Timed Moore automata: test data generation and model checking

Helge Löding
Verified Systems International GmbH
Bremen, Germany
hloeding@verified.de

Jan Peleska
Center of Information Technology
University of Bremen
Bremen, Germany
jp@informatik.uni-bremen.de

***Abstract*—In this paper we introduce Timed Moore Automata, a specification formalism which is used in industrial train control applications for specifying the real-time behavior of cooperating reactive software components. We define an operational semantics for the sequential components (units) with an abstraction of time that is suitable for checking timeout behavior of these units. A model checking algorithm for livelock detection is presented, and two alternative methods of test case/test data generation techniques are introduced. The first one is based on Kripke structures as used in explicit model checking, while the second method does not require an explicit representation but relies on SAT solving techniques.**

***Keywords*-Timed Moore Automata, model-based testing, model checking, livelocks**

I. Introduction

A. Motivation

In this paper we introduce Timed Moore Automata (TMA), a syntactic and semantic extension of classical Moore Automata [6]. TMA are used in industrial train control applications for specifying the real-time behavior of cooperating reactive software components. While the description formalism has been designed by railway control system suppliers, our contribution consists in the elaboration of test automation and model checking methods for TMA specifications. To this end, we define an operational semantics for the sequential components (units) with an abstraction of time that is suitable for checking timeout behavior of these units: The application software only distinguishes between running and elapsed timers, but does not have a notion of the real time span passing between setting a timer and the associated elapsed-timer event. As a consequence, control decisions involving several timers have to be programmed in a way such that any of the timers may elapse first, though, in reality, only a restricted number of elapsed-timer sequences may be possible. Intuitively speaking, we introduce a simulation semantics for the Timed Automata, so that every computation sequence possible in reality is also a valid computation according to the TMA semantics, but not necessarily vice versa.

Based on this semantics, two alternative fully automated model-based unit test generation algorithm are presented. Both algorithms guarantee full requirements coverage and MC/DC coverage on code level, provided that the code is derived from the model according to a well-defined pattern. As a consequence, the tests are also sufficient to be used for checking the results of an automated code generator. The algorithms are inspired by techniques from explicit and bounded model checking, respectively, using Kripke structures or, alternatively, SAT solving techniques to elaborate the concrete test data. Since the success of model-based testing depends on the models' quality we advocate an approach where model verification and model-based testing are performed in an integrated tool set. Indeed, the similarities between data structures and algorithms used in model checking and test data generation suggest that this is also a promising approach from the perspective of tool construction. For the formalism under consideration the main specification problems observed during practical applications consisted in livelocks. We have therefore integrated automated livelock detection with test data generation. The verification is based on the classical Kripke structures used in explicit model checking, because the limited size of the specification models and the Boolean nature of their input and output interfaces suggests this approach. We sketch briefly, how this technique is also used for large-scale models on an abstracted level.

B. Related work

At least in theory, tests against automata-based formalisms can be made exhaustive as shown, for example, in [1], [9]. In practice, however, this would lead to an infeasible number of tests, even for the units under test we are considering in this paper. Our approach therefore focuses on the generation of *useful* test cases in the sense that they are sufficient to establish compliance with the applicable safety-related standards such as [2]. The application of SAT solving to test case generation also plays an important role for structural test generation based on software code. There, due to the more complex data types usually needed for applications more general than the ones described here, SAT solving is embedded into a Satisfiability Modulo Theory framework allowing to handle arithmetic expressions over integers and floats, as well as pointer expressions [7]. In [5] a timed state machine formalism is considered where timers are represented on a less abstract level than in our paper. While in [5] only atomic input symbols are admitted, our approach

handles Boolean input vectors and Boolean transition guards referring to vector component values.

C. Overview

Section II introduces the specification formalism Timed Moore Automata and section III defines its operational semantics. In section IV Kripke structures over TMA are described and a livelock checking algorithm is presented. Based on these explicit model checking techniques, the first collection of test generation algorithms is presented in section V. As an alternative, section VI presents algorithms based on SAT solving, such that the explicit unfolding of the Kripke state space becomes unnecessary. Section VII presents performance measurements for a collection of TMA used in the railway control domain. Finally, section VIII gives a conclusion and an outlook on ongoing work. There we also motivate the utilization of Kripke structures for more powerful modeling formalisms.

II. Formal definition of Timed Moore Automata

TMA process input symbols $\xi = (\xi_1, \ldots, \xi_n) \in \mathbb{B}^n$ and produce outputs $\eta = (\eta_1, \ldots, \eta_m) \in \mathbb{B}^m$. Just as in classical Moore Automata, output values only depend on the current control state $l \in LOC$ the automaton resides in. Classical Moore Automata, however, process their inputs in discrete steps like a synchronous device, and their state transition function is total. In contrast to this, TMA inputs are interpreted as state vectors and processed in run-to-completion mode: For a given input ξ several consecutive transitions may be taken if they are enabled by ξ. As a consequence, TMA transition relations are not total, and a sequence of consecutive transitions triggered by some input ξ leads to a stable control state l if no transition leaving l is enabled by ξ. A cycle of consecutive enabled transitions represents a livelock situation which is to be avoided because it corresponds to an unwanted endless loop in the TMA implementation.

Additionally TMA extend classical Moore Automata by admitting the definition of a set of Boolean timer activation and timer status variables: when entering a control state, timer activation variables T may be set to 1, leading to the activation of associated timers in the runtime environment of the automaton's implementation. With each activation variable T a status variable t is associated and may be referenced in the Boolean expressions of guard conditions: $t = 1$ indicates that the timer has not yet elapsed, and $t = 0$ indicates that the timer has expired. As a result, TMS operate on an abstracted notion of real time; it is only possible to check whether a pre-defined time duration has passed or not, but the value of this duration in concrete time units is unknown on TMA level. A stable control state l is left as soon as a transition leaving l becomes enabled, either by a change in the input vector or by a timer status changing from "activated to "elapsed". These intuitive concepts are formalized in this section.

A. Abstract Syntax

A Timed Moore Automaton consists of a tuple $(LOC, loc_0, VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}, L_{out}, L_{ta}, R)$, where LOC is the set of locations, $loc_0 \in LOC$ is the initial location, VAR_{in} is the set of input variables, VAR_{out} is the set of output variables, VAR_{ta} is the set of timer activation variables, VAR_{ts} is the set of timer status variables. Function $L_{out} : LOC - \{loc_0\} \longrightarrow 2^{VAR_{out}}$ labels each location with the entry action on output variables. $L_{ta} : LOC - \{loc_0\} \longrightarrow (VAR_{ta} \not\longrightarrow \mathbb{B})$ labels each location with its entry action on zero or more timers. $R = LOC \longrightarrow (\mathbb{N} \not\longrightarrow (GUARD \times LOC))$ is a function associating each location with its outgoing guarded transitions: suppose $R(l_0) = \tau$ for some function $\tau : \mathbb{N} \not\longrightarrow (GUARD \times LOC)$. Then for each $n \in \mathrm{dom}\ \tau$ the image $\tau(n)$ specifies a transition with source location l_0. If $\tau(n) = (g_n, l_n)$ then l_n is the target location of $\tau(n)$ and g_n is its guard condition. The argument n defines the transition's priority. Guards are conjunctions of possibly negated atoms from $VAR_{in} \cup VAR_{ts}$; more formally, $GUARD = NG \times G$, where $NG = G = \mathbb{P}(VAR_{in} \cup VAR_{ts})$ denote the sets of negated and non-negated Boolean atoms from $VAR_{in} \cup VAR_{ts}$, respectively. We use the mapping $\beta : VAR_{ta} \longrightarrow VAR_{ts}$ to associate timer activation variables and their corresponding timer status variables.

B. Static Semantics

Some consistency conditions have to be imposed on the syntax in order to identify well-formed TMA. (1) The symbol sets $VAR_{in}, VAR_{out}, VAR_{ta}, VAR_{ts}$ are pairwise disjoint. (2) No transition may have the initial location as target: If $\tau = R(l)$, $n \in \mathrm{dom}\ \tau$ and $\tau(n) = (g, l_n)$, then $l_n \neq loc_0$. (3) The initial location has just one unguarded emanating transition: $\exists l \in LOC - \{loc_0\} : R(loc_0) = \{1 \mapsto (\varnothing, \varnothing, l)\}$. (4) The priorities of transitions emanating from a given location always consist of consecutive numbers, starting with 1: $\forall l \in LOC - \{loc_0\} : \exists m \in \mathbb{N}_0 : \mathrm{dom}\ R(l) = \{1, \ldots, m\}$. For $m = 0$ the domain is empty, this characterizes terminal locations. (3) Contradicting guard conditions are not allowed: If $\tau(n) = (g_n, l_n)$ and $g_n = (N_n, G_n)$, then $N_n \cap G_n = \varnothing$, i.e., an atom may not appear both negated and unnegated in the guard condition.

C. Abstract and concrete syntax – example

Fig. 1 shows an example of the concrete graphical TMA syntax. The initial location loc_0 is denoted by ●, all other locations are represented by rounded boxes which are labeled by their entry actions. For entry actions on output variables, all variables not listed are implicitly set to 0. For timer activations, only the specified actions are performed.

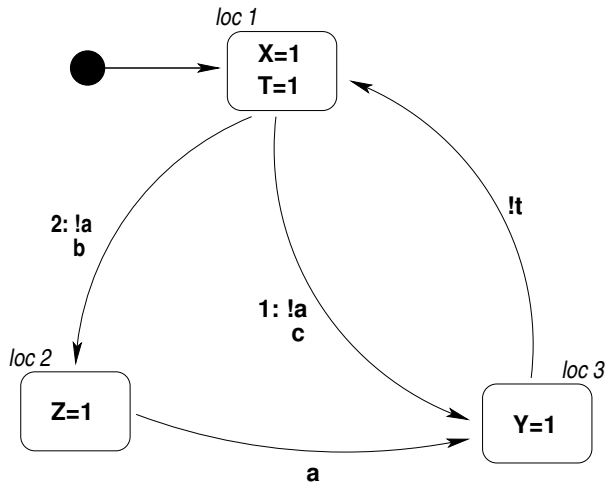

Figure 1. Timed Moore automaton with $VAR_{in} = \{a, b, c\}$, $VAR_{out} = \{X, Y, Z\}$, $VAR_{ta} = \{T\}$, $VAR_{ts} = \{t\}$.

In location loc_1, for example, output X is set to 1 on entry, and Y to 0. Additionally, timer T is activated. If only one transition leaves a location the priority may be omitted, it is implicitly set to 1. Negated atoms a are denoted by $!a$. The abstract syntax of the TMA shown in Fig. 1 is

$$\begin{aligned}
L_{out} &= \{loc_1 \mapsto \{X \mapsto 1, Y \mapsto 0, Z \mapsto 0\},\\
&\quad\ loc_2 \mapsto \{X \mapsto 0, Y \mapsto 0, Z \mapsto 1\}\}\\
&\quad\ loc_3 \mapsto \{X \mapsto 0, Y \mapsto 1, Z \mapsto 0\}\}\\
L_{ta} &= \{loc_1 \mapsto \{T \mapsto 1\},\ loc_2 \mapsto \varnothing,\ loc_3 \mapsto \varnothing\}\\
R &= \{loc_0 \mapsto \{1 \mapsto (\varnothing, \varnothing, loc_1)\},\\
&\quad\ loc_1 \mapsto \{1 \mapsto (\{a\}, \{c\}, loc_3),\\
&\quad\qquad 2 \mapsto (\{a\}, \{b\}, loc_2)\},\\
&\quad\ loc_2 \mapsto \{1 \mapsto (\varnothing, \{a\}, loc_3)\}\}\\
&\quad\ loc_3 \mapsto \{1 \mapsto (\{t\}, \varnothing, loc_1)\}\}\\
\beta &= \{T \mapsto t\}
\end{aligned}$$

III. Semantics of Timed Moore Automata

We define the operational semantics of TMA to be the state transition system (S, S_0, T) with state space S, set of initial states $S_0 \subseteq S$ and transition relation $T \subseteq S \times S$.

A. Variable valuations and state space

The state space S consists of variable valuations for inputs, outputs and timers. In addition it is useful to introduce an auxiliary Boolean variable symbol *stable* indicating that the TMA is in a stable state, i. e. that no further transitions are possible before either some time passes until the next timer elapses or a change in the input vector enables new transitions. States which are not stable are called transient. We model *stable* as an auxiliary output. Let

$$\begin{aligned}
\Sigma_{in} &= VAR_{in} \longrightarrow \mathbb{B}\\
\Sigma_{ts} &= VAR_{ts} \longrightarrow \mathbb{B}\\
\Sigma_{out} &= VAR_{out} \cup \{stable\} \longrightarrow \mathbb{B}\\
\Sigma_{ta} &= VAR_{ta} \longrightarrow \mathbb{B}
\end{aligned}$$

denote the set of associated valuation functions. With these definitions the state space is given by

$$S \subseteq LOC \times \Sigma_{in} \times \Sigma_{ts} \times \Sigma_{out} \times \Sigma_{ta}$$

B. Initial state

The initial state of a TMA has loc_0 as initial location. All outputs are set to 0, all timers in an inactive state, but the inputs may have arbitrary values. Moreover, the initial state is unstable since we have exactly one unguarded transition leaving loc_0:

$$\begin{aligned}
S_0 = \{loc_0\} \times \Sigma_{in} \times (VAR_{ts} \longrightarrow \{0\}) \times\\
(VAR_{out} \cup \{stable\} \longrightarrow \{0\}) \times (VAR_{ta} \longrightarrow \{0\})
\end{aligned}$$

C. Guard valuations and action valuations

We use a fresh symbol ℓ for locations and write $\sigma(\ell) =_{\text{def}} l$ for states $\sigma = (l, \sigma_{in}, \sigma_{ts}, \sigma_{out}, \sigma_{ta}) \in S$. For symbols $x \in VAR_{in} \cup VAR_{ts} \cup VAR_{out} \cup VAR_{ta}$ we define $\sigma(x) =_{\text{def}} \sigma_{in}(x)$ if $x \in VAR_{in}$, $\sigma(x) =_{\text{def}} \sigma_{ts}(x)$ if $x \in VAR_{ts}$, $\sigma(x) =_{\text{def}} \sigma_{out}(x)$ if $x \in VAR_{out}$ and $\sigma(x) =_{\text{def}} \sigma_{ta}(x)$ if $x \in VAR_{ta}$.

We write $\sigma \models x$ if and only if $\sigma(x) = 1$ and $\sigma \not\models x$ if and only if $\sigma(x) = 0$. For a transition $R(\sigma(\ell))(n), n \in \text{dom } R(\sigma(\ell))$, we use the abbreviation $\sigma \models R(\sigma(\ell))(n)$ if $R(\sigma(\ell))(n) = ((N, G), l')$ for some target location l' and the transition guard (N, G) evaluates to 1, that is, all atoms a in the guard condition evaluate to 0 if they appear negated (i. e., $a \in N$), and evaluate to 1 if they appear unnegated (i. e., $a \in G$). More formally,

$$\begin{aligned}
&\sigma \models R(\sigma(\ell))(n) \equiv_{\text{def}}\\
&\quad \exists (N, G) \in GUARD, l' \in LOC :\\
&\qquad R(\sigma(\ell))(n) = ((N, G), l') \wedge\\
&\qquad (\forall a \in N : \sigma \not\models a) \wedge (\forall b \in G : \sigma \models b)
\end{aligned}$$

For an entry action $L_{out}(l)$ on output variables in a location l, we write

$$\begin{aligned}
&\sigma \models L_{out}(l) \equiv_{\text{def}} \sigma(\ell) = l \wedge\\
&\quad (\forall x \in VAR_{out} : (\sigma \models x \Leftrightarrow L_{out}(l)(x) = 1))
\end{aligned}$$

For an entry action $L_{ta}(l)$ on timer activations $\sigma \models L_{ta}(l)$ is defined in an analogous way.

D. Transition relation

Given state S and transition relation $T \subseteq S \times S$ let $\sigma = (l, \sigma_{in}, \sigma_{ts}, \sigma_{out}, \sigma_{ta})$ and $\sigma' = (l', \sigma'_{in}, \sigma'_{ts}, \sigma'_{out}, \sigma'_{ta})$ in the paragraphs below. The elements of transition relation

T are characterized by a predicate $\mathcal{T}(\sigma, \sigma')$ in the sense that $(\sigma, \sigma') \in T \Leftrightarrow \mathcal{T}(\sigma, \sigma')$. Predicate $\mathcal{T}(\sigma, \sigma')$ is defined by

$$\begin{aligned}\mathcal{T}(\sigma, \sigma') \quad \equiv_{\text{def}} \quad & \mathbf{inv}(\sigma) \wedge \mathbf{inv}(\sigma') \wedge \\ & \mathbf{Rule_2}(\sigma, \sigma') \wedge \mathbf{Rule_3}(\sigma, \sigma') \wedge \\ & \mathbf{Rule_4}(\sigma, \sigma')\end{aligned}$$

where the invariant $\mathbf{inv}(\sigma)$ and the rules $\mathbf{Rule_j}(\sigma, \sigma')$ are introduced in the paragraphs to follow.

(1) A state is stable if all guard conditions of leaving transitions evaluate to false. This is formalized by the following invariant, where $\sigma \in S$ with $\sigma = (l, \sigma_{in}, \sigma_{ts}, \sigma_{out}, \sigma_{ta})$:

$$\begin{aligned}&\mathbf{inv}(\sigma) \equiv_{\text{def}} \\ &\quad (\sigma \models \mathit{stable}) \Leftrightarrow \\ &\qquad (\forall n \in \text{dom } R(\sigma(\ell)) : \sigma \not\models R(\sigma(\ell))(n))\end{aligned}$$

(2) Locations, outputs and timer activations remain unchanged if the state is stable. Moreover, elapsed timers do not change their states.

$$\begin{aligned}&\mathbf{Rule_2}(\sigma, \sigma') \equiv_{\text{def}} \\ &\quad (\sigma \models \mathit{stable}) \Rightarrow \\ &\qquad \sigma'(\ell) = \sigma(\ell) \wedge \sigma'_{out} = \sigma_{out} \wedge \sigma'_{ta} = \sigma_{ta} \wedge \\ &\qquad (\forall t \in \mathit{VAR}_{ts} : \sigma \not\models t \Rightarrow \sigma' \not\models t)\end{aligned}$$

(3) In transient states inputs remain stable and time does not pass, that is, all timer states remain unchanged.

$$\begin{aligned}&\mathbf{Rule_3}(\sigma, \sigma') \equiv_{\text{def}} \\ &\quad (\sigma \not\models \mathit{stable}) \Rightarrow \sigma'_{in} = \sigma_{in} \wedge \sigma'_{ts} = \sigma_{ts}\end{aligned}$$

(4) If one or more transitions leaving the current location have guard conditions evaluating to 1, then the one with the best priority (1 is best) is taken. The TMA proceeds to the transition's target location, and the valuations of the output variables change according to the entry action of the target location. A timer activation during an entry action sets the associated timer status to 1. Resetting an active timer by means of its activation variable automatically resets the associated timer status.

$$\begin{aligned}&\mathbf{Rule_4}(\sigma, \sigma') \equiv_{\text{def}} \\ &\quad (\exists n \in \text{dom } R(\sigma(\ell)) : (\sigma \models R(\sigma(\ell))(n) \wedge \\ &\qquad \forall m \in \{1, \ldots, n-1\} : \sigma \not\models R(\sigma(\ell))(m))) \Rightarrow \\ &\qquad \sigma'(\ell) = \pi_2(R(\sigma(\ell))(n)) \wedge \\ &\qquad \sigma' \models L_{out}(\sigma'(\ell)) \wedge \sigma' \models L_{ta}(\sigma'(\ell)) \wedge \\ &\qquad (\forall u \in \text{dom } L_{ta}(\sigma'(\ell)) : \\ &\qquad\quad L_{ta}(\sigma'(\ell))(u) = 1 \Rightarrow \sigma'_{ts} \models \beta(u)) \wedge \\ &\qquad (\forall u \in \text{dom } L_{ta}(\sigma'(\ell)) : \\ &\qquad\quad L_{ta}(\sigma'(\ell))(u) = 0 \Rightarrow \sigma'_{ts} \not\models \beta(u)) \wedge \\ &\qquad (\forall u \in \mathit{VAR}_{ta} - \text{dom } L_{ta}(\sigma'(\ell)) : \\ &\qquad\quad \sigma'(u) = \sigma(u)) \wedge \\ &\qquad (\forall u \in \mathit{VAR}_{ta} - \text{dom } L_{ta}(\sigma'(\ell)) : \\ &\qquad\quad \sigma'(\beta(u)) = \sigma(\beta(u)))\end{aligned}$$

In this rule, π_2 denotes the projection on the second component, that is, the target location, of the transition $R(\sigma(\ell))(n)$. Recall that β maps timer activation variables onto their corresponding timer status variables.

Lemma 1: Let M a Timed Moore Automaton and σ a state of M. Then, if σ is transient, it has exactly one post-state:

$$(\sigma \not\models \mathit{stable}) \Rightarrow (|\{\sigma' \in S \mid \mathcal{T}(\sigma, \sigma')\}| = 1)$$

Proof. If σ is transient then $\mathbf{inv}(\sigma)$ implies that there exists a transition $R(\sigma(\ell))(n)$ leaving location $\sigma(\ell)$ whose guard condition is enabled, that is, $\sigma \models R(\sigma(\ell))(n)$. Without loss of generality let $n \geq 1$ the smallest number satisfying $\sigma \models R(\sigma(\ell))(n)$. Application of $\mathbf{Rule_4}$ now implies the existence of a post-state σ' and uniquely determines $\sigma'(\ell)$, σ'_{out} and σ'_{ta}. For (σ, σ') to be in T the pair also has to satisfy $\mathbf{Rule_3}(\sigma, \sigma')$, and this determines σ'_{in} and σ'_{ts}. $\sigma'(\mathit{stable})$ is uniquely determined from the fact that $\mathbf{inv}(\sigma')$ must hold and $\sigma'(\ell)$ is already fixed. This shows that one and only one σ' exists such that $\mathcal{T}(\sigma, \sigma')$ holds. □

Lemma 2: Let M a Timed Moore Automaton with transition relation $T \subseteq S \times S$. Then T is total, that is, every state $\sigma \in S$ has at least one successor state $\sigma' \in S$ such that $\mathcal{T}(\sigma, \sigma')$.

Proof. From Lemma 1 we know that every transient state has a post-state in T. Now let $\sigma \in S$ be stable. Then $\mathbf{Rule_3}$ does not apply and neither does $\mathbf{Rule_4}$ since $\mathbf{inv}(\sigma)$ implies that the premise of $\mathbf{Rule_4}$ is not fulfilled. Now it is easy to see that there exists at least one (stable or transient) post-state σ' satisfying $\mathbf{inv}(\sigma')$ and $\mathbf{Rule_2}(\sigma, \sigma')$. As a consequence, $\mathcal{T}(\sigma, \sigma')$ holds and therefore $(\sigma, \sigma') \in T$. □

IV. Model checking for Timed Moore automata

This chapter introduces the algorithms used to perform explicit model checking of CTL properties on Timed Moore automata. We use Kripke structures to that end and closely follow [3]. Since these algorithms and the underlying theory are well understood and documented in the literature, we will focus on results that are specific for TMA.

A. Construction of Kripke structures

Every Timed Moore Automaton

$$\begin{aligned}M = (&\mathit{LOC}, \mathit{loc}_0, \mathit{VAR}_{in}, \mathit{VAR}_{out}, \mathit{VAR}_{ta}, \mathit{VAR}_{ts}, \\ &L_{out}, L_{ta}, R)\end{aligned}$$

with operational semantics given by transition system $TS(M) = (S, S_0, T)$ as explained in Section III gives rise to a labeling function $L : S \longrightarrow 2^{AP}$, where AP is the set of atomic proposition over M. Intuitively speaking, $L(\sigma)$ is the set of all propositions that are true in state σ. Now all variables of M have Boolean type; moreover, every location $\ell \in \mathit{LOC}$ may be identified with the Boolean atom $\ell \in \mathbb{B}$, $\sigma \models \ell$ stating that "when in state σ, M resides in location ℓ". As a consequence, the atomic propositions over M are

$$\begin{aligned}AP =_{\text{def}} \ & \mathit{LOC} \cup \mathit{VAR}_{in} \cup \mathit{VAR}_{out} \cup \\ & \mathit{VAR}_{ta} \cup \mathit{VAR}_{ts} \cup \{\mathit{stable}\}\end{aligned}$$

```
function constructSuccessors(in σ : S) : ℙ(S)
  let σ = (l, σ_in, σ_ts, σ_out, σ_ta);
  if (σ ⊨ stable)
    newStates = {σ' = (l', σ'_in, σ'_ts, σ'_out, σ'_ta) ∈ S |
      l' = l ∧ σ'_out = σ_out ∧ σ'_ta = σ_ta ∧
      (∀t ∈ VAR_ts : ¬σ_ts(t) ⇒ ¬σ'_ts(t))}
    ret = calcStable(newStates);
  else
    l' = nextLocation(σ);
    σ_1 = (l', σ_in, σ_ts, σ_out, σ_ta);
    σ_2 = applyLocation(σ_1);
    ret = calcStable({σ_2});
  endif
  constructSuccessors = ret;
end
```

Figure 2. Function *constructSuccessors* constructs the successor set for a given state σ.

and L is determined by $L(\sigma) =_{\text{def}} \{x \in AP \mid \sigma \models x\}$ for all $\sigma \in S$. This construction introduces a Kripke structure $K(M)$ over M by setting $K(M) =_{\text{def}} (S, S_0, T, L)$.

Constructing the Kripke structure in an explicit way requires to obtain an explicit representation of the transition relation $T \subseteq S \times S$. The standard algorithm is a breadth-first search starting with the set of initial states S_0 and calculating transitions by means of a next-state function.

Algorithm *constructSuccessors*(σ) (Fig. 2) explains how to compute the sets of successor states $\{\sigma' \in S \mid \mathcal{T}(\sigma, \sigma')\}$ of a given state σ, as required in algorithm *constructKripke*. Its behavior is dictated by the given system state's running state: If σ is a stable state, then the set of successor states will consist of multiple states with modified inputs and timer states. The set of system states with identical location, identical output valuation, identical timer activation valuation, free input valuation and modified timer status valuation is initially collected in set *newStates*. Recall, that when in a stable state, a timer status may only be modified from 1 (running) to 0 (elapsed), but not vice versa. Function *calcStable* (Fig. 3) modifies the resulting states' *stable* valuation depending on whether the modifications have enables a transition emanating from location l.

If the source system state σ is not stable we employ function *nextLocation* to determine the location of a successor system state. An intermediate system state σ_1 is constructed from initial system state σ by simply changing the associated location. Function *applyLocation* (Fig. 5) applies changes necessary due to entry actions associated with new location l'. Finally, function *calcStable* (Fig. 3) is called to determine whether the resulting successor state is stable.

Function *calcStable* (Fig. 3) loops over all states within a given set and determines, whether each state can transition

```
function calcStable(in states : ℙ(S)) : ℙ(S)
  ret = ∅;
  forall σ ∈ states do
    ret = ret ∪ (σ ⊕ {stable ↦ ¬canTransition(σ)});
  enddo
  calcStable = ret;
end
```

Figure 3. Function *calcStable* reassigns *stable* valuations for a given set of system states.

```
function canTransition(in σ : S) : 𝔹
  ret = false; prio = 1;
  let σ = (l, σ_in, σ_ts, σ_out, σ_ta);
  while (prio ∈ dom R(l)) do
    if (σ ⊨ R(l)(prio))
      ret = true; break;
    endif
    prio = prio + 1;
  enddo
  canTransition = ret;
end
```

Figure 4. Function *canTransition* determines whether a state σ can transition to another location.

to a new location. This is determined by using function *canTransition* (Fig. 4). Each state's *stable* valuation is modified accordingly.

Function *canTransition* (Fig. 4) loops over all transitions emanating from a location l associated with given system state σ in order of priority. If any transition is enabled within σ, the function returns *true*, *false* otherwise. Function *nextLocation* is nearly identical to function *canTransition* (Fig. 4) except that for a given system state σ and associated location l, it will return the target location for the enabled transition emanating from l with highest priority. If no transition emanating from l is enabled, its behavior is undefined. However, since function *nextLocation* is only called for transient states, this poses no problem.

For a given system state σ with associated location l, function *applyLocation* (Fig. 5) modifies valuation functions σ_{out}, σ_{ta} and σ_{ts} according to entry actions associated with l. With the exception of *stable*, all outputs are reassigned according to labeling function L_{out}. All timer activations within $L_{ta}(l)$ are reflected in σ_{ta} and σ_{ts}. For all timer variables, that are not in the domain of $L_{ta}(l)$, timer activation valuations and corresponding timer status valuations remain unchanged.

Since states are valuation functions of Boolean symbols this suggests to encode them as bit-vectors indicating which symbols evaluate to *true* or *false* in σ. Then the labeling

```
function applyLocation(in σ : S) : S
  let σ = (l, σ_in, σ_ts, σ_out, σ_ta);
  σ' = σ;
  forall x in VAR_out do
    σ' = σ' ⊕ {x ↦ (x ∈ L_out(l)};
  enddo
  forall t in dom L_ta(l)do
    σ' = σ' ⊕ {t ↦ L_ta(l)(t)};
    σ' = σ' ⊕ {β(t) ↦ L_ta(l)(t)};
  enddo
end
```

Figure 5. Function *applyLocation* modifies valuations of a given state σ according to entry actions associated with its location.

function L is obtained in a trivial way by collecting in each state the symbols whose associated bits have value 1.

The size of the Kripke structures used in this paper were reduced considerably by means of the delayed nondeterminism technique introduced in [8]: The explicit state representation for valuations $\sigma \in S$ was extended by means of "Don't Care" bits on input valuations σ_{in} indicating whether an input is used in a guard condition of the current control state $\sigma(\ell)$. If an input a is not referenced in any of $\sigma(\ell)$'s guards then the unfolding of its possible values may be delayed, which is marked by setting a don't care bit for a in $\sigma(\ell)$. Only before entering a new location depending on a the unfolding is performed, now leading to different states for each a value. This technique reduces the states and paths of the Kripke structure in a considerable way and was therefore implemented in our tool box. Indeed, the largest of the TMA used for performance evaluation as described later in section VII could not be handled by constructing the corresponding Kripke structure without employing delayed nondeterminism.

B. Checking CTL properties

Given a Kripke structure as introduced above, arbitrary model properties expressible in Computation Tree Logic CTL may be checked, using the well known algorithms of explicit model checking [3]. We will now show in more detail how model checking with respect to livelock properties can be performed for TMA in an optimized manner.

C. Live-lock checking of TMA

A live-lock occurs when a TMA executes a cycle of transient system states. This cycle will never terminate since the TMA is deterministic and the system will never react to changing inputs, because those are only considered when in a stable state. Formally speaking, a live-lock occurs when there exists a cycle $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma_0$ such that σ_0 is reachable from some initial state in S_0, the cycle is consistent with the transition relation, $\forall i \in \{1, \ldots, n\} : \mathcal{T}(\sigma_{i-1}, \sigma_i) \wedge \mathcal{T}(\sigma_n, \sigma_0)$, and each state is transient, i. e., $\forall i \in \{0, \ldots, n\} : stable \notin L(\sigma_i)$. Expressed as a CTL property, live-locks may occur in reachable states σ_0 fulfilling $\sigma_0 \models \mathbf{EG}(\neg stable)$, or, equivalently, in TMA satisfying $\mathbf{EF}(\mathbf{EG}(\neg stable))$. Conversely, for establishing live-lock freedom, the negation of the previous formula has to be proven:

$$livelockFree \equiv_{\text{def}} \mathbf{AG}(\mathbf{AF}(stable))$$

In order to check this property, it must first be transformed into a semantically equivalent form containing only operators $\neg$, $\wedge$, $\vee$, **EX**, **EG** and **EU**:

$$\begin{aligned} & \mathbf{AG}(\mathbf{AF}(stable)) \\ \Leftrightarrow\ & \mathbf{AG}(\neg\mathbf{EG}(\neg stable)) \\ \Leftrightarrow\ & \neg\mathbf{EF}(\mathbf{EG}(\neg stable)) \\ \Leftrightarrow\ & \neg\mathbf{E}(true\ \mathbf{U}\ (\mathbf{EG}(\neg stable))) \end{aligned}$$

In general, checking formulas of the type $\mathbf{EG}\phi$ involves the expensive calculation of (non-trivial) strongly connected components [3, p. 36]. Therefore the following theorem is helpful to increase the efficiency of the live-lock checking algorithm, since the formulas to be checked alternatively in the case of TMA can be handled without the identification of strongly connected components.

Theorem 1: Let M a Timed Moore Automaton. Then

$$(M \models livelockFree) \Leftrightarrow (M \models \neg\mathbf{EF}(\neg\mathbf{EF}(stable)))$$

Proof. Consider a live-lock state σ_0 with property $\sigma_0 \models \mathbf{EG}(\neg stable)$. Then $\sigma_0 \not\models stable$, so, according to Lemma 1, it only has one successor state which is again transient. Repeating this argument for all states on the path p globally satisfying $\neg stable$ implies that p is in fact the *only* path starting at σ_0. We conclude that for TMA the equivalence

$$(\sigma \models \mathbf{EG}(\neg stable)) \Leftrightarrow (\sigma \models \mathbf{AG}(\neg stable))$$

holds. Therefore the existence of a live-lock can be identified by $\mathbf{EF}(\mathbf{AG}(\neg stable))$, or, equivalently, $\mathbf{EF}(\neg\mathbf{EF}(stable))$. As a consequence the absence of live-locks is just the negation of the latter formula, that is, $\neg\mathbf{EF}(\neg\mathbf{EF}(stable))$. □

V. Test data generation

Kripke structures may be utilized to construct test cases according to several test strategies. In this approach, construction of test cases consists of selecting a path to a system state to be tested within a Kripke structure and refining this path into sequences of input assignments and output assertions.

Given a system state $\sigma \in S$ within a Kripke structure $K(M)$, we need to find a path p starting in some initial state $\sigma_0 \in S_0$, containing σ and ending in the next possible stable system state $\sigma' \in S$ with $\sigma' \models stable$. The stability of σ' is required in order to be able to check the SUT with

respect to correctness of the outputs produced and – for grey-box or white-box testing – some aspects of its internal state. Path p and initial state σ_0 are found by performing a backward breadth-first search through the Kripke structure starting in σ. If σ is stable itself, we use it as final system state of our test path since its outputs may be asserted during execution. If σ is a transient system state, we determine the next stable system state σ' and the corresponding path from σ to it by following transition relation T starting from σ. Since Timed Moore Automata are deterministic while passing through transient states, only one such σ' can exist. Observe that these considerations already require M to be live-lock free.

The resulting path p may then be refined into a test case by (1) constructing input assignments to produce initial state σ_0, (2) asserting stable outputs according to each stable system state along the path, and (3) constructing new input assignments according to each respective successor system state to each stable system state along the path. (1) and (3) will then enforce the execution of the selected path, (2) verifies the consistency of implementation and specification.

A. Statement coverage

In order to attain complete statement coverage, it is sufficient to construct test cases covering all locations of a given automaton. Since all assignments to outputs must occur within entry actions of automaton locations, transition coverage is not necessary for statement coverage. For each reachable location $\ell \in LOC$, we find a corresponding state σ in Kripke structure $K(M)$, such that $\sigma \models \ell$. As described above, σ gives rise to a test path p through $K(M)$ and subsequently to a test case visiting location ℓ. When constructing a test case to cover all statements executed within a given location, it is preferable to force the automaton to stabilize in that location in order to be able to assert the effect of all statements within that location. Hence it is advisable to find system states, where the stronger property $\sigma \models (\ell \wedge stable)$ holds. Note that it is not necessary to explicitly require all transitions leaving the location under consideration to be disabled since this is implicitly ensured by $stable$.

B. Decision coverage

For TMA, decision coverage can be achieved by (1) constructing test cases to cover all transitions between locations and (2) constructing test cases enforcing stable states for each location l by setting the inputs appropriately, as long as states σ with $\sigma(\ell) = l$ are not transient for *every* possible input vector. (1) will cover all control structures, that cause locations to change, (2) is needed to negate all such control structures. Combined, (1) and (2) will therefore cause all control conditions to evaluate to $true$ and $false$ at least once. As described above, (2) can be achieved by finding system states σ for each location l, such that $M, \sigma \models l \wedge stable$. To accomplish (1), we need to find a transient system state $\sigma \in S$ for any given transition emanating from a location ℓ, such that σ enables the transition and disables all other outgoing transitions with better priorities. Consider, for example, transitions $(loc_1 \mapsto \{1 \mapsto (\{a\}, \{c\}, loc_3), 2 \mapsto (\{a\}, \{b\}, loc_2)\}) \in T$ from Figure 1. In order to construct a test case covering the transition with priority 2, we need to find a system state σ, such that:

$$\sigma \models loc_1 \wedge \neg a \wedge \neg c \wedge b$$

C. Modified condition / decision coverage

When constructing test cases according to modified condition / decision coverage criteria, we may again take test cases constructed for transition coverage as a basis. For each transition between locations, it remains to be shown by test cases that each atomic guard condition associated with a transition can cause the overall complex guard condition to evaluate to $false$. Since a complex guard condition is merely a conjunction of atomic guard conditions, we require test cases, where input valuations fulfill all but one atomic condition of a given transition. For each location $l \in LOC$ and transition priority $n \in \mathrm{dom}\ R(l)$ consider the pair $(ng, g) \in NG \times G, (ng, g) = \pi_1(R(l)(n)$ formalizing the guard condition of the transition with priority n emanating from location l. We now require a set of modified guard conditions, where precisely one Boolean atom has been inverted. The set of guard conditions, where one negated Boolean atom from original guard condition (ng, g) has been changed to positive polarity is given as:

$$\begin{aligned} GUARD^+(l,n) = \{&(ng', g') \in GUARD \mid \\ &\exists x \in VAR_{in} \cup VAR_{ts} : \\ &\quad ng' = \pi_1(R(l)(n) - \{x\} \wedge \\ &\quad g' = \pi_1(R(l)(n) \cup \{x\}\} \end{aligned}$$

Conversely, the set of guard conditions, where one positive Boolean atom from original guard condition (ng, g) has been changed to negative polarity is given as:

$$\begin{aligned} GUARD^-(l,n) = \{&(ng', g') \in GUARD \mid \\ &\exists x \in VAR_{in} \cup VAR_{ts} : \\ &\quad ng' = \pi_1(R(l)(n) \cup \{x\} \wedge \\ &\quad g' = \pi_1(R(l)(n) - \{x\}\} \end{aligned}$$

Finally, the set of relevant guard conditions for achieving modified condition / decision coverage for given location l and priority n is:

$$GUARD_{mcdc}(l,n) = GUARD^+(l,n) \cup GUARD^-(l,n)$$

For each guard condition $(\{ng_1, \ldots, ng_m\}, \{g_1, \ldots, g_n\}) \in GUARD_{mcdc}(l,n)$, we now require an additional test case by finding a transient system state σ with:

$$\begin{aligned} M, \sigma \models\ & l \wedge \neg stable \wedge \\ & \neg ng_1 \wedge \ldots \wedge \neg ng_m \wedge \\ & g_1 \wedge \ldots \wedge g_n \end{aligned}$$

It is irrelevant, whether such a σ leads to a transient or stable successor state, since for each test case corresponding to a transition to be disabled, either taking another transition or stabilizing in the current location is adequate proof of the effect an inverted guard condition has.

Consider transitions $(loc_1 \mapsto \{1 \mapsto (\{c\},\{a\},loc_3), 2 \mapsto (\{b\},\{a\},loc_2)\}) \in R$ from figure 1 and priority 1. When constructing test cases for modified condition / decision coverage for this transition, we construct $GUARD_{mcdc}(loc_1, 1)$:

$$GUARD^{+}(loc_1,1) = \{(\emptyset,\{a,c\})\}$$
$$GUARD^{-}(loc_1,1) = \{(\{a,c\},\emptyset)\}$$
$$GUARD_{mcdc}(loc_1,1) = \{(\emptyset,\{a,c\}),(\{a,c\},\emptyset)\}$$

The resulting test cases are then constructed by finding system states σ_1 and σ_2 with properties:

$$M,\sigma_1 \models loc_1 \wedge \neg stable \wedge a \wedge c$$
$$M,\sigma_2 \models loc_1 \wedge \neg stable \wedge \neg a \wedge \neg c$$

VI. Test data generation using SAT solving

In this section an alternative to the test generation methods described above is presented: Based on the abstract syntax representation of the TMA model, test cases may be identified as paths through the model, together with logical constraints ensuring that these paths will really be executed. Checking that the constraints can be solved – that is, the test case is feasible – and generating test data from the concrete constraint solutions represents a SAT solving problem which we handle using the MiniSAT tool [4].

A. General behavior

We formalize a trace through a TMA as a sequence of trace transitions. The set of trace transitions $TT \subseteq (LOC \times \mathbb{N})$ consists of tuples $(l,n) \in TT$ of a source location l and the transition priority n of a transition emanating from that source location. Given $trace \in TT^*$, test data for a test case executing that trace consists of a sequence of inputs $inputs \in (\Sigma_{in} \times \Sigma_{ts})^*$ to be assigned in each test step and a corresponding sequence of output $outputs \in (\Sigma_{out} \times \Sigma_{ta})^*$ to be asserted within each test step.

Recall that during execution of TMA some visited locations may be part of a transient system state. This means that a test trace may need to be partitioned into sub-traces corresponding to test steps. Each sub-trace will then end in a stable state where correct output valuations may be asserted and new inputs may be assigned. The function $generateTestData$ (Fig. 6) implements the general behavior of the test data generation for TMA. It receives a test trace as an input parameter and returns a boolean value indicating whether the given test trace is feasible. If feasible, additional output parameters will yield input and output assignments corresponding to single test steps.

At the core of this function, a loop iteratively partitions the given test trace into test steps. This is accomplished by function $nextStep$ (Fig. 8). Each invocation of this function determines, whether another test step along the given test trace is feasible. If so, it returns the inputs $(\sigma'_{in}, \sigma'_{ts})$ necessary to realize the shortest such test step as well as the remaining postfix of the given test trace $trace$, for which additional test steps need to be generated. While test steps along the given test trace are feasible, the loop keeps track of current locations as well as current input and output valuations. On the one hand, this is necessary to provide the needed inputs to $nextStep$, on the other hand this will determine the outputs to be asserted after each test step. Bookkeeping of current locations and variable valuations is accomplished using a concrete interpretation function $interpret$ (Fig. 7), which will compute the next stable successor system state for a given running source system state. The test data generation will only return valid test data, if a given test trace can be completely partitioned into feasible test steps.

```
function generateTestData(in trace : TT*,
                          out inputs :
                                (Σ_in × Σ_ta)*,
                          out outputs :
                                (Σ_out × Σ_ts)*) : 𝔹
    feasible = true;
    inputs =<>;
    outputs =<>;
    σ = σ_0;
    while(feasible and not empty(trace)) do
        (Σ_in × Σ_ta) (σ'_in, σ'_ta);
        feasible = nextStep(trace,
                            σ,
                            (σ'_in, σ'_ts));
        if(feasible) then
            push_back(inputs, (σ'_in, σ'_ts));
            let σ = (loc, σ_in, σ_ts, σ_out, σ_ta);
            σ_in = σ'_in; σ_ts = σ'_ts;
            interpret(σ);
            push_back(outputs, (σ_out, σ_ta));
        endif
    enddo
    return feasible;
end
```

Figure 6. Function $generateTestData$ constructs input and output sequences for a given trace.

B. Concrete interpretation

The concrete interpreter function $interpret : S \to S$ for Timed Moore automata can be implemented using the operational semantics given in chapter III. Given a source system state $\sigma \in S$, such a concrete interpreter function will compute the next stable system state and modify the

```
procedure interpret(inout σ : S)
    let σ = (loc, σ_in, σ_ts, σ_out, σ_ta);
    σ_out = σ_out ⊕ (stable ↦ false)
    while(σ_out(stable) = false) do
        S' = {σ' ∈ S | T(σ, σ')}
        select σ' ∈ S'
        σ = σ'
    enddo
end
```

Figure 7. Procedure *interpret* performs concrete interpretation starting from a given system state σ.

system state accordingly: given the source system state $\sigma = (loc, \sigma_{in}, \sigma_{ts}, \sigma_{out}, \sigma_{ta})$, the interpreter function will initially ensure, that the automaton is in a transient state by overloading $\sigma_{out}(stable)$ to be $false$. Once in a transient state, the interpretation will loop while the automaton transitions. Within each loop, the set of successor states is calculated using T. A successor state is then selected from that set, which will become the new source state for the next loop execution. According to Lemma 1 the set S' will always contain exactly one element and the selection of an element from that set is not arbitrary.

C. Test trace partitioning

Given a system state $(loc, \sigma_{in}, \sigma_{ts}, \sigma_{out}, \sigma_{ta}) \in S$ and a test trace $trace \in TT^*$ starting in location $loc \in LOC$, the function $nextStep$ (Fig. 8) calculates the shortest feasible prefix test trace that ends in a location with a stable system state. In order to accomplish this, an initially empty sequence of trace transitions is iteratively expanded along the given test trace. This sequence is a feasible test step prefix trace if inputs can be constructed, which enforce the trace and cause the final location of the trace to be reached as part of a stable system state. However, if no location along the given test trace can be made stable, but inputs can be found that enforce the entire trace, the entire trace may be viewed as a feasible test step. Since the entire test trace is covered by such a test step, it is irrelevant with respect to the generation of input assignments which location becomes stable next.

Within function $nextStep$ (Fig. 8), sequence $step$ is the test step currently under consideration. Within a loop, it is expanded using trace transitions from the given test trace $trace$. The loop continues while no feasible test step could be found and while there are still trace transitions remaining within the given test trace. For each test step under consideration, function c_{prefix} constructs a boolean constraint expression over input values. If a solution to that constraint can be found, the corresponding inputs will (1) enable all transitions along the test step, (2) disable all transitions with better priorities deviating from the test step, and (3) disable all transitions leaving the final location of the

```
function nextStep(inout trace : TT*,
                  in σ : S,
                  out (σ'_in, σ'_ts) : (Σ_in × Σ_ts) : 𝔹
    sat = false;
    step =<>;
    remain = trace;
    while(not sat and not empty(remain)) do
        trans = pop_front(remain);
        push_back(step, trans);
        constraint = c_prefix(σ, step);
        if(solve(constraint)) then
            sat = true;
            (σ'_in, σ'_ts) = solution(constraint);
            trace = remain;
        endif
    enddo
    if(not sat) then
        constraint = c_full(σ, step);
        if(solve(constraint)) then
            sat = true;
            σ'_in = solution(constraint);
            trace =<>;
        endif
    endif
    return sat;
end
```

Figure 8. Function $nextStep$ partitions a test trace into test steps.

test step in order to make it stable. If no such test step exists and the loop terminates unsuccessfully, function c_{full} will construct a boolean constraint expression, which will only (1) enable all transitions along the entire trace and (2) disable all transitions with better priorities deviating from the entire trace. A solution to this constraint will still yield valid test data since the entire test trace is covered. Calls of $solve$ and $solution$ with a given constraint correspond to invocations of the underlying SAT-solver and yield satisfiability and satisfying valuations respectively.

VII. Performance results

Table I shows performance results for a collection of TMA used as specifications of a "real-world" industrial railway control application. Within table I, each row corresponds to results gathered during model checking and test data generation for a single TMA. Column labels (1) #L, (2) #T, (3) #IN, (4) #TM, (5) #S, (6) t_{KS}, (7) t_{MC}, (8) #TC and (9) t_{TC} denote (1) number of locations within the TMA, (2) number of transitions in the TMA, (3) number of inputs, (4) number of timers, (5) number of Kripke states within the corresponding Kripke structure, (6) time required to construct Kripke structure in milliseconds, (7) time re-

Table I
PERFORMANCE RESULTS

#L	#T	#IN	#TM	#S	t_{KS}	t_{MC}	#TC	t_{TC}
12	34	5	0	310	< 1	< 1	28	< 1
12	37	7	0	501	10	< 1	29	20
12	38	7	0	442	10	< 1	28	60
13	35	8	0	1071	20	20	35	20
16	28	8	1	756	10	10	27	20
18	29	6	2	574	10	10	27	10
18	41	10	0	1485	40	20	38	50
19	28	4	4	306	< 1	< 1	24	10
19	34	7	0	291	< 1	10	33	20
22	38	10	3	513	10	10	37	10
22	40	5	0	455	< 1	< 1	31	10
24	54	14	2	1613	30	30	43	30
25	73	8	3	5095	120	310	63	40
33	48	9	5	3090	50	110	43	20

quired to perform model checking for livelocks on Kripke structure in milliseconds, (8) number of generated test cases ensuring complete transition coverage and (9) time required to construct test cases using the SAT-based approach in milliseconds respectively. All measurements were performed on a standard 2.0 GHz Intel®Core™2 Duo processor with 2 GB of RAM. Note that for some TMA the corresponding Kripke structures were constructed within less than a single millisecond. The construction of Kripke structures never took more than 120 milliseconds to complete. All Kripke structures were sufficiently small to easily fit into RAM. Examining Kripke structures to detect livelocks was always completed within at most 310 milliseconds, many TMA could be checked within less than a single millisecond. Since no TMA contained livelocks (all livelocks detected in previous TMA specification versions had been removed) and since therefore all constructed Kripke states had to be examined, the time measurements constitute worst-case scenarios.

VIII. CONCLUSION

We have introduced the Timed Moore Automata formalism which is used for specification and semi-automated model-based code generation of cooperating software components in railway control systems. An operational semantics for TMA and algorithms for checking these models against livelocks and for automated generation of test cases and associated test data have been presented. Livelock checking and one variant of test generation algorithms were based on Kripke structures as used in explicit model checking. Apart from the fact that, due to the moderate size of the unit specification models, explicit models are acceptable for our railway control application context, there is another reason for utilizing explicit Kripke structures in test automation: When generating test data for models M designed in more general formalisms admitting large data types like 32, 64 and 128 bit integers and floating point numbers, arithmetic conditions and assignments involving arithmetic expressions are abstracted to Boolean atoms. This abstraction induces a model $\mathcal{A}(M)$ simulating the original model M. Kripke structures over $\mathcal{A}(M)$ can be readily exploited to identify classes for equivalence testing: Roughly speaking, different concrete paths through M being abstracted to the same path through $\mathcal{A}(M)$ represent members of the same equivalence class, and therefore it often suffices to test only one or a small number of them.

The test automation techniques described here have been applied in numerous test suites performed by Verified Systems International GmbH for the verification of railway control software of highest criticality level SIL-4 according to the international standard [2]. Compared to conventional unit tests performed before, our techniques reduce the effort by approx. 90%. This extraordinary efficiency improvement could be gained since the TMA models are already provided by the development teams. In other testing campaigns, where testing models had to be elaborated before being able to benefit from the automation capabilities, the efficiency improvement was approx. 30% when compared to conventional test campaigns.

REFERENCES

[1] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.

[2] European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.

[3] Edmund M.Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[4] Niklas Een, Niklas Sörensson. *An Extensible SAT-solver*. SAT 2003.

[5] M. G. Merayo, Manuel Núñez and Ismael Rodríguez. Formal testing from timed finite state machines. *Computer Networks* 52 (2008) 432 – 460.

[6] Moore E. F. *Gedanken-experiments on Sequential Machines.* Automata Studies, Annals of Mathematical Studies, 34, 129153. Princeton University Press, Princeton, N.J.(1956).

[7] Jan Peleska, Helge Löding, and Tatiana Kotas. *Test automation meets static analysis*. In Rainer Koschke, Karl-Heinz Rödiger Otthein Herzog, and Marc Ronthaler, editors, *Proceedings of the INFORMATIK 2007, Band 2, 24. - 27. September, Bremen (Germany)*, pages 280–286.

[8] Bastian Schlich. *Model Checking of Software for Microcontrollers.* PhD Thesis , Aachen, Germany, 2008.

[9] J.G. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.