

Verifying Real-Time Joint Action Specifications Using Timed Automata

Timo Aaltonen

Mika Katara

Risto Pitkänen

Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tta@cs.tut.fi clark@cs.tut.fi rike@cs.tut.fi

Abstract. *In this paper, an approach to the verification of specifications of reactive real-time systems is proposed. DisCo is an object-oriented method based on joint actions. It supports stepwise refinement and specification of real-time properties. A mapping from the DisCo language into timed automata is introduced. Timed automata are finite-state machines extended with features to support the specification and verification of real-time behaviour. An existing model-checking tool is used for the verification task. Although DisCo is a more powerful model of computation than timed automata, many useful properties can be verified. Moreover, even for specifications that cannot be mapped as such, special cases can be used for more informal validation or finding counterexamples. The approach is illustrated by the mapping and verification of a simple example specification, in which two errors are found and corrected.*

1 Introduction

Specification is widely recognized to be the most crucial phase in the design of complex systems involving concurrency, external stimuli and real-time constraints. Errors made in the specification phase can be expensive to fix later in the development cycle. Three factors are of key importance in obtaining correct specifications: (1) The methodology must support the designer's thinking, guide to use proper abstractions, and enable incremental development. (2) Specifications must be validated by developers and users before continuing with implementation. (3) Critical properties of the specification need to be formally verified.

DisCo¹ [9] is a specification method based on joint actions [3,4]. Its main advantages are object-orientation, design methodology based on stepwise refinement by superposition, support for real-time specification, and executability. Based on past results and experiences [15,12,21], it can be argued that DisCo performs well with respect to factors (1) and (2) in

¹ Acronym for *Distributed Co*-operation.

the above list. The layer-based methodology and good support for high-level abstractions liberate the designer from thinking in terms of implementation-level concepts early in the specification process. Executability enables validation of specifications by animating them with the DisCo tool [21]. The tool has turned out to be a useful validation instrument and enhanced communication medium. The method has been successfully used to specify systems larger than typical academic examples; e.g. [19] describes a specification of a slightly simplified I²C bus.

In this article, we concentrate on factor (3), discussing automatic formal verification of properties of specifications given in an improved version the DisCo language [11,10]. We map instantiations of DisCo specifications into *timed automata* [2], a formalism suitable for the verification of systems that consist of communicating processes with real-time constraints. These automata are used for verifying properties of the specification with the model checking tool Kronos [22].

The subset of DisCo specifications that can be verified using the mapping presented here is, from the point of view of expressiveness, at most equivalent to the set of specifications that could be given directly with timed automata. The value of using DisCo as a specification front-end is in the layer-based methodology, genericity, and in the animation facility provided by the DisCo tool. Furthermore, properties of DisCo specifications not verifiable by finite-state model checking may be verified by theorem-proving methods. A mapping from DisCo into the input of a theorem-prover and techniques to assist in the verification of invariant properties are described in [14]. Model checking could also be used for verifying finite-state subsystems of infinite-state specifications. Thus, the two verification approaches complement each other.

The structure of this paper is as follows. In Section 2 we discuss some work related with ours mentioned in the literature. Sections 3 and 4 introduce DisCo and the Generalized Railroad Crossing example. Timed automata, Kronos and TCTL are introduced in Section 5. Section 6 describes the procedure for mapping a DisCo

specification into timed automata, and in Section 7, the actual verification using Kronos is done. In Section 8 we give some conclusions and discuss future research on the subject.

2 Related Work

References to model checking of action systems in the literature are few. Our work is partially based on an earlier mapping [1] from DisCo into a transition system formalism. A model checker for TLA⁺ specifications is described in [23]. The common base logic of DisCo and TLA⁺ makes this work interesting from our point of view. A translation from linear hybrid action systems into linear hybrid automata is presented in [20].

Kronos has been used in verifying specifications written in process algebras, e.g. ATP [18], and in an extension of synchronous language ARGOS [13]. To our knowledge, the mapping presented here is the first one from real-time action systems into timed automata.

3 DisCo

The DisCo method incorporates a specification language, a methodology for developing specifications using the language, and tool support for the methodology. The language is used to construct generic closed-system specifications where objects communicate in multi-object actions. A closed-system specification describes a system together with its environment. The formal basis of the language is *Temporal Logic of Actions (TLA)* [16].

Specifications are developed incrementally. One starts with very simple behaviours, and by *stepwise refinements* adds details until the specification is at the desired level. A refinement is obtained by applying a variant of *superposition* preserving safety properties.

The basic building blocks of the DisCo language are *layers*. A layer is an incomplete view of temporal (e.g. A and B happen by turns) and real-time properties (A happens every 20 ms) in the total system rather than an architectural unit. Layers serve as units of logical modularity. They can be built up from scratch, or from other layers, possibly reusable ones, by refinement or composition.

The most important components of layers are class and action definitions, initial conditions, and assertions. Objects are instances of classes. The attributes of an object can have simple values such as integer, real or truth values, or they can be sets or sequences of simple values. Additionally, the type *time* is available for specifying real-time properties. Objects can also include hierarchical and parallel finite state machines, and references to other objects. As with other object-oriented

languages, inheritance can be used for building more specialized classes from more abstract ones.

An action consists of a name, *roles*, *parameters*, a *guard*, and a *body*. Objects can *participate* in roles. An action is *enabled* if participants can be selected so that the guard expression evaluates to true for them. The execution model has no explicit control flow. Any enabled action can be executed: the choice is nondeterministic. The execution of an action means executing the statements given in its body.

An initial condition expresses a condition that must hold initially, and an assertion expresses a condition that is asserted to hold invariantly.

Real time is modelled as follows. It is assumed that actions are executed instantaneously, and that time will only pass between consecutive actions. There are two dedicated variables: Ω , the real-valued clock variable initialized as zero, and Δ , the global set of deadlines initialized as empty. The value of Ω grows monotonically and in each action, the value of an implicit parameter *now* indicates its execution moment. A deadline consists of a moment of time and a unique identity. Whenever a deadline $\text{now} + d$ is needed for some future action, an explicit statement of the form $t@d$, where t is a variable of type time, is given in an action body to add this deadline to Δ . The value of Ω is prevented to grow beyond this deadline, until an execution of an action with a statement $t@$ in its body has removed it from Δ .

4 Generalized Railroad Crossing

In the following sections, Generalized Railroad Crossing (GRC) [7], a well known benchmark problem, will serve as an example. The problem is to develop a system that operates a gate at a railroad crossing. The system should satisfy safety and utility properties, informally: the gate is down when there is a train in the crossing, and the gate is up when no train is in the crossing. Our solution consists of layers *Trains*, *Gates* and *Controllers*, where the latter refines a composition of the other two layers. The solution is generic, e.g. the specification allows an arbitrary number of trains. Because of space limitations we describe some parts of the specification only verbally.²

Trains Layer *Trains* defines class *Train* including a state machine and variable *exit_dl* (*dl* standing for *deadline*) of type time. The state machine depicts the position of the train with respect to the crossing as follows: In state *FAR*, the train is not in the proximity

² The full specification is available at URL <http://disco.cs.tut.fi/examples/grc/>.

of the crossing, in **NEAR** the train is approaching the crossing, in **AT_GATE** the train is in the crossing, and in **PAST** the train has passed the crossing. There is also an initial condition **Train_init** asserting that initially all trains are in state **FAR**:

```
class Train is
  state : (FAR, NEAR, AT_GATE, PAST);
  exit_dl : time;
end Train;

initially Train_init is  $\forall t : \text{Train} :: t.\text{state} = \text{FAR};$ 
```

The layer includes four actions. Action **approach** has one participant of class **Train**. The action is enabled for trains which are in state **FAR**. It changes the state of the participating object to **NEAR** and sets a deadline for action **exit** to 5 time units from **now**:

```
action approach (t : Train) is
when t.state = FAR do
  t.state -> NEAR();
  t.exit_dl @ 5;
end approach;
```

Action **in** (omitted) changes the state of the participating train from **NEAR** to **AT_GATE**. It can not occur within two time units after an occurrence of **approach**.

Action **out** changes the state of the participating train from **AT_GATE** to **PAST**:

```
action out (t : Train) is
when t.state = AT_GATE do
  t.state -> PAST();
end out;
```

Action **exit** changes the state of the participant from **PAST** to **FAR** and removes the deadline set in **approach**:

```
action exit (t : Train) is
when t.state = PAST do
  t.state -> FAR();
  t.exit_dl @;
end exit;
```

Gates Layer **Gates** introduces class **Gate**. The structure of the class is very similar to **Train**. The states of the included state machine are named **UP**, **GOING_DOWN**, **DOWN** and **GOING_UP**. We assert that the class is singleton, i.e., there is exactly one instance of it. Initially it must be in state **UP**. As in layer **Trains**, there are four actions that change the state of the state machine. The actions are named **lower**, **down**, **raise** and **up**. There are also some time restrictions. Execution of **lower** must be followed by **down** not later than one time unit, and **raise** by **up**, not earlier than one time unit, but not later than two time units. Actions **raise** and **up** are shown below. The execution moment of action **raise** is stored in variable **raise_ot** (**ot** standing for occurring time):

```
action raise (g : Gate) is
when g.state = DOWN do
  g.state -> GOING_UP();
  g.up_dl @ 2;
  g.raise_ot := now;
end raise;

action up (g : Gate) is
when g.state = GOING_UP and now >= g.raise_ot + 1 do
  g.state -> UP();
  g.up_dl @;
end up;
```

Controllers Layer **Controllers** imports layers **Trains** and **Gates** and refines their composition using superposition. Singleton class **Controller** is defined, which again contains a four-state state machine as well as three variables of type **time**, **lower_dl**, **raise_dl** and **approach_ot**. There is also variable **n_trains** of type **integer** which is a counter used to count the number of trains that are in the proximity of the crossing. The states of the controller are **WAIT_FIRST**, **TO_LOWER**, **WAIT_LAST** and **TO_RAISE**, the first of which is asserted to be the initial state.

Imported actions are refined using superposition. Actions **approach** and **exit** of **Trains**, and **lower** and **raise** of **Gates** are refined by adding a new participant of class **Controller** to each. Furthermore, their bodies are augmented by new statements which change the attributes of the new participant.

Action **approach** is refined so that if the controller is in state **WAIT_FIRST** or **TO_RAISE** the state is changed to **TO_LOWER** or **WAIT_LAST**, respectively. In the former case, a deadline is set for action **lower**. The action also increments the value of **n_trains** and stores the moment of execution in variable **approach_ot**. Ellipses are used in the language to denote the parts of the guard and the body given in the imported action:

```
refined approach (t: Train; c: Controller) of approach(t) is
when ... do
  ...
  if c.state = WAIT_FIRST then
    c.state -> TO_LOWER();
    c.lower_dl @ 1;
  elsif c.state = TO_RAISE then
    c.state -> WAIT_LAST();
  end if;
  c.n_trains := c.n_trains + 1;
  c.approach_ot := now;
end approach;
```

The guard of action **lower** (omitted) is strengthened so that the controller must be in state **TO_LOWER** and exactly one time unit has passed from occurrence of **approach** ($\text{now} = \text{c.approach_ot} + 1$). The state of the controller is changed to **WAIT_LAST**, and the deadline set in **approach** is removed.

Action **exit** is refined so that it always decrements the counter **n_trains**. Additionally, if the train that is exiting is the last one in the proximity of the crossing,

and the state of the controller is `WAIT_LAST`, the action changes the state of the controller to `TO_RAISE` and sets a deadline for action `raise`:

```

refined exit (t: Train; c: Controller) of exit(t) is
when ... do
  ...
  if c.n_trains = 1 and c.state'WAIT_LAST then
    c.state -> TO_RAISE();
    c.raise_dl @ 1;
  end if;
  c.n_trains := c.n_trains - 1;
end exit;

```

The guard of action `raise` is strengthened so that the controller must be in state `TO_RAISE`. The state of the controller is changed to `WAIT_FIRST` and the deadline set in `exit` is removed.

```

refined raise (g: Gate; c: Controller) of raise(g) is
when ... c.state'TO_RAISE do
  ...
  c.state -> WAIT_FIRST();
  c.raise_dl @;
end raise;

```

This concludes the specification.

5 Verification Theory and Tools

5.1 Timed Automata

Timed automata are finite-state machines extended with features for modelling of the behaviour of real-time systems over time. There are basically two different variations of timed automata, one of which is based on accepting *timed words* (e.g. [2]), while the other uses *location invariants* (e.g. [22]). The most essential difference between these two variations is that the former associates all timing constraints with state transitions, while the latter associates certain constraints with states (*locations*). Current tools mainly support the latter definition of timed automata, which is why we have chosen the location invariant model as our target formalism.

Background. There is a finite set \mathcal{X} of *clocks*. Each clock $x \in \mathcal{X}$ is associated with a non-negative real value $v(x)$. $\Psi_{\mathcal{X}}$ is a set of predicates over \mathcal{X} defined as a conjunction of atoms of the form $x \# c$ or $x - y \# c$, where $x, y \in \mathcal{X}$, $\# \in \{<, \leq, >, \geq, =\}$, and c is an integer constant. An *assignment* ρ is a function $\rho : \mathcal{X} \rightarrow \mathcal{X}^*$, where \mathcal{X}^* is $\mathcal{X} \cup \{0\}$. $v[\rho]$ denotes the valuation v' such that for all $x \in \mathcal{X}$, $v'(x) = v(\rho(x))$ if $\rho(x) \in \mathcal{X}$, otherwise $v'(x) = 0$. Informally this means that the value of each clock either stays the same or is assigned the value of some other clock or 0.

Definition 1. [22] A *timed automaton* \mathcal{M} is a six-tuple $\langle \mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{P} \rangle$:

1. \mathcal{S} is a finite set of *locations*, of which s_{init} is the initial location.
2. \mathcal{X} is a finite set of *clocks*.
3. \mathcal{L} is a finite set of *synchronization events*.
4. \mathcal{T} is a finite set of *edges*. Each edge t is a 5-tuple $\langle s, L, \psi, \rho, s' \rangle$:
 - (a) $s \in \mathcal{S}$ is the source location,
 - (b) $s' \in \mathcal{S}$ is the target,
 - (c) L is a set of events,
 - (d) $\psi \in \Psi_{\mathcal{X}}$ is the enabling condition, and
 - (e) $\rho : \mathcal{X} \rightarrow \mathcal{X}^*$ is the assignment.
5. $\mathcal{I} : \mathcal{S} \rightarrow \Psi_{\mathcal{X}}$ is a function that associates a condition $\mathcal{I}(s)$ to every location $s \in \mathcal{S}$. $\mathcal{I}(s)$ is called the *invariant* of s .
6. \mathcal{P} associates a set of atomic propositions with each location.

We will informally describe the semantics of timed automata. At any time, the state of an automaton is determined by the location and the values of the clocks. The latter have to be such that the location invariant is satisfied. The automaton may change its location and the values of some of the clocks by executing an enabled transition (i.e., a transition whose enabling condition evaluates to true). Alternatively, time may pass while the location remains unchanged, provided that the location invariant is not violated; the values of all clocks are incremented by an equal amount of time.

An individual timed automaton is typically used to model the behaviour of a process or component in a larger system. Component automata communicate by synchronizing in transitions labelled with elements of the sets of synchronization events. Therefore we need the *product automaton* or *parallel composition* of n automata, which is informally described as follows.

- The locations are n -tuples of locations.
- The set of clocks is the union of the n sets of the components.
- The set of propositions associated with a tuple location is the union of the sets of propositions of the component locations.
- The invariant of a tuple location is the conjunction of the component invariants.
- Transitions in different component automata labelled with the same events are synchronized, internal transitions of the components may be executed independently.
- The enabling condition of a transition is the conjunction of the enabling conjunctions of the participating transitions.
- The assignment is the union of the assignments of the participating transitions.

5.2 Kronos and TCTL

Kronos [22] is a verification tool for real-time systems. The mathematical background of Kronos is based on the theory of timed automata [2] and timed temporal logics.

Verification with Kronos can be done by using a *model checking* or *behavioural* approach. The correctness criteria is expressed in TCTL [8] when the model checking approach is used and another automaton is used for the behavioural verification. In this paper we concentrate on model checking.

The infinity problem induced by dense time is solved in Kronos by symbolic representation of the infinite state space by sets of linear constraints. The infinite state space is discretized and thus model checking can be done as if the automaton was finite.

TCTL is an extension of CTL [5]. TCTL defines two real-time temporal operators: $\exists\phi_1\mathcal{U}_I\phi_2$ and $\forall\phi_1\mathcal{U}_I\phi_2$, informally *possibly* and *inevitably*. A reset quantifier for clocks is defined. Intuitively $\exists\phi_1\mathcal{U}_I\phi_2$ means that there exists some non-zero run ρ starting from state s and a point along the run such that the time spent until that point belongs to interval I , ϕ_2 holds at that point and ϕ_1 holds continuously until that point.

Conventional arithmetic, boolean and temporal operators (like $=, \wedge, \Rightarrow, \forall\Diamond_I$) are also defined. The meaning of $\forall\Diamond_I\phi$ is $\forall true\mathcal{U}_I\phi$.

6 Mapping DisCo into Timed Automata

6.1 Strategy

This section presents a formal procedure for mapping a DisCo specification into timed automata. Each object of an instance of a specification is mapped into a timed automaton, and the actions become the transitions. In addition, real-time constraints are handled separately by constructing an automaton for each time-typed variable of the specification.

The mapping is illustrated in Figure 1. The first box depicts a DisCo specification and the second contains the automata produced by the mapping.

Before the actual mapping, the actions usually need to be replaced by simpler ones using a transformation procedure summarized in the next subsection.

We do not verify the generic system that a DisCo specification actually describes, but a specific instance of it. Furthermore, we are only able to verify finite-state systems. Thus, before beginning, we need to do the following:

1. Choose an instantiation that is to be verified.
2. Set any necessary limitations on data values. Variables and parameters must have finite value ranges.

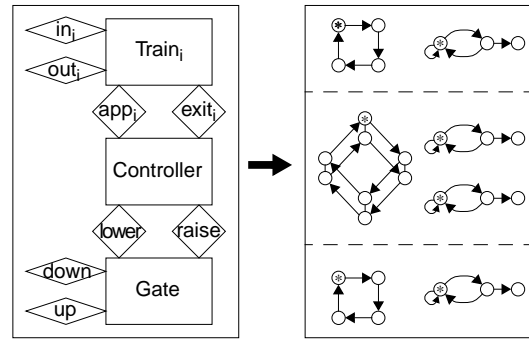


Fig. 1. Scheme of the procedure for mapping DisCo into timed automata.

In this case, we choose an instantiation that has two trains.

For simplicity it is assumed that only basic classes and other core features of the DisCo language are used. Advanced features such as inheritance and complex expressions are not handled by the mapping. These are not essential limitations because these constructs can be preprocessed by the DisCo compiler to obtain an inheritance-free intermediate form utilizing only basic expressions. Non-operational features of DisCo (assertions and initial conditions) affect the transformation only indirectly by constraining the initial state.

6.2 Transforming the Specification into a Mappable Form

In timed automata, the synchronization of component automata in a transition is solely based on the label of the transition. However, in DisCo actions are actually templates for simpler actions; e.g. the participants can in the general case be chosen in many different ways. Thus, to be able to map DisCo actions into transitions of timed automata, we need to split the template actions of a DisCo specification into ones whose label (i.e. action name) is unique for each combination of participants, and moreover, unique for each combination of participant states.

The actions are transformed in the following way:

1. Transform actions with nondeterministic parameters into several actions with fixed parameter values (one action for each possible combination of values).
2. Split each action into as many versions as there are possible participant combinations for it. Each object must be given an explicit identity field which is then checked in the guard.
3. Split actions to remove statements that are not locally determined, i.e. statements that assign to an

object a value depending on the state of another object.

4. Make the guards separable. An example of a non-separable guard is $o.b = p.b$, where o and p are objects and b is a boolean-valued variable of both. An equivalent separable form is $o.b = false \wedge p.b = false \vee o.b = true \wedge p.b = true$.
5. Transform all action guards into their disjunctive normal forms. Break the actions into as many versions as there are disjuncts, taking the disjuncts as the guards of the new actions.

Because of space limitations, we cannot describe the transformation steps in more detail here. Only some of the steps need to be carried out for the example specification. Full descriptions of all steps together with proofs that they result in an equivalent specification are given in [1].³

6.3 Mapping Data

Symbolic representation of data in timed automata is not possible. Therefore, variables have to be *unfolded*. Unfolding refers to embedding different variable values into different states. In order to be able to use real-life verification tools, the automata need to be finite and of relatively small size. In the GRC specification, the only variable whose value needs to be restricted is `n_trains`, of type integer. Since our instantiation has exactly two trains, we know that `n_trains` is never *supposed* to have any other values than 0, 1 or 2. However, in order not to risk making false assumptions, we should *prove* that the mentioned restricted value range is indeed entailed by the specification.

Fortunately, the proofs of the validity of this kind of instantiation-implied restrictions on value ranges can be left on the responsibility of the model checking tool used to verify the actual properties of the specification. We just have to make an educated guess of what the range will be, and add certain transitions to a special error state in the automaton. This will be discussed in more detail in Sections 6.5 and 7.1.

6.4 Limitations of the Mapping

Real time Due to the limitations of timed automata, only certain kinds of real-time specifications can be mapped. We do not claim that a mapping could not be found for some real-time features mentioned below as unsupported. We have chosen to support an expressive enough set of features while trying to keep the

³ The target formalism in the reference is not timed automata, but more traditional transition systems. The transformation steps, however, are not affected by the differences of the formalisms.

mapping relatively simple and straightforward. In our experience, most real-time DisCo specifications naturally adhere to these limitations, and those that do not can relatively easily be modified to do so. For the non-obvious, non-syntactic restrictions, there is no need to investigate whether the specification is in accordance with them before producing the automata: the model checker can be used to do this after the mapping has been applied.

The first limitation is that only integer literals can be used to denote time values, possibly adding the value of `now` in certain situations. This is because timed automata allow only integer assignments and comparisons. Restricting to integers does not reduce expressiveness that much, because time units can always be scaled appropriately, and there very rarely is need to use time values that cannot be expressed with a terminating decimal. Furthermore, values of time variables must not be assigned to other time variables. This could not be handled by timed automata. In DisCo, time values usually have the nature of global specification abstractions, and therefore they need not be assigned to other variables, thus the limitation is not essential. Also, if a deadline is set using a certain time variable, it must be removed before another deadline can be set using the same variable. It is a good idea to adhere to this limitation anyway, or the specification is very likely to be incorrect. Finally, only one assignment statement per time variable is allowed in the specification (this is a static restriction, i.e. dynamically the statement can of course be executed an arbitrary number of times). Without this restriction the handling of comparison expressions would become difficult. The limitation can be circumvented by using more time variables.

Fairness DisCo allows strong fairness requirements in actions, while timed automata have no notion of fairness in their execution model. Thus we are only able to verify safety and real-time properties, which fairness does not affect. This limitation is not too essential, because in real-time specifications we can often use *bounded liveness* expressed using real-time constraints instead.

6.5 Constructing Timed Automata Modelling Objects

We are now ready to do the actual mapping. First we construct automata without any timing properties, i.e. deadline manipulation statements and other references to time-typed variables are ignored altogether at this stage.

In the mapping, we will use the notational convention that the assignment component of an edge is given

as a set of assignment statements. The semantics of this shorthand notation is obvious: the corresponding function maps each clock occurring on the left hand side of an assignment to the right hand side of the assignment, and all other clocks to themselves.

We denote the automaton representing the DisCo object O by $TA(O)$. The construction of the set of states for $TA(O)$ is done in two steps. First, we take as *underlying states* the state space of the hierarchical and parallel DisCo state machines in the object and denote them by S' . States in S' are not parallel or hierarchical; the state space has been formed by binding a unique state to each possible combination of hierarchical and parallel states. Second, unfold the variables, resulting in the states

$$\mathcal{S} = \{ (s', v_1, v_2, \dots, v_n) \in S' \times R'_1 \times \dots \times R'_n \mid \forall 1 \leq i \leq n : \text{if } \mathcal{E}(x_i, s') \text{ then } v_i \in R_i \text{ else } v_i = \lambda \}.$$

where x_1, x_2, \dots, x_n are the variables of the object, R_i is the value range of variable x_i , $R' = R \cup \{\lambda\}$ ($\lambda \notin R_i$), and $\mathcal{E}(x, s)$ is *true* iff variable x is defined in state s (in the DisCo sense, i.e. the scope of the variable is active in state s).

The initial state s_{init} of $TA(O)$ is determined by the initial state of the DisCo specification, obtained in the instantiation, and constrained by the assertions and initial conditions of the specification.

Additionally, a special $error_{TA(O)}$ state is added to the automaton. The subscript $TA(O)$ stands for a unique identifier making the name of the *error* state local to the automaton.

The set of synchronizing events \mathcal{L} of $TA(O)$ is

$$\mathcal{L} = \{ A \mid A \text{ is an action } O \text{ participates in } \}.$$

The set of edges \mathcal{T} of $TA(O)$ is

$$\mathcal{T} = \{ (s, \{A\}, true, \emptyset, s') \in \mathcal{S} \times 2^{\mathcal{L}} \times \{true\} \times \{\emptyset\} \times \mathcal{S} \mid s \langle \langle A \rangle \rangle s' \}$$

where $s \langle \langle A \rangle \rangle s'$ means that

1. all predicates except those referring to time variables in the guard of A depending on the state of object O evaluate to *true* in state s , and
2. executing all statements of A affecting the state of O — except those assigning to time variables — in state s results in state s' .

If s' is a state that does not exist due to a restriction placed on the range of some variable(s), the target state is $error_{TA(O)}$.

The set of clocks \mathcal{X} is empty and the function \mathcal{I} associates the condition *true* to each location. We also

choose a \mathcal{P} component to associate a set of atomic propositions with each location; we use propositions such as AT_GATE_i and FAR_2 which in this case directly correspond to the states of the objects in the DisCo specification.

6.6 Constructing Timed Automata Modelling Real-Time Constraints

For each time-typed variable x of the specification, we construct a separate *constraint automaton* $CA(x)$. The initial location of the automaton has no invariant, and possible other locations have invariants that correspond to DisCo deadlines. There are basically four ways of using time-typed variables in DisCo specifications, and they are treated as described below.

Setting a Deadline Because of the limitations of timed automata, we have to place restrictions on how time variables may be used in the specification. All deadline set statements must be of the form

$$o.dl @ c;$$

where $o.dl$ is a time-typed variable and c is a non-negative integer literal. For each statement t of action $A(t)$ of the above form, a location $Loc(t)$ with the invariant $o.dl \leq c$ and an edge $\langle s_{init}, \{A(t)\}, cond, \{o.dl := 0\}, Loc(t) \rangle$ are added to the automaton $CA(o.dl)$. If the guard of $A(t)$ does not depend on $o.dl$, $cond$ is *true*, otherwise $cond$ is as described below in the context of using a time variable in a guard.

In addition, edges $\langle s, \{A(t)\}, cond, \emptyset, error_{o.dl} \rangle$ are added for each location $s \neq s_{init}$. Location $error_{o.dl}$ is a special location which is not reachable in the final product automaton if the specification does not allow using the same time variable for setting a new deadline without removing the previous one set using the same variable. The unreachability of the location must be verified. The *error* location is local to each constraint automaton; therefore we add the variable name as a subscript.

Removing a Deadline A deadline removal statement is always of the form

$$o.dl @;$$

For each statement t of action $A(t)$ of the above form, edges $\langle s, \{A(t)\}, cond, \emptyset, s_{init} \rangle$ for every $s \in S$ are added to the automaton $CA(o.dl)$. Component $cond$ is as in using a time variable in a guard below.

Assigning a Value to a Time Variable We allow the following form of assignments to time variables:

$$o.tv := now \pm c;$$

where c is a non-negative integer literal. Only one such assignment statement per time variable is allowed in the specification. For each statement t of action $A(t)$ of the above form, edge $\langle s_{init}, \{A(t)\}, cond, \{o.tv := 0\}, s_{init} \rangle$ is added to automaton $CA(o.tv)$. Component $cond$ is as in using a time variable guard (see below). Additionally, edges $\langle s, A(t), true, \emptyset, error_{o.tv} \rangle$ for each $s \neq s_{init}$ are added to the automaton $CA(o.tv)$. This entails that it is illegal to assign to a deadline variable with an active deadline. Again, we have to verify that $error_{o.tv}$ is an unreachable state.

Using a Time Variable in a Guard We allow the following forms of expressions referring to time variables in action guards:

now # $o.tv \pm d$

where $o.tv$ is the time variable, $\# \in \{<, \leq, >, \geq, =\}$ and d is a non-negative integer literal. If action A containing the expression does not set or reset deadlines, edges $\langle s, \{A\}, cond, \emptyset, s \rangle$ for all locations $s \in S$ are added to $CA(o.tv)$. By the following calculation we get the expression $cond$. There may be one assignment to the time variable, and it has to be of the form

$o.tv := now \pm c;$

where c is a non-negative integer literal. We obtain $cond$ by adding the constant component $\pm c$ (where the binary operator \pm is interpreted as the sign of c) of the possible assignment (0 if there is no such assignment) to the constant term $\pm d$ of the comparison expression, obtaining k , and comparing the clock $o.tv$ corresponding to the time variable $o.tv$ with k . Thus, $cond$ becomes

$o.tv \# k.$

If, on the other hand, action A does set or reset deadlines, the edges needed are already in the automaton. We just add the $cond$ component (obtained the same way as above) to them.

7 Verification Using Kronos

7.1 Mapping GRC

As a result of applying the action transformations and the mapping procedure described in Section 6 to the two-train instantiation of the GRC specification presented in Section 4, we obtain 14 timed automata. Four of these correspond to DisCo objects (two Trains, one Gate, and one Controller), and ten to time variables of the specification (one automaton per variable). As examples, the automata corresponding to the Controller object and time variable $raise_dl$ are shown in Figures 2 and 3.

For verification, the produced automata are composed in parallel. The product automaton has 68 locations, 199 transitions, and ten clocks. The number of

clocks could have easily been reduced, and in larger specifications optimization is essential due to performance reasons. Reduction could be done either on the DisCo level (by using fewer time variables) or by using a clock optimization tool such as *Optikron* [6] after the mapping has been applied.

At this stage, we must check that *error* states are unreachable. We find that they are. Thus, the assumptions made in the mapping — regarding the range of the variable n_trains and the manipulation of time variables — are valid.

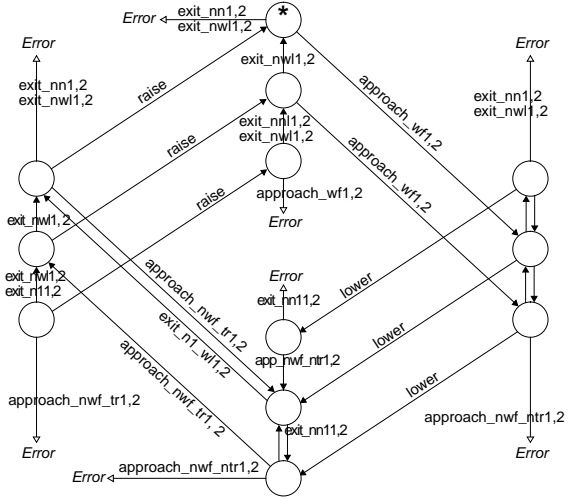


Fig. 2. The Controller automaton.



Fig. 3. The Raise_dl automaton.

7.2 Non-Zenoness

Non-Zenoness, the property that time can proceed beyond any bound, can be verified by verifying the TCTL property

$$init \Rightarrow \forall \square (\exists \diamond_{=1} true).$$

Intuitively the property means that it is always possible for time to proceed by one unit. The result that this

property entails Non-Zenoness has been proved in [8]. Non-Zenoness is an important property of any real-time system. If it does not hold, there probably are conflicting real-time requirements. In such a situation it might be that some safety properties only hold because time stops and nothing happens.

Applying the forward search of Kronos, we find that our specification does *not* satisfy Non-Zenoness. A counterexample is obtained which leads us to notice that the assignment

```
c.approach_ot := now;
```

in action `approach` should not be executed in all situations. Actually, it should only be executed when the *first* train approaches the crossing, i.e. not when one train is already in and a second train approaches. The error leads to a situation in which on one hand it is required that the deadline for executing action `lower` is one time unit after executing `approach` for the *first* time, and on the other hand `lower` is required to be executed exactly one time unit after executing `approach` for the *second* time. These requirements clearly conflict with each other, and a Zeno behaviour is forced.

After correcting the bug, verification of Non-Zenoness is re-attempted. The result is, again, failure. The forward search algorithm yields a counterexample based on which we note the following. In the action `exit`, a deadline is set for action `raise`. However, if action `approach` occurs before `raise`, we are in trouble. The deadline is not removed, yet `raise` is not enabled anymore. Thus, a Zeno behaviour is again forced.

After the correction of the second error, Non-Zenoness is verified successfully using the backward search of Kronos.

7.3 Safety

The safety property “*when a train is in the crossing, the gate is closed*” is expressed in TCTL as follows:

$$\begin{aligned} \text{init} \Rightarrow \forall \square (& (AT_GATE_1 \vee AT_GATE_2) \\ & \Rightarrow DOWN) \end{aligned}$$

Using the backward search algorithm, we find that the property holds for the corrected specification.

7.4 Utility

The informal utility property “*whenever no train is in the crossing, the gate is up*” is too strong to be taken literally. The system has to be allowed some time after the last train has left the crossing before it can reasonably be required that the gate has been lifted. The easiest way to formulate this property is to add an auxiliary

clock `EXIT_TIMER` which is reset when the last train leaves the crossing, and require that

$$\begin{aligned} \text{init} \Rightarrow \forall \square (& (FAR_1 \wedge FAR_2) \\ & \Rightarrow (UP \vee EXIT_TIMER \leq 3)) \end{aligned}$$

The time bound 3 is obtained by summing up the deadline intervals for actions `raise` (1 time unit after `exit`) and `up` (2 time units after `raise`).

Using the backward search of Kronos, the utility property is verified successfully for our corrected specification.

8 Conclusions and Future Work

We have presented an approach to the verification of real-time DisCo specifications. The main technical contribution of this work is the mapping from DisCo specifications into timed automata. Using the model checking tool Kronos we are able to verify many critical properties of finite specification instances. A compiler could be constructed in a straightforward way to implement the mapping presented in this paper.

The approach makes some use of the genericity of DisCo specifications: unlike in many other verification approaches based on model checking, one specification suffices for producing many different instantiations consisting e.g. of different numbers of objects.

The two errors found in the example specification are evidence of the usefulness of the mapping. When we tried to verify the Non-Zenoness property, Kronos output a sequence of transitions leading to a state not satisfying the property. Counterexamples of this kind act as clues for locating errors. Using the erroneous sequence of transitions, it was easy to trace the error in the DisCo specification.

The model checking approach suits well for certain kinds of DisCo specifications and certain kinds of properties. As described in Section 6, the limitations for the specifications are not crucial. It is also possible to model check only some instantiations of parts of the DisCo specification. Liveness properties cannot be verified with the approach. The closest we get are *bounded liveness* properties expressed using timing constraints.

In large specifications, we easily end up with a large state-space and a substantial number of clocks even after optimization. The current tools are not yet able to handle large state spaces efficiently, and especially the number of clocks quickly increases time and memory usage. Since DisCo supports stepwise refinement and preservation of safety properties, we may be able to verify some safety properties using simple layers that describe the system at a higher level of abstraction. Another way to ease the work of the verification tool is

made possible by the separate treatment given for real time in the mapping. Properties that do not depend on real-time constraints could be verified without including the constraint automata in the product system.

We also explored the possibility to use another tool for model checking timed automata. The tool UPPAAL [17] has appealing features such as data variables and a graphical user interface. However, in UPPAAL communication between processes is one-to-one. This would make the mapping of multi-object actions less straightforward since the synchronization would have to be done pairwise.

Model checking and user-controlled mechanical theorem proving complement each other. Model checking can be used to find counterexamples efficiently and proposed invariants can be pre-checked for specific instances before attempting to use a theorem prover for formally proving them for the generic specification.

References

1. Timo Aaltonen and Risto Pitkänen. Verifying safety by combining joint actions with a process-algebraic approach. Technical Report 19, Software Systems Laboratory, Tampere University of Technology, 1999. Available at URL <http://disco.cs.tut.fi/reports/vscjapaa.ps>.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.
4. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.
5. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, number 131 in Lecture Notes in Computer Science. Springer-Verlag, 1981.
6. Conrado Daws. Optikron: a tool suite for enhancing model-checking of real-time systems. In *Proceedings of the 10th Conference on Computer-Aided Verification*, number 1427 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
7. Constance Heitmeyer and Dino Mandrioli (eds.). *Formal Methods for Real-Time Computing*. Number 5 in Trends in Software. John Wiley and Sons, Chichester, U.K., 1996.
8. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
9. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.
10. Hannu-Matti Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.
11. Hannu-Matti Järvinen and Reino Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.
12. Hannu-Matti Järvinen, Reino Kurki-Suonio, and Kari Systä. Experiences in object-oriented modeling with multi-object actions (invited lecture). In H. Kilov and B. Harvey, editors, *Proceedings of the OOPSLA'93 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 61–66, Robert Morris College, 1993.
13. M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *Fifth International Workshop on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science. Springer-Verlag, 1993.
14. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 589–604. Springer-Verlag, 1997.
15. Reino Kurki-Suonio and Tommi Mikkonen. Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, pages 195–199. Springer-Verlag, 1998.
16. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
17. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.
18. X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, September 1992.
19. Risto Pitkänen and Harri Klapuri. Incremental cospecification using objects and joint actions. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume VI, pages 2961–2967. CSREA Press, June 1999.
20. Mauno Rönkkö and Xuandong Li. Linear hybrid action systems. Technical Report 245, Turku Centre for Computer Science, 1999. Available at URL <http://www.tucs.fi/publications/techreports/TR245.html>.
21. Kari Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.
22. Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.
23. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, number 1703 in Lecture Notes in Computer Science, pages 54–66. Springer-Verlag, 1999.