i=1}^{p} \sigma(\ell_i)) \wedge \\ &\quad \sigma(\hat{t}) \in [\underline{t}, \overline{t}] \wedge Inv(\sigma)\}\end{aligned}$$

To fulfil the objectives (1) and (2) defined above, the abstract interpretation algorithm specified in the next section starts on initial state $A_0 =_{\text{def}} {U_0}^{\triangleright} \in L(S)$ and computes elements $A_1, \ldots, A_r \in L(S)$ such that

$$\forall i \in 0 \ldots, r : 1 \sqsubseteq [\Phi](A_i, A_{i+1}) \wedge U_i \subseteq {A_i}^{\triangleleft}$$

Moreover, it returns $c_0 > 0$ such that $\forall i = 0, \ldots, c_0 - 1 : [G](A_i) = 0$. Since $U_i \subseteq {A_i}^{\triangleleft}$, $[G](A_i) = 0$ implies that no solution of G can be found in U_i. Therefore the transition relation has to be unrolled at least c_0 times by the SMT solver in order to find a solution of tc. Moreover, since every solution $\langle \sigma_0, \ldots, \sigma_{c_0}, \ldots, \sigma_r \rangle$ of tc satisfies $\sigma_i \in {A_i}^{\triangleleft}$, we can extract bounding information about possible locations and variable values in each σ_i and pass this on as necessary conditions to the solver. Finally, the abstract interpretation algorithm ensures that the A_i are computed very fast, and ${A_i}^{\triangleleft}$ can be derived with hardly any overhead from A_i; therefore the prerequisites for goal number (3) above are fulfilled.

5.3 Abstract Interpretation Algorithm

For the timed state machines introduced in Section 2 the abstract interpretation algorithm operates as specified in Fig. 2. Function exploreGoal() is invoked on the current concrete system state σ, so $\{\sigma\} = U_0$ in the notation introduced in the previous section, and the assignment $\sigma_A := \{\sigma\}^{\triangleright}$ creates $A_0 = \sigma_A$. In each loop cycle an abstract interpretation step is performed by means of procedure call absInt(σ_A, σ'_A), creating a new abstract state $A_i = \sigma'_A$.

Now the strongest necessary condition that can be derived from the fact that $U_i \subseteq A_i{}^{\triangleleft}$ is added as a conjunct to Boolean output expression β: At first, the possible basic control states where each state machine can reside in are added to β. After that the calculated variable limits applicable in the i^{th} transition are added as further restrictions.

Next it is checked whether there is a chance of solving the test case goal in step i. This is the case if the abstracted goal $[G]$ evaluates to 1 (then it is guaranteed that the goal will be met in step i) or to $\top$. This information is stored in Boolean output array r. The algorithm explores a maximum of c transitions emanating from σ. The number $c_0 > 0$ specified above and to be returned by the function is therefore the minimal index for which $r[i]$ is 1.

Note that it is not always useful to pass the maximal information about necessary conditions to the SMT solver, especially if some conditions are redundant to the information directly derivable from the transition relation. If, for example, the transition relation implies that some variable v does not change during a certain transition step (i.e.,$v_{i+1} = v_i$) then bounding information like $v_{i+1} \in [\underline{v}, \overline{v}]$ is redundant to the information already available. Therefore the current version of the algorithm only passes the information about possible locations in each transition step to the solver.

Fig. 5 shows the basic structure of the abstract interpreter: if the trigger condition for discrete transitions evaluates to 1 in the current abstract state σ_A then only an abstract interpretation of possible discrete transitions takes place. If $[\text{trigger}_D](\sigma_A)$ is guaranteed to be false, only a delay can occur. In that case, function absIntTime() (Fig. 4) calculates the boundaries of the new execution time stamp $\hat{t}$, and the abstractions of all input values x are set to their maximal ranges $D_x{}^{\triangleright} \in L(D_x)$. If $[\text{trigger}_D](\sigma_A)$ evaluates to $\top$, both discrete and delay transitions have to be taken into account and, consequently, the potential post-state is the maximum $\sigma^1_A \sqcup \sigma^2_A$ of the post-states resulting from these two transition types.

The abstract interpretation of a discrete transition is specified in Fig. 5. A a partial auxiliary function $\zeta : V \not\rightarrow \bigcup_{w \in V} L(D_w)$ is used for intermediate recordings of assignments to abstracted variables. For each basic control state ℓ_0 a state machine may potentially reside in, all emanating

```
**function** exploreGoal($\sigma : S,\ G : \text{BExpr},\ c : \mathbb{N}, \textbf{out}\ \beta : \text{BExpr}) : \mathbb{Z}$
**begin**
  $i := 1;\ \sigma_A := \{\sigma\}^{\triangleright};\ \beta := 1;\ r := -1;$
  **while** $i \le c$ **do**
    absInt($\sigma_A, \sigma'_A$);
    **foreach** $s \in SM$ **do** $\beta := \beta \wedge (\bigvee_{\ell \in \sigma'_A(\ell^s_A)} \ell_i)$; **enddo**
    $\beta := \beta \wedge \hat{t}_i \in \sigma'_A(\hat{t}) \wedge (\bigwedge_{x \in I} x_i \in \sigma'_A(x)) \wedge (\bigwedge_{v \in L \cup O} v_i \in \sigma'_A(v));$
    **if** $(1 \sqsubseteq [G](\sigma'_A))$ **then** $r := i$; **break**; **endif**
    $\sigma_A := \sigma'_A;\ i := i + 1;$
  **enddo**
  exploreGoal $:= r;$
**end**
```

Figure 2. Top-level procedure of the state space exploration by means of abstract interpretation. Sets I, L, O denote input, local and output variables, respectively.

```
**procedure** absInt($\sigma_A : L(S), \textbf{out}\ \sigma'_A : L(S)$)
**begin**
  **if** $[\text{trigger}_D](\sigma_A) = 1$ **then**
    absIntDisc($\sigma_A, \sigma'_A$);
  **elseif** $[\text{trigger}_D](\sigma_A) = 0$ **then**
    $\sigma'_A := \sigma_A \oplus \{\hat{t} \mapsto \text{absIntTime}(\sigma_A)\} \oplus \{x \mapsto D_x{}^{\triangleright} \mid x \in I\};$
  **else**
    absIntDisc($\sigma_A, \sigma^1_A$);
    $\sigma^2_A := \sigma_A \oplus \{\hat{t} \mapsto \text{absIntTime}(\sigma_A)\} \oplus \{x \mapsto D_x{}^{\triangleright} \mid x \in I\};$
    $\sigma'_A := \sigma^1_A \sqcup \sigma^2_A;$
  **endif**
**end**
```

Figure 3. Single step abstract interpreter.

```
**function** absIntTime($\sigma_A : L(S)) : \mathbb{IR}_+$
**begin**
  limit $:= \infty;$
  **foreach** $i \in \{1, \dots, p\}$ **do**
    smLimit $:= \underline{\sigma_A(\hat{t})};$
    **foreach** $\ell_0 \in \sigma_A(e_i)$ **do**
      locLimit $:= \infty;$
      **foreach** $\ell \in \ell_0..s_i,\ (\ell, g, a, \ell') \in \omega_{s_i}(\ell)$ **do**
        **if** $(\exists g', t, x : g \equiv (\hat{t} \ge x + t \wedge g')) \wedge [g'](\sigma_A) = 1$ **then**
          $m := \overline{\sigma_A(x)} + \overline{\sigma_A(t)};$
          **if** $m <$ locLimit **then** locLimit $:= m$; **endif**
        **endif**
      **enddo**
      **if** locLimit $>$ smLimit **then** smLimit $:=$ locLimit; **endif**
    **enddo**
    **if** smLimit $<$ limit **then** limit $:=$ smLimit; **endif**
  **enddo**
  absIntTime $:= [\underline{\sigma_A(\hat{t})} + \epsilon, \text{limit}];$
**end**
```

Figure 4. Function calculating the maximal time interval associated with a delay transition.

```
procedure absIntDisc($\sigma_A : L(S)$, out $\sigma'_A : L(S)$)
begin
  $\zeta := \varnothing$; $(q_1, \ldots, q_p) := (\varnothing, \ldots, \varnothing)$;
  foreach $i \in \{1, \ldots, p\}$ do
    foreach $\ell_0 \in \sigma_A(e_i)$ do
      leave := 0;
      foreach $\ell \in \ell_0..s_i$, $\tau \in \omega_{s_i}(\ell)$, $\tau$ ordered by priority do
        if $1 \sqsubseteq$ [trigger$_{s_i}(\tau)$]$(\sigma_A)$ then
          $\sigma^1_A := \sigma$;
          $C($trigger$_{s_i}(\tau), \sigma^1_A)$;
          absIntTransEffect($\sigma^1_A, \tau, \zeta, q_i$);
          if $1 =$ [trigger$_{s_i}(\tau)$]$(\sigma_A)$ then leave := 1; break; endif
        endif
      enddo
      if $\neg$leave then
        $\sigma^2_A := \sigma$;
        $C(\bigwedge_{\ell \in \ell_0..s_i,\ \tau \in \omega_{s_i}(\ell)} \neg$trigger$_{s_i}(\tau), \sigma^2_A)$;
        absIntDoEffect($\sigma_A, \ell_0, \zeta, q_i$);
      endif
    enddo
  enddo
  $\sigma'_A := \sigma_A \oplus \{e_i \mapsto q_i \mid i = 1, \ldots, p\} \oplus$
        $\{w \mapsto \zeta(w) \mid w \in$ dom $\zeta\}$;
end
```

Figure 5. Discrete transition abstract interpreter.

transitions from ℓ_0 and its higher-level locations are investigated. If a transition τ may fire, that is, if its abstracted trigger condition trigger$_{s_i}(\tau)$ evaluates to 1 or $\top$ in the pre-state σ_A, a copy σ^1_A of the pre-state is first contracted, using the knowledge that trigger$_{s_i}(\tau)$ must have evaluated to 1 in order to get the effect of τ.

This effect on the abstracted state space is calculated by procedure absIntTransEffect() which records these results by changing ζ: Suppose the effect of τ (see detailed definition of $\epsilon(\tau)$ in Section 2) comprises a value assignment $w :=$ expr;. If w is not yet in the domain of ζ, this means that it is the first potential write to w during this abstracted discrete transition. Therefore ζ's domain is extended by setting $\zeta := \zeta \oplus \{w \mapsto [\text{expr}](\sigma^1_A)\}$;, where [expr] is the lifted version of the assignment's right-hand side expression. The abstract expression evaluation is performed on the contracted abstract state σ^1_A. If w is already in dom ζ, this means that another transition might also write to w. In order to approximate the discrete transition effects in a conservative manner, we build the join of both potential effects, that is, we set $\zeta := \zeta \oplus \{w \mapsto \zeta(w) \sqcup [\text{expr}](\sigma^1_A)\}$;. Finally, absIntTransEffect() adds the target basic control state associated with τ to the set q_i of potential target locations.

If none of the transitions emanating from a location in $\ell_0..s_i$ is guaranteed to fire, that is, trigger$_{s_i}(\tau) = \top \vee 0$ for all of these τ and therefore leave $= 0$, the do actions associated with the locations in $\ell_0..s_i$ may be executed. Their effect on the abstract state space is calculated by absIntDoEffect() which works similar to absIntTransEffect(), but adds the source location ℓ_0 to q_i and operates on a copy of the source state contracted with the knowledge that all transition triggers must have evaluated to 0, in order to get the effect of these do-actions.

At the end of procedure absIntDisc(), the abstracted write effects are all joined for each affected variable w in $\zeta(w)$. Moreover, all basic control states which may potentially be visited by each state machine s_i as a result of this discrete transition are stored in the respective sets q_i, leading to the new location abstractions for each s_i in the new abstract valuation σ'_A. This join of potential write results and target locations ensures that all potential concrete target states contained in U_{i+1} are really contained in $\sigma'^{\lhd}_A$.

6. Conclusion

We have described 3 of the basic methods applied by the RT-Tester test case and test data generator, in order to construct concrete test data for symbolic test cases represented as constraint satisfaction problems: (1) the model behavior is encoded by means of a transition relation, (2) a SMT solver determines solutions of the CSP induced by symbolic test cases, and (3) the solution process is sped up by means of abstract interpretation. In [23] performance evaluations are discussed, in particular with respect to the acceleration gained from abstract interpretation. Further performance data will be published in [20].

The following methods will be considered in future revisions of this technical report:

- Non-chronological backtracking supported by abstract interpretation is used to find suitable model states from where complex test goals may be reached more easily, in order to speed up the test data generation process.

- For a given set of test goals the model may be reduced using cone-of-influence calculation techniques, so that only components affecting the reachability of the goals under consideration remain in the reduced model. This leads to smaller transition relations, which in turn speeds up the constraint solution process.

- For exploring the model state space random simulation techniques may be used in order to find more suitable model states from where the SMT solver may reach certain goals more easily.

References

[1] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG*

6.1 international conference on Testing software and systems, ICTSS'10, pages 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of FMCAD 2009*, pages 69–76. IEEE, 2009.

[3] R. Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University Linz, Austria, November 2009.

[4] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] P. Cousot. Abstract interpretation: Theory and practice. 11–13 April 2000.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[8] A. David, K. G. Larsen, S. Li, and B. Nielsen. Timed testing under partial observability. In *Proc. 2nd International Conference on Software Testing, Verification and Validation (ICST'09)*, pages 61–70. IEEE Computer Society, 2009.

[9] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In J. a. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*, chapter 26, pages 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[10] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[11] H. Jain and E. M. Clarke. Efficient SAT Solving for Non-Clausal Formulas using DPLL, Graphs, and Watched Cuts. In *46th Design Automation Conference (DAC)*, 2009.

[12] L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.

[13] S. Jha, R. Limaye, and S. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[14] J. Jung, A. Sülflow, R. Wille, and R. Drechsler. SWORD v1.0. Technical report, 2009. SMTCOMP 2009: System Description.

[15] H. Löding and J. Peleska. Timed moore automata: test data generation and model checking. In *Proc. 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE Computer Society, 2010.

[16] The Mathworks. *Simulink Product Description*. `http://www.mathworks.com`.

[17] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 5:59–77, 2003.

[18] J. Peleska. Applied formal methods - from CSP to executable hybrid specifications. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years. Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*, volume 3525 of *LNCS*, pages 293–320. Springer-Verlag GmbH, 2005.

[19] J. Peleska. A unified approach to abstract interpretation, formal verification and testing of c/c++ modules. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium*, volume 5160 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2008.

[20] J. Peleska, F. Lapschies, H. Löding, P. Smuda, H. Schmid, E. Vorobev, and C. Zahlten. Embedded systems testing benchmark, 2011. http://www.informatik.uni-bremen.de/agbs/testingbenchmarks.

[21] J. Peleska and H. Löding. Symbolic and abstract interpretation for c/c++ programs. In *Proceedings of the 3rd intl Workshop on Systems Software Verification (SSV08)*, Electronic Notes in Theoretical Computer Science. Elsevier, February 2008.

[22] J. Peleska, H. Löding, and T. Kotas. Test automation meets static analysis. In R. Koschke, K.-H. R. Otthein Herzog, and M. Ronthaler, editors, *Proceedings of the INFORMATIK 2007, Band 2, 24. - 27. September, Bremen (Germany)*, pages 280–286.

[23] J. Peleska, E. Vorobev, and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *Nasa Formal Methods, Third International Symposium, NFM 2011*, volume 6617 of *LNCS*, pages 298–312, Pasadena, CA, USA, April 2011. Springer.

[24] S. Ranise and C. Tinelli. Satisfiability modulo theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.

[25] RTCA,SC-167. *Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B*. RTCA, 1992.

[26] S. Schneider. *Concurrent and Real-time Systems – The CSP Approach*. Wiley and Sons Ltd., 2000.

[27] N. Sörensson. MiniSat 2.2 and MiniSat++ 1.1. Technical report, 2010. SAT-Race 2010: Solver Descriptions.

[28] C. T. Suite, 2010. http://www.conformiq.com.

[29] S. Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. Doctoral thesis, Humboldt-University Berlin, Germany, 2010.

[30] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like Prover Using Word Level Information. In *Proceedings of VLSI-SoC 2007*, pages 88–93, 2007.