Automated Test Case Generation with SMT-Solving and Abstract Interpretation

Jan Peleska, Elena Vorobev, and Florian Lapschies

Department of Mathematics and Computer Science
University of Bremen, Germany
{jp,elenav,florian}@informatik.uni-bremen.de

Abstract. In this paper we describe an approach for automated model-based test case and test data generation based on constraint types well known from bounded model checking. Our main contribution consists of a demonstration showing how this process can be considerably accelerated by using abstract interpretation techniques for preliminary explorations of the model state space. The techniques described support models for concurrent synchronous reactive systems under test with clocks and dense-time.

1 Introduction

Motivation and Overview. In this paper we present results for model-based test case and test data generation for concurrent real-time systems. The expected behavior of the system under test is specified by a model whose abstract syntax representation is used to derive suitable *symbolic test cases* which are represented as logical constraints G over model computations. The term "symbolic" is used in the sense that at this stage no concrete test data exists yet in order to stimulate a model computation satisfying G. The concrete test data is gained by handling constraint satisfaction problems (CSPs) of the type

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

These CSPs are well-known from the field of bounded model checking: σ_0 is a pre-state from where a model exploration should start. $\Phi(\sigma_i, \sigma_{i+1})$ denotes the transition relation, represented as a first order predicate relating pre-states σ_i to possible post-states σ_{i+1}. $G(\sigma_c)$ is a predicate representing the symbolic test case, so solving $tc(c, G)$ yields test data to satisfy G by performing c transitions from the pre-state σ_0.

In the general case G will not only refer to the target state σ_c but to the complete computation $\sigma_0, \dots, \sigma_c$. By introducing additional *observer components*, however, this more general situation can be reduced to the one captured in (1): the observer runs concurrently with the model and checks whether $G(\sigma_0, \dots, \sigma_c)$ is satisfied. If this is the case the observer performs an auxiliary transition to a

target location ℓ indicating "$G(\sigma_0, \ldots, \sigma_c)$ is satisfied". Then the test case may be re-formulated to "ℓ shall be reached after $c+1$ transitions". In practice, however, this introduction of observers is only infrequently required, because most test cases can be identified by means of predicates on a model state σ_c alone.

For HW/SW integration and system integration testing it is desirable to find the shortest path from σ_0 to a state satisfying G. Therefore it is tried to consecutively solve $tc(1, G), tc(2, G), \ldots$, and stop as soon as a c has been found for which solution of $tc(c, G)$ exists. Given a collection of test cases $G_1, \ldots, G_k$ it is desirable to find a model computation $\sigma_0, \ldots, \sigma_n$ where all of these G_i are covered (not necessarily in a given order). The existence of such a computation has the advantage that the SUT will be driven into a larger number of internal states, as when testing only one G_i at a time and resetting the SUT in between, since this increases the confidence into the SUT reliability. Moreover, SUT resets are often time consuming when testing integrated HW/SW systems. Therefore G is usually specified as the disjunction of the remaining goals to be covered, and every G_i that is reached is removed from this disjunction. If, however, a test case G_i cannot be covered from a given pre-state σ_p within an acceptable number of steps, it is advisable to perform *backtracking* to a suitable state σ_{p-q} from where it is less time consuming for the SMT solver to reach this goal (recall that in general, the running time of the SMT solver depends exponentially on the number c in formula (1), specifying how many times the transition relation is unrolled). Finding this state represents another challenge, because trying to solve the CSP from some σ_{p-q} where G_i cannot be reached within the given limit of transitions wastes time to an extent where backtracking no longer offers any advantage. For tackling CSPs of the type (1) we use an SMT solver which is sketched in Section 2.

Main Contribution. We present an *abstract interpretation algorithm* for concurrent synchronous real-time models which, given an initial state σ_0 and a test case goal G returns a natural number c_0 such that it is guaranteed that no solution of $tc(i, G)$ exists for $0 < i < c_0$. Additionally the abstract interpretation yields boundary conditions to be fulfilled by every solution of $tc(j, G), j \geq c_0$. These conditions can be exploited by the SMT solver to speed up the solution process. To our best knowledge no abstract interpretation algorithms for the concurrent synchronous real-time system paradigm have been suggested before, in particular not for the objective of speeding up automated test data generation (see paragraph on related work below).

The experiments described in Section 5 show that use of the abstract interpreter accelerates a solution process of $tc(1, G), tc(2, G), \ldots$ by an average factor of 1.44 just by being able to avoid infeasible tries to solve $tc(i, G)$ for $i < c_0$. If backtracking is applied the results are even more significant, since the abstract interpreter is very fast in detecting states from where no solution of $tc(i, G)$ exists within admissible range of i: here the average acceleration is 3.09. Observe that the experiments have not been performed on case studies, but on models developed for real-world testing campaigns in the automotive domain.

While our test automation framework is independent on the concrete modeling language[1], we sketch an UML2-based modeling formalism in Section 3 which is suitable to specify the expected behavior of synchronous concurrent real-time systems, in order to illustrate the main contribution of the paper.

Related Work. Modeling formalisms for synchronous systems are of considerable practical value in the field of safety-critical control systems. The formalism presented here is based on UML2.0. A more powerful formalism is SCADE [8] which is widely used in the avionic domain. Our main contribution would work equally well for the SCADE modeling language, because it does not depend on the concrete syntax "front-end", but only on the synchronous paradigm and the availability of the transition relation.

Our abstract interpretation approach is inspired by Cousot's work [5, 4] and uses facts from interval analysis [12]. The Astrée abstract interpreter [6] is specialized on the analysis of embedded C-code and can also handle the effect of concurrent access to global program variables. Our abstract interpretation algorithm does not compete with, but is somewhat complementary to Astrée and its underlying methods: our abstract interpreter aims at the analysis of models on a more abstract level than C code. Similar to Timed Automata, it takes into account the valuations of dense-time clocks ("timers") which is not needed in the domain where Astrée is applied. Moreover, the modeling formalism used in this paper follows closely Harel's Statecharts in the semantics presented in [10] with synchronous execution of enabled transitions in parallel components, while Astrée operates on the semantics of a restricted class of C programs, where concurrency is expressed by interleaving of actions.

The problem of deciding the satisfiability of logical (first order) formulas where propositions may be constraints of 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 [14]. Like most other state-of-the-art SMT solvers [2, 13] 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 a SAT solver decide the satisfiability. Various extensions to pure bit blasting have been proposed [3, 1, 16] which have inspired the SONOLAR implementation, and our solver was ranked second in the division for solving closed quantifier-free formulas over fixed-size bit vectors (QF_BV) at the Satisfiability Modulo Theories Competition (SMT COMP 2010).

2 SMT Solver

Our SMT solver SONOLAR follows the bit blasting approach, so Boolean, integral and floating-point variables are encoded as fixed-width bit vectors, where

[1] An algorithm to generate the transition relation Φ from a given abstract syntax representation of the model suffices in order to support the formalism.

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 into 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 [2, 13]. 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 [15, 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.

SONOLAR has the capability to be called incrementally. This technique allows us to add constraints between solver runs and to add constraints 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.

3 Modeling Formalism

In this section we sketch a modeling formalism for illustration purposes. It is based on an UML2 profile, and Fig. 1 — 3 present a sample model specifying the operation of an automotive controller handling turn indication and emergency flashing. Each model is structured into hierarchic components operating concurrently. Fig. 1 shows the SUT interacting with the testing environment TE via SUT input interfaces (TurnIndLeft,TurnIndRight) (positions (0,0), (1,0), (0,1) of the turn indicator lever), EmerFlash (=1 if emergency flash button is pressed), Voltage (percentage of the nominal voltage) and outputs (FlashLeft,FlashRight) (state of turn indication lamps left and right). The legal ranges of variables are specified by a model invariant (for example, TurnIndLeft/Right may not both be 1 in a normal behavior test), and optionally the admissible TE behaviors can be further restricted by associating nondeterministic timed state machines with the TE model component.

In our example the SUT is further structured into sub-components FLASH_-and and OUTPUT_CTRL. The former controls the decision whether or not to

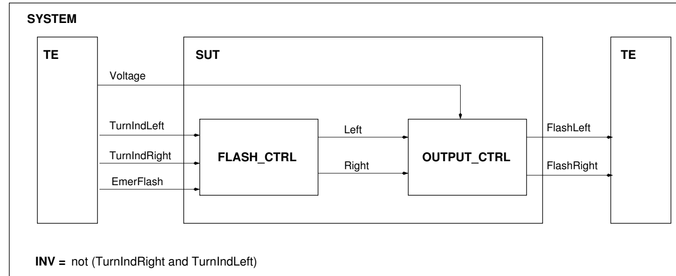

Fig. 1. Complete system consisting of TE and SUT.

activate the turn indication lamps on the left-hand, right-hand or both sides. The latter controls the flashing cycles and automatically switches the lamps off if the actual voltage is less or equal 80% of the nominal voltage. This behavior is encoded by means of state machines S1, S2 as shown in Fig. 2 and 3.

While the EmerFlash button is not pressed, state machine S1 resides in control state EMER_OFF, where the state of the turn indicator lever is simply passed on to OUTPUT_CTRL via internal variables Left and Right, which is expressed by the do-action and its associated assignments. As soon as the EmerFlash button is pressed a state machine transition to basic control state EMER_ACTIVE is performed, where both Left and Right are switched to 1. The state machine transitions inside higher-level control state EMER_ON cope with the situation where the turn indicator lever state changes while emergency flashing is active: turn indication overrides emergency flashing (state TURN_IND_OVERRIDE). When resetting the turn indication lever, emergency flashing is resumed.

State machine S2 reacts on the status of Left, Right and Voltage. As long as Voltage > 80, non-zero states of Left and Right lead to flash cycles with periods of 560 time units. This is controlled by a clock variable t which is reset in basic control states ON and OFF and leads to state machine transitions as soon as the guards $t \geq 340$ or $t \geq 220$ become true. Semantically the clock is encoded as an ordinary real-valued variable, and each clock reset corresponds to storing the current model execution time $\hat{t}$ in t. The guard conditions are then internally evaluated as conditions $\hat{t} \geq t + 340$ and $\hat{t} \geq t + 220$, respectively.

The behavioral semantics of concurrent components is synchronous: both state machines evaluate the same pre-state. If the guard conditions of some transitions between control states evaluate to true a *discrete model transition* is performed by deterministically and simultaneously firing the enabled transitions with the highest priority in each component. The effect of each state machine transition may consist in a change of control states accompanied by a write to internal variables and outputs, while inputs remain unchanged. For calculating these write effects all expressions on the right-hand sides of assignments are evaluated in the pre-state, so that no evaluation order has to be considered. On the other hand, synchronous assignments performed by concurrent components

to the same variables have to be consistent, otherwise a racing condition occurs which has to be fixed in order to gain a valid model. Only if discrete state machine transitions are disabled, a *delay model transition* is performed: the model execution time $\hat{t}$ is advanced by a positive amount, but at most as up to a value where the next timer condition might become true. New values may be placed on the input interfaces, otherwise the model state remains unchanged.

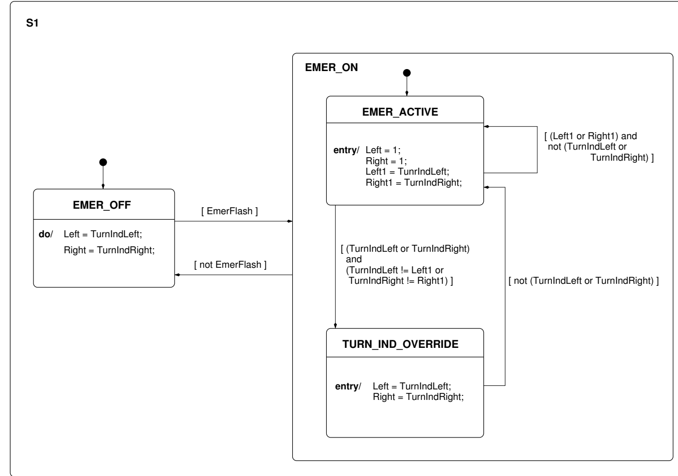

Fig. 2. Statechart S1 associated with component FLASH_CTRL, controlling decisions "flash left" and "flash right".

4 Abstract Interpretation

In this section the detailed specification of the abstract interpretation algorithm is presented. The exposition requires some basic knowledge about lattices and Galois connections, for details readers are referred to [7].

Abstract Domains. Abstract interpretation performs over-approximation on possible model computations. For this approximation we map the concrete data types of state space components to so-called *abstract domains* which are lattices suitable for approximating concrete value sets for each state component. (1) The basic control states $\ell \in Loc(s)$ of each state machine s in the model have concrete data type Boolean, $\sigma(\ell) = 1$ signifying that the state machine resides in ℓ when the system is in state σ. We use the power set lattice $2^{Loc(s)}$ as the associated abstract domain: an element $\{\ell_1, \dots, \ell_k\} \in 2^{Loc(s)}$ represents the knowledge that the state machine currently resides in one of the basic control state $\ell_1, \dots, \ell_k$. We use symbol ℓ_A^s to denote this set-valued control state abstraction for state machine s. (2) Model variables of type Boolean are mapped

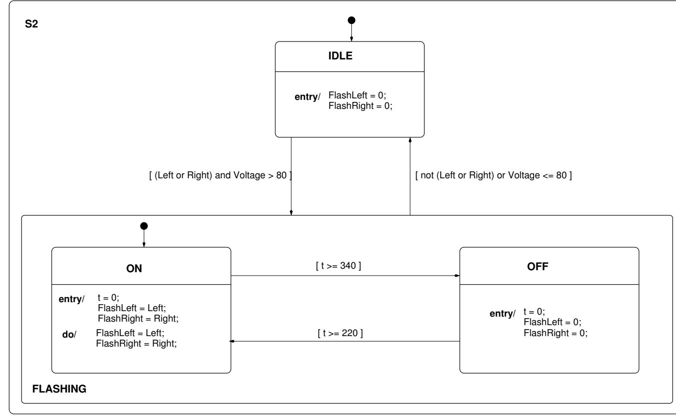

Fig. 3. Statechart S2 associated with component OUTPUT_CTRL managing indication lights and associated flash cycles.

to the lattice $L(\mathbb{B}) = \{\bot, 0, 1, \top\}$ with $\bot \sqsubseteq 0, 1 \sqsubseteq \top$ and $0, 1$ incomparable. Floating point and integer types are mapped to their associated interval lattices. Recall that the lattice join operation is defined by $[x_0, x_1] \sqcup [y_0, y_1] =_{\text{def}} [\min(x_0, y_0), \max(x_1, y_1)]$ for interval lattices, and that the meet operation is just set intersection, $[x_0, x_1] \sqcap [y_0, y_1] =_{\text{def}} [x_0, x_1] \cap [y_0, y_1]$. Model execution time $\hat{t}$ and timer variables are abstracted to intervals over non-negative reals.

Galois Connection. A set $U =_{\text{def}} \{\sigma_1, \ldots, \sigma_n\}$ of concrete model states is mapped to its abstraction $\sigma_A =_{\text{def}} U^{\rhd}$ by setting $\sigma_A(x) = [\min(\{\sigma(x) \mid \sigma \in U\}), \max(\{\sigma(x) \mid \sigma \in U\})$ for integer and float variable symbols x. For Booleans b we define $\sigma_A(b) = \top$ if $\{\sigma(b) \mid \sigma \in U\} = \{0, 1\}$, $\sigma_A(b) = 0$ if $\{\sigma(b) \mid \sigma \in U\} = \{0\}$ and $\sigma_A(b) = 1$ if $\{\sigma(b) \mid \sigma \in U\} = \{1\}$. Furthermore, $\sigma_A(\ell_A^s) = \{\ell \in Loc(s) \mid \exists \sigma \in U : \sigma(\ell) = 1\}$ for the abstracted locations ℓ_A^s of state machines s. Conversely, each abstract state σ_A may be mapped to a set of concrete states by means of the mapping

$$\begin{aligned}{\sigma_A}^{\lhd} =_{\text{def}} \{\sigma \mid \forall\, b : \sigma(b) \sqsubseteq \sigma_A(b) \wedge \forall\, x : \sigma(x) \in \sigma_A(x) \wedge \\ \forall s : \forall\, \ell \in Loc(s) : \sigma(\ell) = 1 \Leftrightarrow \ell \in \sigma_A(\ell_A^s)\}\end{aligned}$$

where b denotes Booleans, s state machines and x floating point and integer model variables. The pair of mappings ${}^{\rhd}, {}^{\lhd}$ represents a Galois connection and its characteristic property $a^{\rhd} \sqsubseteq_2 b \Leftrightarrow a \sqsubseteq_1 b^{\lhd}$ ensures that the algorithm introduced below really computes an over-approximation of all possible computation states.

Goal of the Abstract Interpretation Algorithm. The abstract interpretation algorithm starts from the abstraction $\sigma_A^0 = \{\sigma_0\}^{\rhd}$ of a concrete pre-state σ_0 and calculates a single bounded abstract computation sequence $\langle \sigma_A^0, \ldots, \sigma_A^c \rangle$ such

that each concrete computation $\langle \sigma_0, \ldots, \sigma_c \rangle$ starting in σ_0 is approximated by the abstract sequence in the sense that

$$\forall i \in \{0, \ldots, c\} : \sigma_i \in {\sigma_A^i}^{\lhd}$$

Now suppose that the test case goal G is fulfilled in state σ_c of the concrete computation. Interpreted as a Boolean function on the state space, predicate G may be lifted to the abstract domain by defining

$$[G](\sigma_A) = \begin{cases} 1 \text{ if } \forall \sigma \in {\sigma_A}^{\lhd} : G(\sigma) = 1 \\ 0 \text{ if } \forall \sigma \in {\sigma_A}^{\lhd} : G(\sigma) = 0 \\ \top \text{ otherwise} \end{cases}$$

Since $\sigma_c \in {\sigma_A^c}^{\lhd}$ and $G(\sigma_c) = 1$, evaluation of $[G](\sigma_A^c)$ will result in 1 or $\top$, that is, $1 \sqsubseteq [G](\sigma_A^c)$. Conversely, G will not hold in any σ_i as long as $[G](\sigma_A^i) = 0$. Therefore the objective of the abstract interpretation algorithm is to return the smallest $c_0 \geq 0$ such that $1 \sqsubseteq [G](\sigma_A^{c_0})$ holds. Given this c_0 the SMT solver can try to solve the test case constraint satisfaction problems $tc(c, G)$ specified in (1) with $c = c_0, c_0 + 1, \ldots$, and without having to investigate the feasibility of $tc(m, G)$ for $m < c_0$. Since the abstract interpreter operates significantly faster than the SMT solver, a considerable speed-up can be expected from the fact that the solver skips these $tc(m, G)$.

Abstract Interpretation Algorithm – Introductory Example. To give an intuitive idea of the abstract interpretation algorithm specified formally further below, we assume that our sample system is initialized in a state σ with $\sigma(\hat{t}) = 0$ and σ(TurnIndLeft/Right) = 0, σ(EmerFlash) = 1, σ(Voltage) = 85 and all internal variable and output valuations equal to zero. Suppose further that our test objective is to cover the condition $G \equiv$ S1.ACTIVE.OVERRIDE $\wedge$ S2.FLASHING.OFF starting from this given initial system state. If the abstract interpretation function exploreGoal() is called with $c = 6$ then the algorithm explores abstract interpretation states as shown in the table below, where the columns have the following meaning: **TT** = transition type (DIScrete or DELay or both (DD)); **Si** = sets of possible control states state machines S1, S2 reside in; **TIL, TIR, E, V** = input valuations for TurnIndLeft,...,Voltage; **L, R, L1, R1** = valuations of model variables Left,...,Right1; **t,$\hat{\mathbf{t}}$** = valuations of timer variable t and current execution time $\hat{t}$; **FL, FR** valuation of outputs FlashLeft, FlashRight.

The abstract interpretation algorithm starts by mapping the concrete initial state into its abstract counterpart; the result is displayed in row 0 of the table below: control states are mapped to singleton sets because there is no uncertainty which locations are active. Boolean values are represented in $L(\mathbb{B})$ in the same way, and numeric values are mapped to their single-point interval counterparts. As a result of the initial state valuation only discrete transitions are possible until abstract state 2 is reached, from where only a delay transition may occur. After the delay the inputs may assume arbitrary values, so they are marked by $\top$. Moreover, the model time $\hat{t}$ may have been increased by some positive amount

less or equal 340, where the next timer is bound to elapse. The next transition leading to abstract state 4 may be discrete or a delay, and – due to the full-range input valuations – all guards depending on inputs evaluate to $\top$. As a consequence abstract state 4 admits arbitrary control states, and $[G]$ evaluates to $\top$, so this is the first state where a solution for G may be found. The abstract interpretation algorithm returns with $c_0 = 4$ and also provides a constraint

$$\begin{aligned}
&\beta \equiv \mathrm{ACT}_1 \wedge \mathrm{IDLE}_1 \wedge t_1 = 0 \wedge \hat{t}_1 = 0 \wedge \\
&\mathrm{ACT}_2 \wedge \mathrm{ON}_2 \wedge t_2 = 0 \wedge \hat{t}_2 = 0 \wedge \\
&\mathrm{ACT}_3 \wedge \mathrm{ON}_3 \wedge t_3 = 0 \wedge \hat{t}_3 \in (0, 340] \wedge \\
&t_4 \in [0, 340] \wedge \hat{t}_4 \in (0, 679]
\end{aligned}$$

indicating the restrictions valid at each concrete computation step. This may be used by the SMT solver to reduce the search space.

#	TT	S1	S2	TIL	TIR	E	V	L	R	L1	R1	t	$\hat{\mathbf{t}}$	FL	FR
0.		{OFF}	{IDLE}	0	0	1	[85,85]	0	0	0	0	[0,0]	[0,0]	0	0
1.	DIS	{ACT}	{IDLE}	0	0	1	[85,85]	1	1	0	0	[0,0]	[0,0]	0	0
2.	DIS	{ACT}	{ON}	0	0	1	[85,85]	1	1	0	0	[0,0]	[0,0]	1	1
3.	DEL	{ACT}	{ON}	$\top$	$\top$	$\top$	[0,100]	1	1	0	0	[0,0]	(0,340]	1	1
4.	DD	{OFF, ACT, OVR}	{IDLE, ON, OFF}	$\top$	$\top$	$\top$	[0,100]	$\top$	$\top$	0	0	[0,340]	(0,679]	$\top$	$\top$

Main Function. The top-level function of the abstract interpretation algorithm operates as specified in Fig. 4. Function exploreGoal() is invoked on the current concrete system state σ, and inputs the test case goal G according to Formula (1). Integer $c > 0$ denotes the limit of interpretation steps to be performed. Output β represents a constraint to be constructed by the function. On function return, β contains restrictions about the possible computations states leading to a solution. This auxiliary information may be used by the SMT solver to restrict the search space. The assignment $\sigma_A := \{\sigma\}^{\triangleright}$ creates the abstract start state associated with input σ. In each loop cycle i an abstract interpretation step is performed by means of procedure call $\mathrm{absInt}(\sigma_A, \sigma'_A)$, creating a new abstract state σ'_A. The knowledge that each concrete computation state σ_i is contained in ${\sigma'_A}^{\triangleleft}$ is exploited by adding conjuncts to constraint β, restricting the possible valuations of σ_i: for each state machine s the disjunction of all possible basic control states ℓ the machine may reside in are added as a conjunct to β. Observe that index i adds version information to the basic location identifier ℓ, since this applies to the ith computation state reachable from start state σ. Further restrictions added to β are the bounds for the model execution time $\hat{t}$ in step i and intervals for admissible variable values in this step.

Condition $(1 \sqsubseteq [G](\sigma'_A))$ is evaluated to check whether there is a chance of solving the test case goal in step i. If this is the case the function returns with value i as the first possible computation step number where G may become true, and β contains the restrictions accumulated up to step i. If $[G](\sigma'_A)$ evaluates to 0, the next interpretation cycle is prepared. If limit c is reached without encountering an abstract state satisfying $(1 \sqsubseteq [G](\sigma'_A))$ the function returns with code -1.

```
function exploreGoal(σ : S, G : BExpr, c : ℕ, out β : BExpr) : ℤ
begin
  i := 1; σ_A := {σ}^▷; β := 1; r := −1;
  while i ≤ c do
    absInt(σ_A, σ'_A);
    foreach s ∈ SM do β := β ∧ (⋁_{ℓ∈σ'_A(ℓ^s_A)} ℓ_i); enddo
    β := β ∧ t̂_i ∈ σ'_A(t̂) ∧ (⋀_{x∈I} x_i ∈ σ'_A(x)) ∧ (⋀_{v∈L∪O} v_i ∈ σ'_A(v));
    if (1 ⊑ [G](σ'_A)) then r := i; break; endif
    σ_A := σ'_A; i := i + 1;
  enddo
  exploreGoal := r;
end
```

Fig. 4. Top-level procedure of the state space exploration by means of abstract interpretation. Sets I, L, O denote input, local and output variables, respectively.

Abstract Interpretation Step Procedure. Fig. 5 shows the procedure absInt() for performing one abstract interpretation step: 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 the condition for a discrete model transition to be enabled, $[\text{trigger}_D](\sigma_A)$, is guaranteed to be false, only a delay can occur. In that case, function absIntTime() (Fig. 6) 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 ${D_x}^{\triangleright} \in L(D_x)$[2]. 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 $\sigma_A^1 \sqcup \sigma_A^2$ of the post-states resulting from these two transition types.

Abstraction of Delay Transitions. The calculation of the time bounds for a delay transition is subtle, as can be seen in Fig. 6: The maximal delay may be infinite if no active timer is being observed in the current system state abstracted by σ_A. Therefore the variable limit which is used to store intermediate and final upper bounds of the time growth is initialised by ∞[3]. If some timers are active, the delay is limited by the shortest value at which some state machine is guaranteed to fire a discrete transition. Therefore a loop over all state machines indexed by $i \in 1, \ldots, p$ is performed, and the maximal delay which may occur in one state machine is stored in smLimit. To determine smLimit, the minimal delay

[2] D_x denotes the concrete data type of x. Operator $\oplus$ used in Fig. 5 denotes functional overriding: function $f \oplus \{x \mapsto y\}$ coincides with $f(z)$ for all arguments $z \neq x$, but maps x to y.

[3] In concrete test equipment implementations some suitable value greater than the longest timeout value defined in the SUT model is used instead of ∞, in order to guarantee new stimuli from test equipment to SUT within a reasonable amount of time.

locLimit for each location the state machine may currently reside in, where a timed transition guard is guaranteed to become true is determined. The smallest smLimit-value calculated over all state machines is the global upper bound limit to be returned as the upper bound of the new $\hat{t}$-value[4], because at least one state machine is guaranteed to fire a discrete transition until limit. Since some time has to pass during delay transitions, the lower bound of the new $\hat{t}$-value has to be greater than the old lower bound $\underline{\sigma_A(\hat{t})}$.

---

procedure absInt($\sigma_A : L(S)$, **out** $\sigma'_A : L(S)$)
begin
  if $[\text{trigger}_D](\sigma_A) = 1$ **then**
    absIntDisc(σ_A, σ'_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(σ_A, σ^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

Fig. 5. Single step abstract interpreter.

Abstraction of Discrete Transitions. The abstract interpretation of a discrete transitions is specified in Fig. 7. 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 transitions from ℓ_0 and its higher-level locations are investigated. If a transition τ may fire, that is, if its abstracted trigger condition $\text{trigger}_{s_i}(\tau)$ 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 $\text{trigger}_{s_i}(\tau)$ must have evaluated to 1 in order to get the effect of τ[5].

This effect on the abstracted state space is then calculated by procedure absIntTransEffect() which records these results by changing ζ: Suppose the effect of the transition 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 $\zeta :=$

---

[4] For variables x interpreted in an interval lattice we use $\underline{\sigma_A(x)}$ and $\overline{\sigma_A(x)}$ to denote the lower and upper bounds of their interval valuation, respectively.

[5] For interval lattices we have natural contractors for arithmetic constraints: for example in $L(\mathbb{Z})$, $C_<(x < y; [\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) =_{\text{def}} ([\underline{x}, \min(\overline{x}, \overline{y} - 1)], [\max(\underline{x} + 1, \underline{y}), \overline{y}])$ defines contractions for x and y under the hypothesis that $x < y$ evaluated to true.

function absIntTime($\sigma_A : L(S)$) : $\mathbb{IR}_+$
begin
  limit := ∞;
  foreach $i \in \{1, \ldots, p\}$ **do**
    smLimit := $\underline{\sigma_A(\hat{t})}$;
    foreach $\ell_0 \in \sigma_A(\ell^i_A)$ **do**
      locLimit := ∞;
      foreach $\ell \in \ell_0..s_i,\ (\ell, g, a, \ell') \in \omega_{s_i}(\ell)$ **do**
        if $(\exists g', t, x : g \equiv (\hat{t} \geq x + t \wedge g')) \wedge [g'](\sigma_A) = 1$ **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 := $(\underline{\sigma_A(\hat{t})}, \text{limit}]$;
end

Fig. 6. Function calculating the maximal time interval associated with a delay transition.

$\zeta \oplus \{w \mapsto [\text{expr}](\sigma^1_A)\}$, 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. This join of potential write results and target locations ensures that all potential concrete target states σ_i are really contained in ${\sigma'_A}^{\lhd}$.

If no transition emanating from a location in $\ell_0..s_i$ is guaranteed to fire, that is, $\text{trigger}_{s_i}(\tau) \in \{0, \top\}$ 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 new abstract state σ'_A is constructed by changing the pre-state σ_A with respect to the new sets of potentially active basic control states and the new abstract valuations of variables that have been potentially written to during the abstract interpretation step.

```
procedure absIntDisc(σ_A : L(S), out σ'_A : L(S))
begin
  ζ := ∅; (q_1, ..., q_p) := (∅, ..., ∅);
  foreach i ∈ {1, ..., p} do
    foreach ℓ_0 ∈ σ_A(ℓ^i_A) do
      leave := 0;
      foreach ℓ ∈ ℓ_0..s_i, τ ∈ ω_{s_i}(ℓ), τ ordered by priority do
        if 1 ⊑ [trigger_{s_i}(τ)](σ_A) then
          σ^1_A := σ_A; C(trigger_{s_i}(τ), σ^1_A);
          absIntTransEffect(σ^1_A, τ, ζ, q_i);
          if 1 = [trigger_{s_i}(τ)](σ_A) then leave := 1; break; endif
        endif
      enddo
      if ¬leave then
        σ^2_A := σ_A; C(⋀_{ℓ∈ℓ_0..s_i, τ∈ω_{s_i}(ℓ)} ¬trigger_{s_i}(τ), σ^2_A);
        absIntDoEffect(σ^2_A, ℓ_0, ζ, q_i);
      endif
    enddo
  enddo
  σ'_A := σ_A ⊕ {e_i ↦ q_i | i = 1, ..., p} ⊕ {w ↦ ζ(w) | w ∈ dom ζ};
end
```

Fig. 7. Discrete transition abstract interpreter.

5 Conclusion and Evaluation Results

The evaluation of the combined abstract interpretation, SMT-solving and backtracking approach has been performed using five real-world test models for the system test of automotive control functions which are intellectual property of Daimler[6]: (1) Model TURNIND specifies all automotive functions acting on the turn indication lights, such as turn indication and emergency flashing. (2) Models STOP-START and (3) STOP_START_SYS specify the behavior of the stop-start mechanism controlling automated engine cutoff when stopping at red lights on HW/SW integration and system integration level, respectively. (4) Model POWERWINDOW specifies the functionality of the electronic window regulation, including detection of and reaction on blocking window states, and specialized functions like automated opening of windows for the purpose of ventilation in crash situations and automated closing of windows when entering tunnels. (5) Model POWERTRUNK describes the functionality of the electronic closing mechanism of the trunk lid. Although none of these models involves floating-point arithmetic our system is capable of handling these.

[6] It is currently discussed with Daimler, whether at least one of these models may be published because this would represent valuable information for tool benchmarking. We hope that this will be the case by the time of the NFM2011 conference.

For the evaluation, coverage goals were defined for each model. These goals consisted in specific state machine transitions to be reached, which was equivalent to coverage of certain requirements. Then the test case/test data generation was activated with different techniques, and the execution times have been measured and inserted into the table shown in Fig. 8. This table shows considerable performance improvements for the situations where abstract interpretation is used, with very few outliers where the abstract interpretation leads to a slowdown. Without backtracking the generator was 1.44 times faster on average when using the abstract interpreter. The results were even better with backtracking enabled: with abstract interpretation we observed an average acceleration by a factor of 3.09. This dramatic speed-up when using the abstract interpreter in combination with backtracking can largely be attributed to the fact that the abstract interpreter is very fast at immediately discarding backtracking points from which no new goals can be covered, whereas the solver would spend a lot of time to do so.

While in our current approach the algorithm stops unrolling the transition relation as soon as at least one goal can be satisfied it is generally desirable to satisfy as many goals as possible within a sequence of transitions. Therefore we plan to explore the possibility to extend the present constraint satisfaction problem to an optimization problem that aims to maximize the number of satisfied goals. The necessary means to achieve this are provided by *Partial MAX-SAT* techniques [9].

Model/Config	$\#\mathbf{g_t}$	**#s**	$\mathbf{d_s}$	$(\#\mathbf{g_r})$	$\mathbf{d_{sa}}$	$(\#\mathbf{g_r})$	$\mathbf{d_{sb}}$	$(\#\mathbf{g_r})$	$\mathbf{d_{sba}}$	$(\#\mathbf{g_r})$
TURNIND/1	15	35	5.49	(3)	2.95	(3)	18.06	(3)	3.45	(3)
TURNIND/2	27	35	53.26	(7)	20.91	(7)	82.21	(7)	22.08	(7)
TURNIND/3	46	35	11.68	(8)	7.67	(8)	45.15	(8)	9.70	(8)
TURNIND/4	9	35	5.30	(2)	3.17	(2)	21.39	(2)	3.81	(2)
TURNIND/5	17	35	5.19	(3)	2.94	(3)	18.08	(3)	3.56	(3)
TURNIND/6	11	35	5.32	(2)	2.54	(2)	17.43	(2)	3.02	(2)
POWERTRUNK/1	2	50	55.68	(1)	67.90	(2)	109.93	(2)	67.71	(2)
POWERWINDOW/1	58	40	27.99	(9)	18.18	(9)	89.15	(9)	21.58	(9)
STOP-START/1	13	50	269.62	(3)	376.01	(3)	436.06	(13)	546.09	(13)
STOP-START/2	9	50	3.23	(9)	5.83	(9)	3.20	(9)	5.83	(9)
STOP-START/3	19	50	378.67	(15)	434.45	(15)	619.66	(15)	451.08	(15)
STOP-START/4	28	50	10.93	(17)	10.19	(17)	69.99	(17)	14.07	(17)
STOP-START/5	32	50	6.59	(7)	2.96	(7)	18.44	(7)	3.72	(7)
STOP-START/6	36	50	6.60	(7)	2.96	(7)	18.40	(7)	3.71	(7)
STOP-START/7	36	50	217.12	(36)	191.28	(36)	217.58	(36)	191.39	(36)
STOP-START/8	28	50	998.58	(28)	478.49	(28)	995.35	(28)	477.65	(28)
STOP-START/9	4	50	340.88	(4)	365.99	(4)	341.35	(4)	367.15	(4)
STOP-START/10	12	50	331.50	(8)	358.51	(8)	479.75	(8)	356.80	(8)
STOP-START/11	26	50	337.62	(18)	302.26	(18)	508.46	(18)	315.20	(18)
STOP_START_SYS/1	21	50	588.45	(10)	523.12	(10)	833.10	(21)	648.90	(21)

$\#\mathbf{g_t}$: number of goals to be covered, **#s**: maximal number of transition steps, $\mathbf{d_s}$: execution duration [s] with solver, $\mathbf{d_{sa}}$: execution duration [s] with solver and abstract interpretation, $\mathbf{d_{sb}}$: execution duration [s] with solver and backtracking, $\mathbf{d_{sba}}$: execution duration [s] with solver, abstract interpretation and backtracking, $\#\mathbf{g_r}$: number of covered goals

Fig. 8. Test generation results.

References

1. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Proceedings of FMCAD 2009. pp. 69–76. IEEE (2009)
2. Brummayer, R.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Johannes Kepler University Linz, Austria (November 2009)
3. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Proceedings of TACAS 2007. Lecture Notes in Computer Science, vol. 4424, pp. 358–372. Springer (2007)
4. Cousot, P.: Abstract interpretation: Theory and practice (11–13 April 2000)
5. Cousot, P., Cousot, R.: 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. pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: Okada, M., Satoh, I. (eds.) Eleventh Annual Asian Computing Science Conference (ASIAN'06). pp. 1–24. Springer, Berlin, Tokyo, Japan, LNCS (Dec 6–8 2006), (to appear)
7. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (2002)
8. Esterel Technologies: SCADE Suite Product Description, `http://www.estereltechnologies.com`
9. Fu, Z., Malik, S.: On solving the partial max-sat problem. In: Biere, A., Gomes, C. (eds.) Theory and Applications of Satisfiability Testing - SAT 2006, Lecture Notes in Computer Science, vol. 4121, pp. 252–265. Springer Berlin / Heidelberg (2006)
10. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (October 1996)
11. Jain, H., Clarke, E.M.: Efficient SAT Solving for Non-Clausal Formulas using DPLL, Graphs, and Watched Cuts. In: 46th Design Automation Conference (DAC) (2009)
12. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer-Verlag, London (2001)
13. Jung, J., Sülflow, A., Wille, R., Drechsler, R.: SWORD v1.0. Tech. rep. (2009), sMTCOMP 2009: System Description
14. Ranise, S., Tinelli, C.: Satisfiability modulo theories. TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems 21(6), 71–81 (2006)
15. Sörensson, N.: MiniSat 2.2 and MiniSat++ 1.1. Tech. rep. (2010), SAT-Race 2010: Solver Descriptions
16. Wille, R., Fey, G., Große, D., Eggersglüß, S., Drechsler, R.: SWORD: A SAT like Prover Using Word Level Information. In: Proceedings of VLSI-SoC 2007. pp. 88–93 (2007)