Symbolic Test Case Generation for Time-Discrete Hybrid Systems

Jan Peleska*
University of Bremen, Germany
jp@tzi.de

Martin Fränzle†, Tino Teige†
University of Oldenburg, Germany
{fraenzle|teige}@informatik.uni-oldenburg.de

Abstract

In this article we present a model-based test case specification and associated test data generation methods for embedded systems processing Boolean, integral and real-valued variables. Testing experts are relieved from the task of constructing input data to the system under test in an explicit way and manually calculating the expected reactions. Instead, test cases are specified by means of temporal logic formulae using the Duration Calculus, allowing to describe classes of concrete runs which are considered equivalent for the test objectives to be investigated. The concrete input data to the system under test and its expected reactions are automatically generated using an approach based on interval analysis. Following this approach, test data generation is handled as an interval constraint solving problem. The basic solution technique based on pavings of the solution set and bi-partitioning algorithms are accelerated by using a tightly integrated combination of interval constraint propagation and a variety of novel techniques originating from Boolean SAT solving methods, which have been adapted for the mixed Boolean, integer and real-valued variable setting.

1 Introduction

Motivation. Following [22], a *test case* is a *set of test inputs, execution conditions, and expected results developed for a particular objective*. In practice, test case construction from specifications is often performed in two steps: First, the specification model is analyzed for a variety of paths through the model which might be suitable for checking the verification objective in mind. Second, (potentially timed) sequences of input data are elaborated in order to enforce the guard conditions required to cover one of these paths in a concrete run of the SUT. Following this observation from practical test case design, we introduce *symbolic test cases* as equivalence classes of runs, that is, bounded executions of the system under test (SUT), such that each run is equally well suited to investigate the verification objective. Equivalence classes are characterized by implicit specifications, restricting the set of all runs that are possible according to the SUT specification model by means of temporal logic formulae. We consider the construction of this formula, that is, the characteristics to be satisfied by a run in order to establish SUT compliance with a certain verification objective, as the critical task for the testing experts, whereas the concrete data construction should be automated as far as possible. The objective of this article is therefore to describe the complete chain of methods required, from modeling the SUT and elaborating symbolic test case specifications, to construction of concrete sequences of input data to the SUT and calculation of expected results.

Overview. The first part of this contribution deals with specification formalisms for the required SUT behavior and its associated test cases. In Section 2, the basic modeling technique for SUTs is introduced: We focus our investigation on *time-discrete input-output hybrid systems (TDIOHS)*; these are state-based reactive systems with shared variable I/O, operating on Boolean, integer and real variables. The class of time-discrete input-output hybrid systems is more restrictive than the *Hybrid Automata* described in [13]. However, TDIOHS induce a programming model which is widely used for embedded systems: Tasks are controlled by main loops starting with the transfer of input to internal state variables. After that, calculations are performed during a processing phase, using internal variables only. A main loop cycle is ended by copying updated values from internal to output variables. Time is regarded as an external resource which is updated

*Partly supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme SPP 1064 on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* (SPP 1064, HYBRIS, http://www.tzi.de/agbs/projects/hybris).

†Partly supported by the Deutsche Forschungsgemeinschaft DFG as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org).

during the input phase. In Section 3 we introduce a discrete variant of the *duration calculus (DC)* for specifying symbolic test cases. Duration is abstracted by the number of computation steps performed during bounded test executions. Compared to other variants of temporal logic, DC is distinguished by the possibility to express how "often" and how "long" certain properties hold during these executions. These capabilities are illustrated in Section 4.

The remainder of this article focuses on the techniques allowing to construct concrete test data and associated SUT outputs leading to runs which are admissible executions of the SUT model, at the same time satisfying the constraints imposed by the test case specification. Regardless of the external representation of logical constraints, such as used in transition guards of the SUT model or in test case specifications, a uniform internal representation for logical constraints is required. The associated syntax and its interpretation are described in Section 5. Given an SUT model $\mathcal{H}$, the possible runs of bounded length k can be encoded as a logical formula $\psi^k_{\mathcal{H}}$ over this syntax, as will be described in Section 6. Encoding the restrictions imposed by the test case in an analogous way yields a second logical formula $BMC(\phi, k)$ (Section 7), so that the concrete test data, leading to admissible runs of the model which are compliant with the symbolic test case specification, can be obtained as solutions of $\psi^k_{\mathcal{H}} \wedge BMC(\phi, k)$.

For the solution of logical formulae involving Boolean, integer and real-valued variables we advocate an approach based on interval analysis, so that $\psi^k_{\mathcal{H}} \wedge BMC(\phi, k)$ can be regarded as an interval constraint solving problem. The basic approach for solving these problems by constructing pavings of the solution set using bi-partitioning techniques, requires exponential time and is known to be too slow even for situations involving only small numbers of variables. However, the number of bi-partitioning steps can be considerably reduced by application of interval constraint propagation. Moreover, recent techniques for lazy clause evaluation, conflict-driven learning and non-chronological backtracking, originating from SAT solvers for Boolean problems can be adapted for our objectives. As described in Section 8, the tight integration of these algorithms result in a highly efficient machinery suitable to construct concrete test data for industrial-size problems.

We close with a discussion of the results achieved and references to related work in Section 9.

2 Time-Discrete Hybrid I/O Automata

A *time-discrete input-output hybrid system* (TDIOHS) is a tuple $\mathcal{H} = (Loc, Init, V, I, O, Trans)$ where Loc is a finite set of *locations* and V is a finite set of (discrete and continuous) variables. I, $O \subseteq V$ are sets of *input* and *output* variables, respectively, with $I \cap O = \emptyset$. $Init : Loc \to Guard$ is a function mapping locations to predicates describing the initial states, where $Guard$ denotes the quantifier-free predicates over V. $Assign$ is the set of all pairs $(\vec{x}, \vec{t})$, where $\vec{x} = (x_1, x_2, \ldots)$ is the vector of all variables in $V - I$ (a.k.a. *controlled* variables) and $\vec{t} = (t_1, t_2, \ldots) \in T^{|V-I|}$, in which T is the set of all arithmetic terms over variables V, involving the usual arithmetic operators including transcendental ones. $Trans \subseteq Loc \times Guard \times Assign \times Loc$ is the set of *transitions*. $Labels = \{\lambda \in Guard \times Assign \mid \exists l_1, l_2 \in Loc : (l_1, \lambda, l_2) \in Trans\}$ is the set of *transition labels*.

Let $val \in Val = \prod_{v \in V} \mathrm{dom}(v)$ be a *valuation of all variables occurring in* $\mathcal{H}$, and let the value of $v \in V$ and of the term t over V under val be denoted by $val(v)$ and $val(t)$, respectively. A *state* of the TDIOHS $\mathcal{H}$ is a pair $(l, v) \in Loc \times Val$. A *computation step* in $\mathcal{H}$ is a pair $((l_1, val_1), (l_2, val_2))$ of states such that $\exists (l, g, (\vec{x}, \vec{t}), l') \in Trans : l = l_1, l' = l_2,\ val_1(g) = \texttt{true} \wedge val_2(\vec{x}) = val_1(\vec{t})$. We write $(l_1, val_1) \longrightarrow (l_2, val_2)$ if there exists a computation step $((l_1, val_1), (l_2, val_2))$.

A *run* of a TDIOHS $\mathcal{H}$ is an infinite sequence $\langle (l_1, val_1), (l_2, val_2), \ldots \rangle$ of states which satisfies the properties *(1) Initiation:* $val_1(Init(l_1)) = \texttt{true}$ and *(2) Consecution:* $\forall i \in \mathbb{N} : (l_i, val_i) \longrightarrow (l_{i+1}, val_{i+1})$. A *$k$–bounded* run is a run of a fixed length $k \in \mathbb{N}$, i.e. the sequence of states $\langle p_1, \ldots, p_k \rangle$ is finite. The set of all k–bounded runs of $\mathcal{H}$ is denoted by $Run(\mathcal{H}, k)$.

The TDIOHS $\mathcal{H}$ is called *deterministic* if in every possible run $\langle (l_1, val_1), (l_2, val_2), \ldots \rangle$ of $\mathcal{H}$ only one transition is enabled at a time:

$$\forall i \in \mathbb{N}; \tau_1 = (l_i, g, a, l), \tau_2 = (l_i, g', a', l') \in Trans : \\ val_i(g) = \texttt{true} \wedge val_i(g') = \texttt{true} \Rightarrow \tau_1 = \tau_2$$

Let $V(\vec{t})$ denote the set of variables from V referenced in $\vec{t}$ and $Stable(\vec{x}, \vec{t})$ denote the variable components x_i of vector $\vec{x}$ whose values are unaffected by the associated assignment term t_i in any valuation (so $t_i \equiv x_i$). An *input location* $l_1 \in Loc_I \subseteq Loc$ is specified by the requirement that every transition entering l_1 only executes assignments where input variables are copied to *local* variables $x_i \in V - (I \cup O)$, that is,

$$Loc_I = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : \\ V(\vec{t}) \subseteq I \wedge O \subseteq Stable(\vec{x}, \vec{t})\ \}$$

An *output location* $l_1 \in Loc_O \subseteq L$ is characterized by the requirement that all transitions entering this state perform only assignments from local to output

variables:

$$Loc_O = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : V(\vec{t}) \subseteq V - (I \cup O) \subseteq Stable(\vec{x}, \vec{t})\ \}$$

An *internal processing location* $l_1 \in Loc_P \subseteq Loc$ is characterized by the requirement that entry assignments may only read from and write to local variables:

$$Loc_P = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : V(\vec{t}) \subseteq V - (I \cup O) \wedge O \subseteq Stable(\vec{x}, \vec{t})\ \}$$

A TDIOHS is called *I/O safe* if it possesses no other locations apart from input, processing and output locations, and the free variables of all guards are members of $V - I$. These conditions imply that input changes during processing steps are disregarded by the system: Only when a new input location is entered the new input valuations are copied to local variables, and are used by guards and assignment terms.

For the remainder of this paper we require all TDIOHS to be deterministic and I/O safe. For the purpose of test case specifications (see Section 4) it is useful to introduce auxiliary variables and associated assignments for each TDIOHS, so that these variables' valuations contain information about the parts of the TDIOHS which have been covered by the test case executions. To this end, we introduce two families of variables: For locations l, in_l evaluates to `true` as long as the TDIOHS resides in l. For a transition $\tau = (l_0, g, a, l_1)$ variable tgd_τ ("triggered τ") evaluates to *true* as soon as location l_1 is entered via τ and remains `true` until l_1 is left again. Formally speaking, V is partitioned into $V = V_b \cup V_a$ ("behavioural variables" V_b and "auxiliary variables" V_a), such that

$$\begin{array}{l} V_b \cap V_a = \emptyset \\ V_a = \{\ tgd_\tau \mid \tau \in Trans\ \} \cup \{\ in_l \mid l \in Loc\ \} \\ \forall l \in Loc; \tau \in Trans : \mathrm{dom}(tgd_\tau) = \mathrm{dom}(in_l) = \mathbb{B} \\ I \cup O \subset V_b \\ \forall g \in Guard : V(g) \subset V_b \\ \forall l_1 \in Loc, val \in Val : val(Init(l_1)) = \texttt{true} \Rightarrow \\ \quad (val(in_{l_1}) = \texttt{true} \wedge \\ \quad\ (\forall l \in Loc - \{\ l_1\ \} : val(in_l) = \texttt{false}) \wedge \\ \quad\ (\forall \tau \in Trans : val(tgd_\tau) = \texttt{false})) \end{array}$$

$$\begin{array}{l} \forall \tau = (l_1, g, a, l_2) \in Trans; val_1, val_2 \in Val : \\ \qquad (l_1, val_1) \longrightarrow (l_2, val_2) \wedge val_1(g) = \texttt{true} \Rightarrow \\ \quad ((\forall l \in Loc : val_2(l) \Leftrightarrow (l = l_2)) \wedge \\ \quad\ (\forall \tau' \in Trans : val_2(tgd_{\tau'}) \Leftrightarrow (\tau' = \tau)) \end{array}$$

Observe that these definitions are only meaningful if the TDIOHS is deterministic. In particular we require that there exists only one initial location l_1 and valuation *val* with $val(Init(l_1)) = \texttt{true}$.

Example 1. Fig. 1 shows an example of a TDIOHS $\mathcal{H}_1$ operating on variables from $V = I \cup O \cup L \cup V_a$ with input variables $I = \{a_{in}, b_{in}\}$, output variables $O = \{y, fail\}$, and local variables $L = \{a, b, c, n, t\}$. Variable *fail* is Boolean-valued, n, t are integers and the other variables are real-valued. The basic task of $\mathcal{H}_1$ is to monitor inputs a_{in}, b_{in} and transform them into outputs $y = \phi(a, b)$. To this end, a stability condition on inputs, $\beta(a, b, t)$, has to be observed: Outputs $y = \phi(a, b)$ may only be written after $\beta(a, b, t)$ held stably for 3 input cycles. If inputs a_{in}, b_{in} are so unstable that more than 10 changes between $\beta(a, b, t)$ and $\neg\beta(a, b, t)$ occur before a stable phase of 3 time units has been reached, $\mathcal{H}_1$ terminates (location l_7) and sets its failure output *fail* to `true`. After $\beta(a, b, t)$ has become stable, the output $y = \phi(a, b)$ is updated as long as $\beta(a, b, t)$ holds (locations l_3, l_4, l_5), after that y remains unchanged until $\beta(a, b, t)$ holds stably once more.

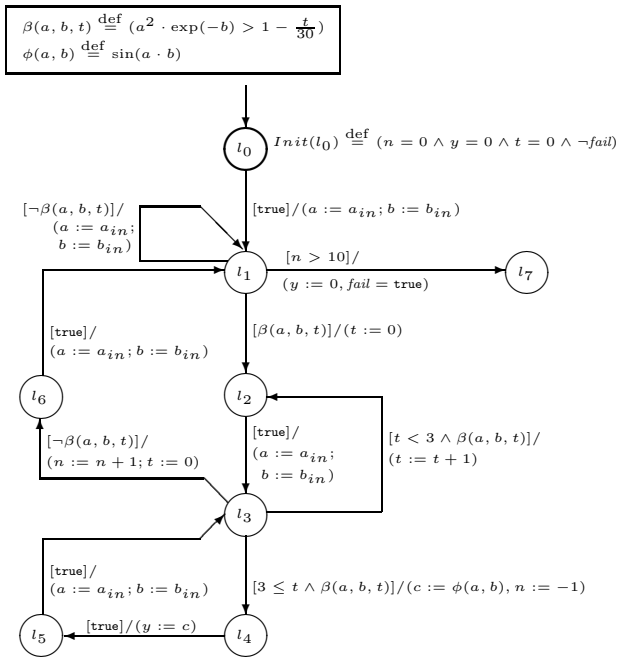

Figure 1: TDIOHS $\mathcal{H}_1$ of Example 1.

Following the definitions above, $\mathcal{H}_1$ has input locations $Loc_I = \{l_1, l_3\}$, output locations $Loc_O = \{l_5, l_7\}$ and internal processing locations $Loc_P = \{l_2, l_4, l_6\}$. We assume that l_0 is the only admissible initial location, so $Init(l) = \texttt{false}$ for $l \neq l_0$. To ease readability, guard conditions are written in square brackets and assignments $(\vec{x}, \vec{t})$ are written in programming style $x_i = t_i, x_j = t_j, \ldots$, omitting all variables from $Stable(\vec{x}, \vec{t})$. The auxiliary variables from set V_a and the operations thereon are not explicitly shown in Fig. 1, but introduced and manipulated according to the rules defined above. □

3 Duration Calculus

As will be explained in the next section, abstract test cases are equivalence classes of k-bounded TDIOHS runs. To specify these classes, an assertion technique is required, so that membership of a run r to an equivalence class is characterized by r's compliance with the assertion. Candidates for specifying these types of assertions are *trace logic* [25], *linear time temporal logic (LTL)* [17] or the *duration calculus (DC)* [24] advocated by the authors for this purpose. The particular suitability of DC will be motivated in Section 4; the present section briefly introduces DC syntax and semantics. Observe that several versions of DC semantics exist, each of particular suitability in its own application domain. Apart from differing expressive power in the admissible formulae, the most notable distinction consists in the notion of time: The original DC versions [24] were based on dense-time models, in [9] the advantages of discrete-time models are elaborated, and [21] does introduce a variant featuring both dedicated operators for counting computation steps and for metric time. As computation steps or sampling rounds are natural entities for test case generation, where one stimulus per sampling point has to be generated, we will employ a version of DC where temporal operators refer to computation steps.

Given a TDIOHS $\mathcal{H} = (Loc, Init, V, I, O, Trans)$, the syntax of a DC formula ϕ is specified by

$$\begin{array}{rcl} \phi & ::= & \Sigma\langle state\, predicate\rangle\langle rel\rangle\langle nat.\, number\rangle \mid \\ & & \neg\phi \mid (\phi \wedge \phi) \mid (\phi \frown \phi) \\ \langle rel\rangle & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \end{array}$$

where the *state predicates* are arbitrary quantifier-free predicates over variables from V, built from arithmetic expressions over the variables of $\mathcal{H}$ (involving basic operations $+, -, \cdot, /, \sin, \exp, \ldots$) by means of relations $=, <, \leq, >, \geq, \neq$ and Boolean connectives. In particular, all guard expressions of a TDIOHS are members of *Predicates*, but *Predicates* also contains expressions involving auxiliary variables from V_a, which are not admissible in guard expressions.

As semantic models to be associated with DC formulae we use k-bounded runs of the TDIOHS $\mathcal{H}$. Let DC denote the set of all syntactically valid DC formulae. The interpretation $[\![\ ]\!]_r : DC \to \mathbb{B}$ with respect to run $r \in Run(\mathcal{H}, k)$, $r = \langle (l_1, val_1), \ldots, (l_k, val_k)\rangle$ is inductively defined by

$$\begin{array}{rcl} [\![\Sigma P\ \omega\ m]\!]_r & \stackrel{\text{def}}{=} & |\{i \in \{1,\ldots,k\} \mid val_i(P)\}|\ \omega\ m\ , \\ [\![\neg\phi]\!]_r & \stackrel{\text{def}}{=} & \neg[\![\phi]\!]_r\ , \quad [\![\phi \wedge \psi]\!]_r \stackrel{\text{def}}{=} [\![\phi]\!]_r \wedge [\![\psi]\!]_r\ , \\ [\![\phi \frown \psi]\!]_r & \stackrel{\text{def}}{=} & \exists m \in \{0,\ldots,k\} : \\ & & \qquad [\![\phi]\!]_{\langle (l_1,val_1),\ldots,(l_m,val_m)\rangle} \wedge \\ & & \qquad [\![\psi]\!]_{\langle (l_{m+1},val_{m+1}),\ldots,(l_k,val_k)\rangle}\ , \end{array}$$

where $\omega \in \{=, <, \leq, >, \geq, \neq\}$ and $m \in \mathbb{N}$. Intuitively speaking, $\Sigma P \geq m$ holds if the valuation of P is `true` in at least m computation steps of the run r. The *chop operator* $\frown$ allows assertions of the type $\phi \frown \psi$, meaning that run r can be split into an initial part satisfying ϕ, followed by a part satisfying ψ.

With the interpretation $[\![\ ]\!]_r$ as above, we can introduce additional operators for DC formulae as syntactic abbreviations:

$$\begin{array}{rcl} \Diamond\phi & \stackrel{\text{def}}{=} & (\texttt{true} \frown \phi \frown \texttt{true}) \\ \Box\phi & \stackrel{\text{def}}{=} & \neg\Diamond\neg\phi \\ (\eta\ \omega\ m) & \stackrel{\text{def}}{=} & (\Sigma\texttt{true}\ \omega\ m) \\ (\Sigma P = \eta) & \stackrel{\text{def}}{=} & (\Sigma(\neg P) = 0) \\ \lceil P \rceil & \stackrel{\text{def}}{=} & (\Sigma P = \eta \wedge \eta > 0) \end{array}$$

Note that $\Diamond\phi$ actually means "in some subsegment of the run, ϕ holds", i.e. ranges only over the bounded horizon of the finite run under inspection. Likewise, $\Box\phi$ means "in all subsegments of the run, ϕ holds". η denotes the length of the run in computation steps, and $\lceil P \rceil$ expresses that predicate P invariably holds throughout the run, which has nontrivial length.

4 Symbolic Test Cases

A *symbolic test case* $TC(\phi)$ for a given TDIOHS $\mathcal{H}$ is an equivalence class of bounded runs specified by a DC formula ϕ. Formally speaking,

$$TC(\phi) = \{r \in Run(\mathcal{H}, k) \mid k \in \mathbb{N} \wedge [\![\phi]\!]_r\}$$

The term *symbolic* is used since the concrete test data (i. e., sequences of inputs to the system under test and expected output data from the SUT) is not determined, as long as the test case is only specified by ϕ. If suitable valuations can be found so that at least one bounded run of $\mathcal{H}$ lets ϕ evaluate to `true` then $TC(\phi)$ is *feasible*, otherwise *infeasible*. A run $r \in TC(\phi)$ is called a *concrete* test case. Sequence $\langle (val_i|I) \mid i = 1,\ldots,k\rangle$ is called the *test (input) data* and sequence $\langle (val_i|O) \mid i = 1,\ldots,k\rangle$ is called the *expected result*. In these expressions, $(f|X)$ denotes function domain restriction to elements from X. If the DC formula ϕ only refers to auxiliary variables from V_a then the test case is called *abstract*: The DC formula then only refers to locations and transitions but never to valuations of guards and assignments.

Example 1 (continued): Symbolic Test Cases. This example shows how typical test case definitions can be performed by only referring to the locations

and transitions to be covered during test executions.

(1) Maximal bounds of test execution runs. A standard test strategy consists in exploring the SUT only up to a certain depth k, meaning that test executions should not have more than k computations steps. Using the DC symbol η, this is expressed as $C_{bnd} \stackrel{\text{def}}{=} (\eta \leq k)$.

(2) Normal and exceptional behavior test cases. One of the basic strategies of test case design is the distinction between tests exercising the SUT under normal and under exceptional conditions. For $\mathcal{H}_1$ introduced in Section 2, exceptional behavior consists in unstable inputs (i. e., changes between $\beta(a,b,t)$ and $\neg\beta(a,b,t)$ over a period of more than 10 cycles. This always leads to termination in location l_7. As a consequence, exceptional behavior test cases can be characterized by condition $C_{ex} \stackrel{\text{def}}{=} \Sigma in_{l_7} \geq 1$, while normal behavior fulfills assertion $C_{norm} \stackrel{\text{def}}{=} \Sigma in_{l_7} = 0$.

(3) Decision coverage. For structural testing, test cases specify that certain regions of the specification model or the software code should be covered by test executions. The decision coverage goal, for example, requires that each transition of the specification model (or each branch in the control flow graph of the code) should be exercised at least once during a test suite. A test case for covering some transition τ of $\mathcal{H}_1$ is characterized by formula $C_\tau \stackrel{\text{def}}{=} \Sigma tgd_\tau \geq 1$. Test case generation for decision or multiple condition/decision coverage has been discussed in more detail in [2].

(4) Optimally stable behavior. A test case exercising the SUT under optimal environment conditions requires to have stable changes from $\neg\beta(a,b,t)$ to $\beta(a,b,t)$ without ever "bouncing" back to $\neg\beta(a,b,t)$. This means that paths along locations $l_2 \to l_3 \to l_6$ should not occur. Using DC, this is expressed as

$$C_{opt} \stackrel{\text{def}}{=} \neg\Diamond(\lceil in_{l_2} \rceil \frown \lceil in_{l_3} \rceil \frown \lceil in_{l_6} \rceil)$$

(5) Frequency ratio of different behaviors. One of the advantages of DC when compared to other temporal logics like LTL consists in the possibility to count how often certain computation steps are performed or how often certain conditions evaluate to `true` or `false`. This can be used to specify frequency ratios of different types of SUT behavior to be observed during test executions. Assume, for example, that we wish to specify a test case where the SUT spends twice as many execution steps in states where $\beta(a,b,t)$ holds than is states satisfying $\neg\beta(a,b,t)$. Assume further that this ratio should be roughly valid throughout the test, so that $\beta(a,b,t)$ holds in 200 out of 300 execution steps, with an acceptable deviation of ± 10. This can be expressed as

$$C_{ratio} \stackrel{\text{def}}{=} \Box(\eta = 300 \Rightarrow 190 \leq \Sigma\beta(a,b,t) \leq 210)$$

(6) Boundary tests. A typical test objective is tor explore boundary conditions in the SUT behavior. For $\mathcal{H}_1$, a boundary test consists in creating the maximal instability still counting as normal behavior. This means that the limit value $n = 10$ should be reached during the test execution. The corresponding DC assertion is $C_{limit_1} \stackrel{\text{def}}{=} \Diamond\lceil n = 10 \rceil$. A boundary test that *always* exercises instability to the admissible limit can be specified by

$$C_{limit_2} \stackrel{\text{def}}{=} \Box(\Diamond(\lceil in_{l_2} \rceil \frown \lceil in_{l_3} \rceil) \Rightarrow \Diamond(\lceil in_{l_2} \wedge n = 10 \rceil \frown \lceil in_{l_3} \rceil))$$

5 Logic

Our approach to automated testing exploits arithmetic constraint solving for test case generation and test evaluation. The constraint solver underlying our approach (to be described in Sect. 8) addresses satisfiability of non-linear arithmetic constraints over real-valued and integer variables plus Boolean variables for encoding the control flow, i.e. the input constraint formulae are built from quantifier-free constraints over the reals, integers and from propositional variables using arbitrary Boolean connectives. The atomic constraints are relations between potentially non-linear terms involving transcendental functions, like $\sin(x + \omega t) + ye^{-t} \leq z + 5$. By the front-end of our constraint solver, these constraint formulae are rewritten to equisatisfiable, quantifier-free constraint formulae in conjunctive normal form, with atomic propositions ranging over propositional variables and arithmetic constraints confined to a form resembling three-adress code. Thus, the *internal* syntax of constraint formulae is as follows:

$$\begin{array}{rcl}
\mathit{formula} & ::= & \{\mathit{clause} \wedge\}^* \mathit{clause} \\
\mathit{clause} & ::= & (\{\mathit{bound} \vee\}^* \mathit{bound}) \mid \\
 & & (\mathit{bound} \vee \mathit{equation}) \\
\mathit{bound} & ::= & \mathit{variable}\ \mathit{comp}\ \mathit{rational_const} \\
\mathit{comp} & ::= & < \mid \leq \mid > \mid \geq \\
\mathit{variable} & ::= & \mathit{arith_var} \mid \mathit{boolean_var} \\
\mathit{arith_var} & ::= & \mathit{real_var} \mid \mathit{int_var} \\
\mathit{equation} & ::= & \mathit{triplet} \mid \mathit{pair} \\
\mathit{triplet} & ::= & \mathit{arith_var} = \mathit{arith_var}\ \mathit{bop}\ \mathit{arith_var} \\
\mathit{pair} & ::= & \mathit{arith_var} = \mathit{uop}\ \mathit{arith_var} \\
\mathit{bop} & ::= & + \mid - \mid * \mid / \mid \ldots \\
\mathit{uop} & ::= & - \mid \sin \mid \exp \mid \ldots
\end{array}$$

Such constraint formulae are interpreted over valuations $\sigma \in (BV \to \mathbb{B}) \times (RV \to \mathbb{R}) \times (IV \to \mathbb{Z})$, where BV, RV, and IV are the sets of Boolean, real-valued, and integer variables, respectively. Let $Var := BV \cup RV \cup IV$ be the set of all variables. $\mathbb{B}$ is identified with the subset $\{0,1\}$ of $\mathbb{R}$ such that any rational-valued bound on a Boolean variable v corresponds to a literal v or $\neg v$. The definition of satisfaction is standard: a

constraint formula ϕ is satisfied by a valuation iff all its clauses are satisfied, i.e. iff at least one atom is satisfied in any clause, where the term *atom* refers to both bounds and equations. Satisfaction of atoms is wrt. the standard interpretation of the arithmetic operators and ordering relations over the reals.

6 Predicative encoding of TDIOHS

In order to perform test case generation by constraint solving techniques, we generate a symbolic representation of runs of length $k \in \mathbb{N}$ as in bounded model checking [3]. There are various ways of doing this, all with specific strengths and weaknesses. We present here one particular form of such an unrolling which is very similar to the one used by Audemard et al. for BMC of linear hybrid automata [1], albeit modify it to non-linear, yet time-discrete case of TDIOHS.

Let $\mathcal{H} = (Loc, Init, V, I, O, Trans)$ be a TDIOHS. In order to encode a transition sequence of $\mathcal{H}$ of some given length $k \in \mathbb{N}$, we proceed as follows:
(1) For each discrete state $\sigma \in Loc$ we take k Boolean variables σ^i, with $1 \leq i \leq k$. The value of σ^i encodes whether the automaton $\mathcal{H}$ is in state σ in step i. Here, we take "one-hot" encoding, i.e. $\sigma^i = \texttt{true}$ iff $\mathcal{H}$ is in state σ in step i. With one-hot encoding, there consequently is, for any $i \leq k$, exactly one $\sigma \in Loc$ such that σ^i holds, which is enforced in the symbolic representation of the run through the formula

$$\bigwedge_{i=1}^{k} \left(\sum_{\sigma \in Loc} \sigma^i = 1\right) \ ,$$

where $\sum_{\sigma \in Loc} \sigma^i = 1$ abbreviates an equi-satisfiable CNF fragment over triplets and bounds that is obtained by introduction of helper variables. For example, $\sigma_1 + \sigma_2 + \sigma_3 = 1$ denotes the CNF $(false \geq 1 \vee h_1 = \sigma_1 + \sigma_2) \wedge (false \geq 1 \vee h_2 = h_1 + \sigma_3) \wedge (h_2 \leq 1) \wedge (h_2 \geq 1)$ with fresh helper variables h_1, h_2 and $false$ being a Boolean variable defined by the clause $(false \leq 0)$.
(2) For each transition $\tau \in Trans$ we take $k-1$ Boolean variables τ^i, with $1 \leq i \leq k-1$. The value of τ^i encodes via one-hot encoding whether the ith move in the run is transition τ. Wellformedness of the unrolling in the sense that exactly one transition is taken in each step is guaranteed by conjunctively adding the CNF

$$\bigwedge_{i=1}^{k-1} \left(\sum_{\tau \in Trans} \tau^i = 1\right)$$

to the formula.
(3) For each continuous and discrete state component $x \in V \setminus I$ we take $k-1$ (real-valued or integer) variables x^i. The value of x^{i+1} encodes the value of x after the ith transition in the run. For each $1 \leq i \leq k-1$ and each $y \in I$ we do, furthermore, take one (real-valued or integer) variable y^i representing the respective input value in the ith step of the run. This allows us to formalize the *assignments effected by transitions* by

$$\bigwedge_{i=2}^{k} \bigwedge_{\tau \in Trans} \bigwedge_{x \in V \setminus I} \left(\tau^i \leq 0 \vee x^i = t_x^{i-1}\right) \ ,$$

where t_x^{i-1} denotes the term t assigned to x by transition τ, yet with all occurrences of variables $z \in V$ being substituted by z^{i-1}. Note that an equi-satisfiable triplet form of the equation $x^i = t_x^{i-1}$ may again be obtained through introduction of helper variables.
(4) The *interplay between discrete states and transitions* requires that τ^i implies $\sigma^i = 1$ and $\sigma'^{i+1} = 1$ for $\tau = (\sigma, g, a, \sigma')$. This can be expressed by the CNF

$$\bigwedge_{i=1}^{k-1} \bigwedge_{\tau \in Trans} \bigwedge_{x \in V \setminus I} \left(\left(\tau^i \leq 0 \vee \sigma^i \geq 1\right) \wedge \left(\tau^i \leq 0 \vee \sigma'^{i+1} \geq 1\right)\right) \ .$$

(5) Furthermore, enabledness of the transition, i.e. validity of the *transition guard*, is enforced through the CNF

$$\bigwedge_{i=1}^{k-1} \bigwedge_{\tau \in Trans} \left(\tau^i \leq 0 \vee g^i\right) \ ,$$

where g^i denotes the guard g with all occurrences of variables $z \in V$ being substituted by z^i.
(6) Finally, we have to add constraints describing the allowable *initial states* through the guarded linear constraint system

$$\bigwedge_{\sigma \in Loc} \left(\sigma^1 \leq 0 \vee init_\sigma^1\right) \ ,$$

where $init_\sigma^1$ denotes $Init(\sigma)$ with all occurrences of variables $z \in V$ being substituted by z^1. Conjunction of (1) to (6) yields a formula $\psi_{\mathcal{H}}^k$ formalizing the runs of $\mathcal{H}$ of length k. Satisfying valuations of $\psi_{\mathcal{H}}^k$ are, after removal of helper variables, in one-to-one correspondence to the runs of $\mathcal{H}$ of length k. Using standard techniques from predicative semantics [12], above translation scheme can be extended to both shared variable and synchronous message-passing parallelism, thereby yielding formulae of size linear in the number of parallel components. The same applies for hierarchical automata models, like e.g. StateCharts [4].

7 Encoding the Test Case Specification

In order to be able to generate runs through the TDIOHS which satisfy the test case specification given as a DC formula ϕ, we need to obtain an encoding within the constraint logics of Sect. 5 of traces of length $k \in \mathbb{N}$ satisfying ϕ. [9] provides a polynomial solution for this. Given ϕ and k, this procedure generates a propositional formula $BMC(\phi, k)$ in CNF form of size $O(k^3 \cdot |\phi|)$.

$BMC(\phi, k)$ contains some auxiliary variables beyond the problem variables occurring freely in ϕ. Ignoring these, the models satisfying $BMC(\phi, k)$ are in one-to-one correspondence to the runs of length $\leq k$ satisfying ϕ, with the same naming convention applied for distinguishing the different temporal instances of variables as in Sect. 6. I.e., there is a satisfying valuation of $BMC(\phi, k)$ with $x^i = v$ iff there is a run of length $\leq k$ satisfying ϕ with x in step k having value v.

Due to this correspondence in the naming conventions, the constraint formula $\psi_{\mathcal{H}}^k \wedge BMC(\phi, k)$ formal-

izes all runs of $\mathcal{H}$ of length at most k which satisfy the test case specification ϕ.

8 Solver

For generating concrete test cases we use the solver technology we introduced in [10]. This approach exploits the similarities of *interval constraint propagation* (ICP) (e.g. [6, 5]) and *DPLL SAT solving* (e.g. [19, 7, 8]). Recent algorithmic enhancements of propositional SAT solving like lazy clause evaluation, conflict-driven learning (see below), and non-chronological backjumping that were instrumental to the enormous performance gains in SAT solving are thus adapted to ICP-based arithmetic constraint solving. This tight integration of algorithms from both domains has proven to be very powerful, tackling arithmetic formulae with thousands of real-valued, integer and Boolean variables, thereby providing speedup factors of over a million compared to versions lacking SAT-based conflict-driven learning [10, Sect. 5].

Constraint-Solving Algorithm. Instead of real- and integer-valued valuations of variables, our constraint solving algorithm manipulates interval-valued valuations $\rho \in (BV \to \mathbb{I}_\mathbb{B}) \times (RV \to \mathbb{I}_\mathbb{R}) \times (IV \to \mathbb{I}_\mathbb{Z})$, where $\mathbb{I}_\mathbb{B} = 2^\mathbb{B} \setminus \emptyset$ and $\mathbb{I}_\mathbb{R}$ (resp. $\mathbb{I}_\mathbb{Z}$) is the set of non-empty convex subsets of $\mathbb{R}$ (resp. $\mathbb{Z}$). (Note that this definition covers the open, half-open, and closed non-empty intervals over $\mathbb{R}$ and $\mathbb{Z}$, including unbounded intervals.) Slightly abusing notation, we write $\rho(x)$ for $\rho_{\mathbb{I}_\mathbb{B}}(x)$, $\rho_{\mathbb{I}_\mathbb{R}}(x)$ or $\rho_{\mathbb{I}_\mathbb{Z}}(x)$ when $\rho = (\rho_{\mathbb{I}_\mathbb{B}}, \rho_{\mathbb{I}_\mathbb{R}}, \rho_{\mathbb{I}_\mathbb{Z}})$ and $x \in BV$, $x \in RV$ or $x \in IV$, respectively. If both σ and η are interval assignments then σ is called a *refinement* of η iff $\sigma(v) \subseteq \eta(v)$ for each $v \in Var$. Given $\varepsilon > 0$, we call an interval valuation ρ *ε-consistent* with a constraint formula ϕ iff $\forall x \in Var : \sup \rho(x) - \inf \rho(x) \leq \varepsilon$ and each clause of ϕ contains at least one consistent atom. Consistency of atoms is defined as follows:

$$\begin{aligned}
&\rho \models x \sim c \quad \text{iff} \quad \rho(x) \subseteq \{u \mid u \in \mathbb{R}, u \sim c\} \\
&\qquad \text{for } x \in Var, c \in \mathbb{Q}, \\
&\rho \models x = y \circ z \quad \text{iff} \quad \rho(x) \cap \rho(y) \hat{\circ} \rho(z) \neq \emptyset \\
&\qquad \text{for } x, y, z \in RV \cup IV, \circ \in bop, \\
&\rho \models x = \circ y \quad \text{iff} \quad \rho(x) \cap \hat{\circ} \rho(y) \neq \emptyset \\
&\qquad \text{for } x, y \in RV \cup IV, \circ \in uop,
\end{aligned}$$

where $\hat{\circ}$ is a conservative interval extension of operation $\circ$, i.e. satisfies $i_1 \hat{\circ} i_2 \supseteq \{x \circ y \mid x \in i_1, y \in i_2\}$ for all intervals i_1 and i_2 [18].

An interval valuation ρ is called *inconsistent with a formula (or clause, atom)* ϕ iff there is no ε-consistent refinement η of ρ for any $\varepsilon > 0$. Deciding inconsistency of an atom (yet not, in general, a formula) and ε-consistency of a formula over an interval valuation is straightforward. If ρ is neither ε-consistent nor inconsistent with ϕ then we call ϕ *inconclusive on* ρ.

SAT-based interval constraint solving. We present here an algorithm that, given an $\varepsilon > 0$ and a constraint formula ϕ, exploits SAT-solving and interval constraint propagation (ICP, for short) to find an ε-consistent valuation of ϕ, if existent, and reports inconsistency of the formula otherwise. The underlying idea of our algorithm is that the two central operations of ICP-based arithmetic constraint solving —interval contraction by constraint propagation and by interval splitting— correspond to asserting bounds on variables $v \sim c$ with $v \in Var$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{Q}$. Likewise, the decision steps and unit propagations in DPLL proof search correspond to asserting literals. A unified DPLL- and ICP-based proof search on a formula ϕ from the formula language of Sect. 5 can thus be based on asserting or retracting atoms of the formula language, thereby in lockstep refining or widening an interval valuation ρ that represents the current set of candidate solutions:

1. Proof search on ϕ starts with an empty set of asserted atoms and the interval valuation ρ being the minimal element wrt. the refinement relation on interval valuations, i.e. all intervals being maximal ($\{\texttt{false}, \texttt{true}\}$ for Boolean variables and $\mathbb{R}$ (resp. $\mathbb{Z}$) or —if the variable has a bounded range— a maximal subrange thereof for real-valued (resp. integer) variables).
2. It continues with searching for *unit clauses* in ϕ, i.e. clauses that have only one inconclusive (on ρ) atom left and all other atoms being inconsistent with the current interval valuation ρ. If such a clause is found then its unique unassigned atom is asserted. (The effect of asserting an atom will become apparent in step 3.) The asserted atom stems from the formula ϕ or some learned conflict-clause and may thus be an arbitrary bound, triplet, or pair. Step 2 is repeated until all unit clauses have been processed.
3. If there is an asserted atom a that is inconclusive under the current interval valuation ρ then the contractors corresponding to a are applied to ρ. In the case of triplets and pairs, these contractors are the usual contractors of ICP (cf., e.g., [20, 14]). Given a constraint ϕ and an interval valuation ρ, so-called *contractors* compute another interval valuation $C(\phi, \rho)$ such that $C(\phi, \rho) \subseteq \rho$ and $C(\phi, \rho)$ contains all solutions of ϕ in ρ. For bounds $a = v \sim c$, contraction amounts to replacing $\rho(v)$ with $\rho(v) \cap \{x \in \mathbb{R} \mid x \sim c\}$, no matter whether they are literals or bounds on real-valued or integer variables. In case of triplets and pairs, the contractions obtained are in turn asserted as bounds (this is redundant for contractions stemming from bounds, as the asserted atoms would be equal to the already

asserted bound which effected the contraction).

This step is repeated until no further contraction is obtained (In practice, one stops as soon as the changes become negligible.), or until contraction detects a conflict in the sense of some interval $\rho(v)$ becoming empty. In case of a conflict, some previous splits (cf. step 4) have to be reverted, which is achieved by backtracking —thereby undoing all assertions being consequences of the split— and by asserting the complement of the previous split. Furthermore, a reason for the conflict can be recorded as a conflict clause, thus pruning the remaining search space (see below). If no conflict arose then, if new unit clauses resulted from the contraction, the algorithm continues at step 2, otherwise at 4.

4. The algorithm checks whether ρ is ε-consistent with ϕ and stops if so. Otherwise, it applies a *splitting step*: it selects a variable $v \in Var$ that is interpreted by a non-point interval (i.e., $|\rho(v)| > 1$) and splits its interval $\rho(v)$ by asserting a bound that contains v as a free variable and which is inconclusive on ρ. (Note that being inconclusive on $\rho(v)$ implies that the set of models of the bound neither covers $\rho(v)$ nor is disjoint to $\rho(v)$. I.e., the asserted bound splits $\rho(v)$ into two non-trivial parts. Consequently, the complement of such an assertion also is a bound and is inconclusive on ρ too.) Thereafter, the algorithm continues at 2. If no such variable v exists then the search space has been exhausted and the algorithm stops with result "unsatisfiable".

Conflict-driven Learning. In order to be able to tell reasons for conflicts (i.e., empty interval valuations) encountered, our solver maintains an implication graph akin to that known from propositional SAT solving [23]: all asserted atoms are recorded in a stack-like data structure (unwound upon backtracking, when the assertions are retracted). Within the stack, each assertion not originating from a split, i.e. each assertion a originating from a contraction (including unit propagations), comes equipped with pointers to its antecedents. In this case, a is a bound, i.e. a literal or a inequation $v \sim c$. The antecedent of a is an atom b containing the variable v plus a set of bounds for the other free variables of b which triggered the contraction a.

By following the antecedents of a conflicting assignment, a reason for the conflict can be obtained: reasons correspond to cuts in the antecedent graph, and such reasons can be "learned" for pruning the future search space by adding a *conflict clause* containing the disjunction of the negations of the atoms in the reason. We use the unique implication point technique [23] to derive a conflict clause which is general in that it contains few atoms and which is asserting upon backjumping to the last decision level contributing to the conflict, i.e. upon undoing all decisions and contractions younger than the chronologically youngest decision among the antecedents of the conflict.

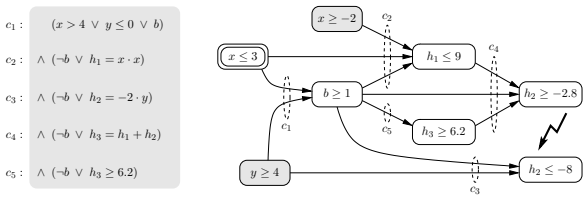

Figure 2: Conflict analysis.

An example of our conflict analysis scheme is depicted in Fig. 2. Let the clause set $c_1, \ldots, c_5$ be a fragment of a formula to be solved. Assume $x \geq -2$ and $y \geq 4$ have been asserted on decision levels k_1 and k_2, resp., and another decision level is opened by asserting $x \leq 3$. The resulting implication graph, ending in a conflict on h_2, is shown on the right. Edges relate implications to their antecedents, dashed ellipses indicate the propagating clauses. Following the implication chains from the conflict yields the conflict clause $\neg(x \geq -2) \vee \neg(x \leq 3) \vee \neg(y \geq 4)$ which becomes unit after backjumping to decision level $max(k_1, k_2)$, propagating $x > 3$.

Generating Concrete Test Cases. Whenever the above constraint solving algorithm stops with an ε-consistent valuation ρ, we start a second phase that concretizes the candidate trace encoded by ρ. (Note that ρ encodes a bundle of concrete, real-valued valuations, as it provides non-point intervals for most of the entailed variables. Each of those real-valued valuations may or may not satisfy the formula, i.e. may or may not be a concrete run of the system.) We do so by successively replacing the intervals for input variables y given by $\rho(y)$ with concrete values in $\rho(y)$. In between, we continue with ICP and -if necessary- backtracking and choice in order to refine ρ, thus enhancing the likelyhood of finding a concrete test case. Therefore, the algorithm proceeds as follows:

5. The algorithm selects a variable $y_j \in I$ and a step number $1 \leq i < k$ such that $|\rho(y_j^i)| > 1$, i.e. $\rho(y_j^i)$ is non-point, if such an y_j^i exists. It then takes an arbitrary value $v_j^i \in \rho(y_j^i)$, replaces the interval $\rho(y_j^i)$ by the point interval $\rho'(y_j^i) = \{v_j^i\}$, and continues at step 2. (Classical heuristics from testing suggest to either take the mean value $v_j^i = \frac{\inf \rho(y_j^i) + \sup \rho(y_j^i)}{2}$ or some extremal value in $\rho(y_j^i)$.) If no such y_j^i exists then we have completed construction of the test case. The result of this step is an ε-consistent valuation that assigns point intervals to all inputs. Thus, we have concrete

values for all inputs and interval-based, overapproximative reasoning narrowed down to interval width ε has not detected conflicts. The ε-consistent valuation thus obtained makes existence of a corresponding concrete path likely. Note that, due to inclusion of the concretization step 5 into the overall backtrack search, the concretized trace may well deviate from the candidate trace found after step 4. This happens if the candidate trace turns out to be non-concretizable. Backtracking can alter both the choices of transitions and internal states as well as of input valuations when this situation occurs.

Test case execution. Test execution obtained after step 5 amounts to successively, in time steps $i = 1$ to k, feeding the input sequence

$$\langle(\rho(y_1^1),\ldots,\rho(y_n^1)),\ldots,(\rho(y_1^{k-1}),\ldots,\rho(y_n^{k-1}))\rangle \ ,$$

where $\{y_1,\ldots,y_n\} = I$, into the SUT, and to record its answers $\langle(z_1^2,\ldots,z_m^2),\ldots,(z_1^k,\ldots,z_m^k)\rangle$, where $\{z_1,\ldots,z_m\} = O$.

Now, in order to check whether the test was successful, we once again return to constraint solving, thus completing a test-and-prove cycle. Based on the interval valuation ρ obtained from concretization (step 5), we narrow the intervals for the outputs based on the observed values:

6. The algorithm replaces ρ with

$$\rho' = x \mapsto \begin{cases} \rho(x) \cap [\underline{z}_j^i, \overline{z}_j^i] & \text{if } x = z_j^i, z_j \in O, \\ \rho(x) & \text{otherwise.} \end{cases}$$

If the constraint solver, after continuing at step 2, can complete this to an δ-consistent valuation for some arbitrarily chosen $\delta < \varepsilon$ then the test is considered successful and the solver terminates, reporting "test passed". Otherwise, the test did actually activate another path in the model than we have expected. We then try whether the outcome nevertheless is reasonable by starting solving from scratch, yet taking into account the concrete input valuations supplied to the SUT and the measured responses observed on the SUT:

7. Restart the solver at step 1 with the constraint system $\psi_{\mathcal{H}}^k \wedge \mathit{Inp} \wedge \mathit{Out}$, where $\psi_{\mathcal{H}}^k$ is the constraint formula from Sect. 6 which formalizes the dynamics of $\mathcal{H}$, $\mathit{Inp} = \bigwedge_{y_j \in I, 1 \le i < k} (y_j^i = v_j^i)$ encodes the input valuation chosen in the concrete test case, and $\mathit{Out} = \bigwedge_{z_j \in O, 2 \le i \le k} (z_j^i \ge \underline{z}_j^i) \wedge (z_j^i \le \overline{z}_j^i)$ encodes the knowledge about the outputs obtained through measurements on the SUT. If this run yields "unsatisfiable" then we have discovered a deviation between $\mathcal{H}$ and the SUT and thus terminate with result "test failed".

8. Otherwise, we restart the solver with the constraint system $\psi_{\mathcal{H}}^k \wedge \mathit{BMC}(\phi, k) \wedge \mathit{Inp} \wedge \mathit{Out}$. If this yields unsatisfiable then we terminate with "test inconclusive" as not terminating after step 7 implies that the SUT behaves as demanded by $\mathcal{H}$, yet failing now implies that the path taken does not correspond to the test goal. Otherwise, i.e. if no inconsistency is found, we terminate with "test passed".

9 Discussion and Related Work

The main contribution of this paper is twofold. First, we present a more abstract and more convenient way to specify test cases, namely by stating a DC formula specifying the shape of a set of symbolic test cases. Apart from being more convenient for testing experts than explicit construction of test input data and associated SUT outputs, the approach also has the technical benefit to avoid explicit internal representation of test cases by lists or trees representing the concrete runs. Instead, all "candidate runs" which are suitable to check a given test objective are represented by a single propositional constraint formula automatically generated from the DC specification. In contrast to mere backtracking search in a tree representation of test cases, this representation permits use of SAT-solving and constraint-solving techniques to efficiently traverse the search space. Second, by solving the constraint formula $\psi_{\mathcal{H}}^k \wedge \mathit{BMC}(\phi, k)$ to construct concrete test cases, the powerful learning mechanism described in Sect. 8, which previously was applied to the system model $\psi_{\mathcal{H}}^k$ only, does now also cover the test case specification $\mathit{BMC}(\phi, k)$. Thus, we have unified learning on both layers, being able to accelerate both through the recent advances in SAT-based bounded model-checking. To the best of our knowledge, no equivalent techniques operating with real-valued variables in an undecidable domain (involving, a.o., transcendental functions) has been investigated in the literature. Other approaches to test-case generation using SAT-solving either target finite state systems, as in the realm of hardware verification (e.g. [15]), or are confined to decidable domains like formulas containing uninterpreted function symbols with equality or purely linear arithmetic [11]. Our test case generation and test evaluation algorithm, while being an offline algorithm, nevertheless contains a feedback between the constraint solving procedure on the one hand and the SUT on the other hand. The SUT is invoked with input data generated by the solver, the output data then again are used by the solver to further prune its search space and evaluate the test outcome. This test-and-prove cycle can be considered as an *offline* version of the algorithm implemented in UPPAAL-TRON [16]. TRON is an *online* test generation and evaluation tool for real-time systems (modeled as networks of timed automata), where symbolic state-space traversal by the test generator is

performed while the SUT is evolving, with the resulting reachable state set later on being refined based on the response observed from the SUT.

Though this paper focusses on the concepts of a method integration for automating test generation, a majority of the techniques described have already been implemented: For the purpose of generating test cases achieving *multiple condition/decision coverage* against specification models or control flow graphs of concrete code, the interval constraint solving methods have been implemented in a tool and are currently evaluated using industrial-sized applications from the fields of railway and avionics control systems [2]. The accelerations by means of methods originating from Boolean SAT solving problems have been implemented and evaluated in a constraint solving tool for large arithmetic formulae [10]. The translation of DC formulae to SAT problems has been investigated in [9].

References

[1] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *ENTCS*, 89(4), 2004.

[2] B. Badban, M. Fränzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[3] A. Biere, A. Cimatti, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*. Springer, 1999.

[4] U. Brockmeyer. *Verifikation von STATEMATE Designs*. Doctoral dissertation, Dpt. of Comp. Science, Universität Oldenburg, Germany, 1999.

[5] J. G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[6] E. Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Commun. ACM*, 5:394–397, 1962.

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[9] M. Fränzle. Take it NP-easy: Bounded model construction for duration calculus. In W. Damm and E. R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2002*, volume 2469 of *LNCS*, pages 245–264. Springer, 2002.

[10] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Interval Constraint Solving Using Propositional SAT Solving Techniques. In *Proceedings of the CP 2006 First International Workshop on the Integration of SAT and CP Techniques*, pages 81–95, 2006.

[11] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[12] E. C. R. Hehner. Predicative programming. *Commun. ACM*, 27:134–151, 1984.

[13] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[14] T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[15] H.-M. Koo and P. Mishra. Test generation using sat-based bounded model checking for validation of pipelined processors. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 362–365. ACM Press, 2006.

[16] K. G. Larsen, M. Mikucionis, and B. Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press, 2005.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer, 1992.

[18] R. E. Moore. *Interval Analysis*. Prentice Hall, NJ, 1966.

[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*, June 2001.

[20] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, 1990.

[21] P. Pandya. Interval duration logic: Expressiveness and decidability. *ENTCS*, 65(6), 2002.

[22] SC-167. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992.

[23] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[24] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[25] J. Zwiers. *Compositionality, Concurrency, and Partial Correctness — Proof Theories for Networks of Prcoesses and Their Relationship*, volume 321 of *LNCS*. Springer, 1989.