Constructing Test Automata from Graphical Real-Time Requirements

Henning Dierks[1] and Marc Lettrari[2]

[1] University of Oldenburg, Department of Computer Science,
P.O.Box 2503, 26111 Oldenburg, Germany
dierks@informatik.uni-oldenburg.de
[2] OFFIS, Escherweg 2, D-26121 Oldenburg, Germany
lettrari@informatik.uni-oldenburg.de

Abstract. A semantics for a graphical specification language of real-time properties (Constraint Diagrams) is presented. The new semantics is given in terms of Timed Automata. A model in terms of Timed Automata satisfies the property given by a Constraint Diagram if the model in parallel composition with the semantics of the Constraint Diagram can reach a certain state. This kind of question can be checked by all model-checkers for Timed Automata. A prototype of a tool is presented that automatically translates an appropriate Constraint Diagram into the input language of the tool Uppaal.

1 Introduction

Whenever continuous time is necessary to model a real-time system, Timed Automata [AD90,AD94] are usually applied. The advantage of this model is the existence of decision procedures which have been successfully implemented by model-checking tools like Uppaal [LPW97] and Kronos [Yov97]. In principle, it is decidable whether a system of Timed Automata satisfies a formula given in TCTL (timed computation tree logic, [ACD90,HNSY94]). However, reachability properties can be verified more efficiently.

One of the main obstacles to verify formally a real-time system is to specify the desired properties correctly. In the case of reachability this task is not difficult. However, many properties of interest are not simple reachability questions. Then the user has the choice between these possibilities:

- To express the property in TCTL formulas or
- to build a *test automaton* that serves as a wrapper for the property, i.e. the test automaton reaches a certain state if and only if the property is (is not resp.) satisfied.

The disadvantage of the first method is that it is error-prone in the sense of a mismatch between the property expressed by the formula and the property in mind of the specifier. The disadvantage of the second method is that the construction of a test automaton is also error-prone.

W. Damm and E.-R. Olderog (Eds.): FTRTFT 2002, LNCS 2469, pp. 433–453, 2002.

In this paper we propose the following approach. So-called "Constraint Diagrams" (CDs for short) have been proposed in [Die96,Kle00] as a graphical specification language for real-time properties. We present a *test automaton semantics* for CDs that is equivalent (in an appropriate sense) to the semantics given by [Die96,Kle00] in terms of Duration Calculus [ZHR91,HZ97]. All details about this equivalence including the proof can be found in [Let00][1]. The reasons for choosing CDs are

- the graphical oriented syntax (in assumption/commitment style) which employs sequences of phases to specify the behaviour and
- the formal semantics which is tailored to meet the human intuition.

The decision to construct test automata from CDs is driven by the demand for wide range of applicability. This is guaranteed by test automata because reachability tests are implemented in all available tools for Timed Automata.

The test automaton T that is the semantics of a Constraint Diagram C has a particular state q_{bad}. A model M in terms of Timed Automata satisfies the property expressed by C if and only if q_{bad} is not reachable for $M \parallel T$.

Related Work: A similar graphical language for requirements' capture are Symbolic Timing Diagrams [Sch01] (STD) and its real-time extension RTSTD [FJ97]. One of the main differences between CDs and (RT)STDs is that the latter do not allow commitments in the past.

Constructing test automata is not a new idea (e.g. [HLR93]) and has also been applied to Timed Automata [ABL98,LPW98]. The latter approaches construct test automata from formulas in temporal logics to overcome the lack of full TCTL in Uppaal. As discussed above these approaches still have the risk of a mismatch between the property in mind and the property written as formula. Our approach tries to minimise this risk by choosing CDs which are designed to be more readable and accessible for non-experts in temporal logics.

Structure: The paper is organised as follows: In Sect. 2 we introduce a running example, the well-known case study "Generalised Railroad Crossing". Properties of this case study are specified by CDs in Sect. 3 in order to explain the syntax and semantics briefly. In Sect. 4 we present the main contribution of this paper, namely the test automaton semantics for a subset of CDs. The prototypic implementation of a tool is explained in Sect. 5. Before we conclude we apply our result again to the running example in Sect. 6. Note that this paper concentrates on the description of the test automaton semantics and its application. Another issue is to establish the correspondence between this new semantics and the Duration Calculus semantics given in [Kle00]. This problem requires a relation between the semantical basis of Timed Automata (ie. timed traces) and the semantical basis of Duration Calculus (ie. function with domain $\mathbb{R}_{\geq 0}$. In [Die99,DFMV98a] such a relation can be found. On the basis of this relation equivalence proofs were worked out in [Let00] and cannot be presented here due to space limitations.

[1] Appendix A contains brief introductions to both DC and the DC semantics for CDs.

2 The Generalised Railroad Crossing

Our approach is illustrated by a case study. The problem of specifying a generalised railroad crossing (GRC) and verifying its properties was posed in [HL94] and suggested as a benchmark problem in comparing the suitability of different formal methods in [HM96], where many solutions are given. In [DD97] both the behaviour of the system and its properties were specified by CDs and an implementation was given in terms of PLC-Automata [Die99]. The question whether the implementation meets the specification was answered by semantical arguments on the basis of the temporal logic Duration Calculus [HZ97,ZHR91]. With the approach presented in this paper, the verification of CDs can be done automatically.

The description of the GRC in [HL94] is:

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e. $I \subseteq R$. A set of trains travels through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set λ_i of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have trains that entered the crossing after τ_i).
> Given two constants ξ_1 and ξ_2, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property:** $t \in \cup_i \lambda_i \Longrightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)
>
> **Utility Property:** $t \notin [\tau_i - \xi_1, \nu_i + \xi_2] \Longrightarrow g(t) = 90$ (The gate is up when no train is in the crossing.)

3 Constraint Diagrams

Constraint Diagrams (CDs for short) have been introduced in [Die96] as a graphical description language for real-time requirements. The motivation stems from the graphical notion of timing diagrams that are often used to specify properties of hardware designs. The idea of CDs is a representation of intervals in an assumption/commitment style. In Fig. 1 an example of a simple CD is given.

This CD can be read as follows: It constrains the behaviour of two Boolean variables called S and A. For all (prefixes of infinite) computations that satisfy the assumptions the commitments have to be fulfilled. The assumptions are the unboxed entities in Fig. 1. They specify that there is an interval with a length of 10 time units where the Boolean variable S is false. The commitments are

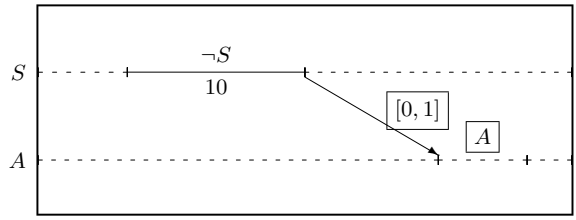


Fig. 1. Specification of a watchdog.

the boxed entities which require the Boolean variable A to be true after the $\neg S$ interval within 1 time unit. Hence, the CD in Fig. 1 specifies that whenever we observe an interval of length 10 where $\neg S$ holds, then the variable A ("alarm") should become true within at most one time unit.

The benefit of CDs are both the accessibility and the readability for non-experts in comparison to formulas given in temporal logic. The following examples of CDs specify the behaviour of trains in the case study GRC. The track under consideration in this case study (i.e. region R) can be empty ("E" for empty), at least one train is approaching the crossing and no train is crossing (i.e. no train is in I) ("A" for approaching), or at least one train is crossing (Cr for "crossing"). The CDs in Fig. 2 specify the following properties of the track:

a) Initially the track is empty;
b) if the track is empty, a train may approach; approaching trains need at least ε_1 time units to reach the crossing;
c) if A holds, a train may cross (Cr).

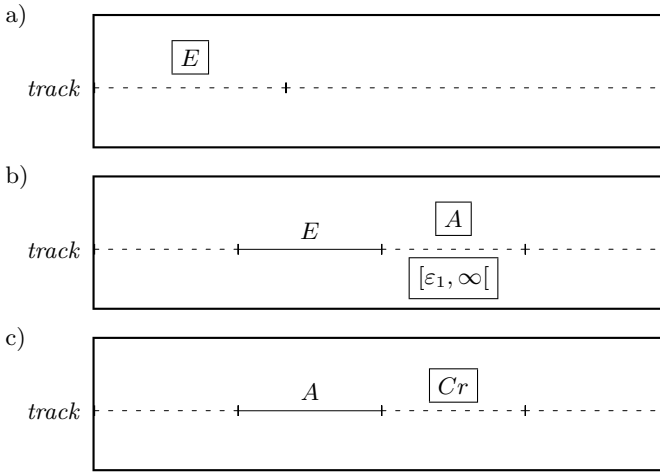


Fig. 2. Specification of track behaviour.

The first CD of Fig. 2 consists of two intervals with no assumptions at all which is indicated by the dashed lines. The commitment is that during the first

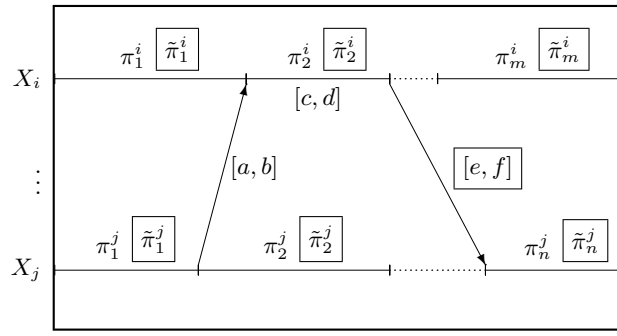


Fig. 3. General syntax of CDs.

interval the track has to be empty. The semantics of this CD basically requires the following: "Each finite prefix of an observation of the track which can be split into two intervals, can be split into two intervals with an empty track during the first interval."

The second CD of Fig. 2 assumes four intervals with the assumption that the track is in E during the second period. The commitment requires state A during the third interval if the track leaves state E. This has to hold for at least ε_1 time units if the track leaves state E. Note that the semantics of CDs always require just a *prefix* of the commitments[2]. In the case of the second CD this would also allow the track to remain E forever.

The last CD of Fig. 2 requires all A phases to be succeeded by Cr phases.

We discuss the CD of Fig. 3 to explain more generally syntax and semantics of CDs. Note that the CDs presented above contain some notational abbreviations which are explained later in this chapter. Figure 3 presents the general syntax of CDs.

A CD consists of a number of lines where each line symbolises the behaviour of a finitely typed variable. Each line is split into some phases[3]. Each phase carries two annotations:

[2] The idea of accepting also prefixes of commitments does not seem to be natural at first sight. However, without those prefixes the semantics of CD would not meet the intuitive meaning of the drawings. Consider the watchdog in Fig. 1 again. If a prefix of a computation with length 11 is observed where $\neg S$ was true within the interval $[0.5, 10.5]$, then we would not expected that this computation violates the CD because the alarm still might happen in time, namely within $]11, 11.5]$. Hence, as long as a computation τ can be *extended* in such a way that the extended computation satisfies the commitments of a CD, then τ satisfies the CD.

[3] If the duration of a phase is not specified, it is possible that the duration is 0. This fits to the Timed Automaton model where several transitions may happen at the same time point. This is sometimes called "two-dimensional time". If this is not desired, it is possible for all CDs presented in this paper to specify explictly that the duration of a phase is greater than 0 (cf. Subsect. 4.5).

- The unboxed annotation is a state assertion speaking about the variable of that line. This assertion represents the *assumption* for that phase. E.g. we assume that X_i satisfies initially π_1^i.
- The boxed annotation is a state assertion speaking about the variable of that line. This assertion represents the *commitment* for that phase. E.g. we require that X_j satisfies initially $\tilde{\pi}_1^j$.

Moreover, arrows between borders of phases are allowed which carry also two annotations:

- The unboxed annotation is an interval that represents the *assumed* time distance between the given events. E.g. we assume that the initial phase of X_i holds longer than the initial phase of X_j. The time difference is in $[a, b]$.
- The boxed annotation is an interval that represents the *required* time distance between the given events. E.g. we require that the second phase of X_i ends earlier than the beginning of the last phase of X_j. The time difference has to be in $[e, f]$.

The meaning of the CD is: For all computations and for all $t \in \mathbb{R}_{\geq 0}$ holds: Whenever the computation behaves within period $[0, t]$ in a way such that the variables evolve as given by the π_k^i assertions and the unboxed intervals, then it is possible to find a partition of $[0, t]$ such that a prefix of the commitments ($\tilde{\pi}_k^i$ and the boxed intervals) is satisfied. An introduction of the formal semantics employing Duration Calculus can be found in App. A.

Several abbreviations have been introduced for notational convenience. Those we need in this paper are the following: If a state assertion π_k^i is equivalent to true, then π_k^i is omitted and a dashed line is drawn instead of a solid line. If a state assertion $\tilde{\pi}_k^i$ is equivalent to true, it can be omitted. If an interval annotation of an arrow is $[0, \infty[$, it can be omitted. Intervals of the form $[r, r]$ are abbreviated by r. If an arrow restricts the beginning and the end of the same phase, the arrow itself is not drawn and the interval annotations are given as additional annotations of the phase. E.g. the $\neg S$ phase in Fig. 1 is assumed to last 10 time units. The arrow starting at the beginning and ending at the end of the $\neg S$ phase is not drawn. The time interval $[10, 10]$ is written as 10 and annotated to the phase instead of the arrow.

4 Test Automaton Semantics for Constraint Diagrams

Due to the expressiveness of Constraint Diagrams we cannot expect to be able to assign a test automaton semantics for all CDs[4]. Hence, we look for interesting subsets of CDs for which a test automaton semantics can be assigned. In [Let00] several subsets of CDs have been identified for which a test automaton semantics can be given.

[4] Appendix C presents an example for which no test automaton semantics can be found that uses only finitely many clocks.

4.1 Requirements for the Model

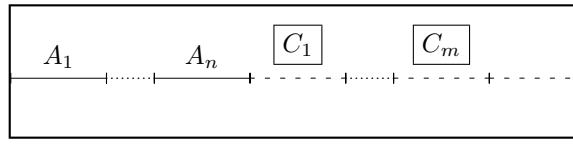
The test automaton semantics $\mathcal{T}(\mathcal{C})$ for a CD $\mathcal{C}$ is constructed such that a model M in terms of Timed Automata satisfies the property expressed by C if and only if state q_{bad} of $\mathcal{T}(\mathcal{C})$ is not reachable for $M \parallel \mathcal{T}(\mathcal{C})$[5]. However, this holds only if the model M satisfies the following requirements:

- Initially M sets all information, i.e. at time 0 a synchronisation with the test automaton happens that signals the initial values.
- All changes of the relevant variables for the CD have to be signalled to the test automaton via synchronisation. Due to the construction of the latter it is allowed to execute stutter synchronisations, i.e. to signal the same values subsequently.

By this it is clear what it means that a system satisfies a given state assertion π over a variable X: If the last synchronisation of M for variable X signals a change of the value of X to the value σ, then the system satisfies currently π iff σ satisfies π (in symbols: $\sigma \models \pi$).

4.2 Future Commitments

The following kind of CDs requires the system to ensure a sequence of C_i phases after the occurrence of a sequence of A_j phases. In other words: When the system is engaged in a computation that satisfies the assertions $A_1, A_2, \ldots, A_n$ in this order, then the system should continue the computation such that it will also satisfy the commitments $C_1, C_2, \ldots, C_m$ in this order. The corresponding CD is given below.



where $n, m \geq 1$, $A_n \neq \mathsf{true}$, $C_i \neq \mathsf{true}$ for all $1 \leq i \leq m$, A_n and C_1 are disjoint ($A_n \implies \neg C_1$), and all neighbouring commitments are disjoint ($C_i \implies \neg C_{i+1}$ with $1 \leq i < m$). We call the set of all CDs of this kind CD_1.

The ideas of the construction of this test automaton are simple. We expect the system to keep the test automaton informed about all relevant changes of the variable X. The test automaton has a certain state q_{bad} that is only reachable when the system violates the property specified by the CD $\mathcal{C} \in CD_1$. That means, the question whether $\mathcal{C}$ is satisfied boils down to the question whether q_{bad} is not reachable. The construction of $\mathcal{T}(\mathcal{C})$ is designed in a way that the

[5] In the literature are several variants of both Timed Automata and parallel composition of them. In this paper we use basically the model of [AD94] and their notion of parallel composition. That means that a σ-labelled transition can only be executed iff each automaton in the system that uses label σ executes a σ-labelled transition.

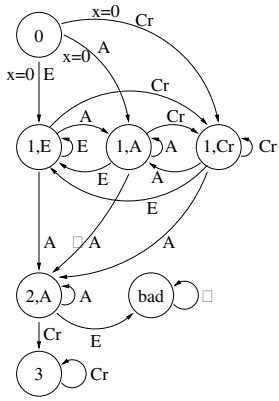
test automaton never blocks a synchronisation since it can always switch into a special state q_{good} to stop looking for a violation of the property. In the case of CD_1 a violating computation has to satisfy the assumptions $A_1, \ldots, A_n$ in that order. Then it has to change the value of X such that A_n is not satisfied anymore. Then the value should satisfy C_1. If not, $\mathcal{C}$ is already violated. Otherwise, it can only be violated by a change of X to a value that satisfies neither C_1 nor C_2. If the value changes to C_2 the system has to satisfy C_2 further on or C_3 and so on.

Let $\mathcal{C} \in CD_1$ be a CD that speaks about the variable X with type Σ. A test automaton $\mathcal{T}(\mathcal{C})$ for $\mathcal{C}$ is formally defined as follows: (We assume that $\varepsilon \notin \Sigma$ is a fresh synchronisation label.)

$$
\begin{aligned}
\mathcal{T}(\mathcal{C}) &= (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{IV}, S_0) \text{ with} \\
\mathcal{S} &= \{q_0, q_{n+1}, \ldots, q_{n+m}, q_{bad}, q_{good}\} \cup \{q_{i,\sigma} | 1 \leq i \leq n, \sigma \models A_i\} \\
\mathcal{X} &= \{x\} \\
\mathcal{L} &= \Sigma \cup \{\varepsilon\} \\
\mathcal{IV}(q) &= \mathsf{true} \text{ for all } q \in \mathcal{S} \\
S_0 &= \{q_0\} \\
\mathcal{E} &= \{(q_0, \sigma, x = 0, \emptyset, q_{1,\sigma}) \mid \sigma \models A_1\} && (1) \\
&\cup \{(q, \sigma, \mathsf{true}, \emptyset, q_{good}) \mid q \in \mathcal{S}, q \neq q_{bad}, \sigma \in \Sigma\} && (2) \\
&\cup \{(q_{i,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{i,\sigma'}) \mid 1 \leq i \leq n, \sigma, \sigma' \models A_i\} && (3) \\
&\cup \{(q_{i,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{i+1,\sigma'}) \mid 1 \leq i < n, \sigma \models A_i, \sigma' \models A_{i+1}\} && (4) \\
&\cup \{(q_{i,\sigma}, \varepsilon, \mathsf{true}, \emptyset, q_{i+1,\sigma}) \mid 1 \leq i < n, \sigma \models A_i \wedge A_{i+1}\} && (5) \\
&\cup \{(q_{n,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{n+1}) \mid \sigma \models A_n, \sigma' \models C_1\} && (6) \\
&\cup \{(q_{n,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{bad}) \mid \sigma \models A_n, \sigma' \not\models A_n \vee C_1\} && (7) \\
&\cup \{(q_{n+i}, \sigma, \mathsf{true}, \emptyset, q_{n+i}) \mid 1 \leq i \leq m, \sigma \models C_i\} && (8) \\
&\cup \{(q_{n+i}, \sigma, \mathsf{true}, \emptyset, q_{n+i+1}) \mid 1 \leq i < m, \sigma \models C_{i+1}\} && (9) \\
&\cup \{(q_{n+i}, \sigma, \mathsf{true}, \emptyset, q_{bad}) \mid 1 \leq i < m, \sigma \not\models C_i \vee C_{i+1}\} && (10) \\
&\cup \{(q_{bad}, \sigma, \mathsf{true}, \emptyset, q_{bad}) \mid \sigma \in \Sigma\} && (11)
\end{aligned}
$$

The state space includes an initial state q_0, states $q_{i,\sigma}$ for each assumption A_i and each $\sigma \models A_i$, states q_{n+j} for each commitment phase C_j, and special states q_{bad} and q_{good}. The idea of q_{bad} is that this state is reachable iff the given CD is not satisfied. State q_{good} is always reachable from all states with all synchronisations. That ensures that the test automata cannot block any behaviour of the system under consideration. Hence, all traces that are admissible *without* the test automaton remain admissible.

We need only one clock x and the set of synchronisation labels is given by the type Σ of the variable X together with the fresh label ε. Moreover, the test automaton does not need invariants. The transitions are defined as follows: Initially we expect that A_1 holds when the system starts its computation (1). In (2) we add transitions from all states to q_{good} to avoid blocking. In (3) we handle stuttering steps of assumption phases. In cases where a synchronisation happens



The correctness of this test automaton is quite obvious. All paths to reach state q_{bad} have to ensure that after an A-phase an E-phase was observed. Moreover, if a computation produces such a wrong state change, then the test automaton is able to observe this: First, it may stay in $\{q_{1,E}, q_{1,A}, q_{1,Cr}\}$. Then it guesses correctly when the A-phase starts that is succeeded by an E-phase. In this case it changes to $q_{2,A}$. When the E-phase starts it changes into q_{bad}.

Fig. 4. Test automaton for Fig. 2 c).

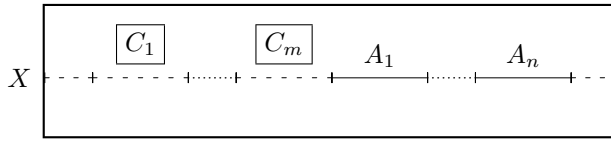
which allows us to proceed in the assumptions we can apply transitions in (4). If the last synchronisation belongs to both the current and the following assumption, then (5) allows us to proceed spontaneously in the assumptions. When all assumptions are given we apply (6) if a synchronisation is given that satisfies the first commitment. If a synchronisation is given that neither belongs to the last assumption nor to the first commitment, then we have seen a counterexample for the CD and enter q_{bad} (7). In (8) we allow stuttering steps in commitments whereas (9) allows us to proceed in the commitments provided that an appropriate synchronisation happens. If during a commitment a synchronisation occurs that neither belongs to the current nor to the following commitment, then we have seen a counterexample (10). Finally, (11) introduces idle transitions for q_{bad}.

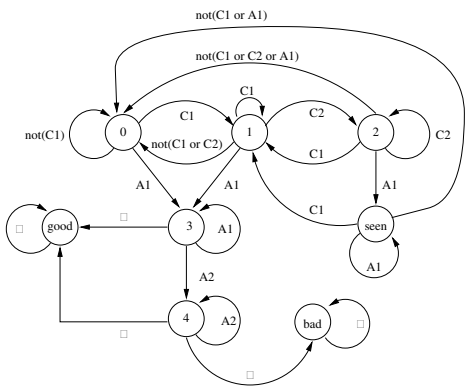
The automaton $\mathcal{T}(\mathcal{C})$ is basically a nondeterministic finite automaton that accepts a regular language, because CDs in CD_1 have no time requirements. Hence, an equivalent minimal deterministic automaton could be constructed. We refrain from that here, since introducing time assumptions and time commitments is easier with the given structure.

The CD in Fig 2 c) is in CD_1 with $A_1 = \mathsf{true}$, $A_2 = A$, and $C_1 = Cr$. The type of the variable $X = track$ is $\{E, A, Cr\}$. The corresponding test automaton is given in Fig. 4. Note that for sake of readability we omitted state q_{good} and all transitions to this state.

4.3 Past Commitments

The following kind of CDs requires the system to ensure a sequence of C_i phases before the occurrence of a sequence of A_j phases.





All paths reaching state q_{bad} ensure that an A_1-A_2-sequence happens without a preceeding C_1-C_2-sequence. Therefore all computations reaching q_{bad} violate the CD. Moreover, if a computation violates the CD the test automaton may guess correctly how long it has to stay in the states $\{q_0, q_1, q_2\}$. Note that the only nondeterministic choices are in q_0 and q_1 for A_1-signals. If the assumptions of the CD have to be met it changes into state q_3 which is possible because the preceeding phases are not a C_1-C_2-sequence. Hence, all violating computation can reach q_{bad}.

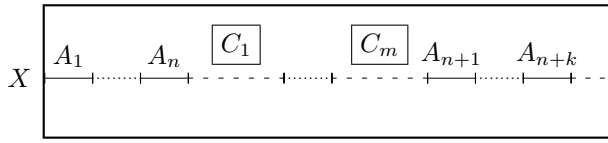
Fig. 5. Test automaton for a CD $\mathcal{C} \in CD_2$.

where $n, m \geq 1$, $C_i \neq \mathsf{true}$ for all $1 \leq i \leq m$, and A_1, $C_1, \ldots, C_m$ are *pairwise* disjoint. We call the set of all CDs of this kind CD_2.

The additional requirements for the commitments C_j allow us to construct a test automaton for these CDs as follows: The test automaton can deterministically check whether all commitments have occurred before the first assumption A_1 is visible. After the recognition of A_1 the automaton searches nondeterministically for the successive assumptions. If it is successful after an incorrect past a counterexample for the property was found. A formal description of the semantics of diagrams in CD_2 is given in App. B. The automaton $\mathcal{T}(\mathcal{C})$ with $m = n = 2$ is shown in Fig. 5 where we assume for simplicity that A_1 and A_2 are singletons.

4.4 Mixed Assumptions and Commitments

In [Let00] it is shown that it is possible to assign a test automaton for the following patterns of CDs:



with the following assumptions:

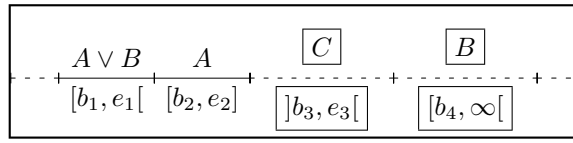
- $n, m, k > 0$,
- replacing the A_1,...,A_n phases by a dashed line would yield a CD in CD_2,
- replacing the A_{n+1},...,A_{n+k} phases by a dashed line would yield a CD in CD_1, and
- $A_n \Longrightarrow \neg A_{n+1}$.

4.5 Time Requirements

The time requirements which are allowed in our approach are

- time assumptions for assumption phases A_i and
- time commitment for commitment phases C_j.

Due to the construction of $\mathcal{T}(\mathcal{C})$ for a CD $\mathcal{C}$ in a form of the previous sections it is simple to add those time requirements. Instead of a formal treatment we will discuss the following example:



The CD without time requirements belongs to CD_1 with $A_1 = \mathsf{true}$, $A_2 = A \vee B$, $A_3 = A$, $C_1 = C$, and $C_2 = B$. The test automaton semantics of this example is given in Fig. 6. This test automaton is a convincing example that it is non-trivial to construct correct test automata by hand. It is far too easy to introduce a misleading edge or typo such that a verification would lead to wrong conclusions. Thus, tool support is needed.

Figure 6 omits state q_{good} again; the horizontal lines indicate the following: The first horizontal line annotated with $x := 0$ indicates that all crossing edges carry this annotation. Similarly the second horizontal line with $b_1 \leq x \leq e_1$ and $x := 0$ as annotations. Due to the construction of this test automaton a computation that leads into state q_{bad} has to ensure a phase where $A \vee B$ holds (states $q_{2,A}$, $q_{2,B}$) for a duration in $[b_1, e_1[$. After that it can reach state $q_{3,A}$ where it has to remain for a duration in $[b_2, e_2]$. If the system signals a change of the variable X to B before b_2 time units have elapsed, then the assumptions of the CD are not fulfilled. In this case the omitted transition to q_{good} is applicable. If B happens after b_2 and before e_2 time units, the assumptions are fulfilled, but the system does not satisfy the property since C was required. If the system fails to change X before e_2 time units are elapsed, the test automaton may also change to q_{bad} with an ε-transition. If the expected transition to C happens in time, the test automaton may change to q_4 to check the remaining commitments.

When the test automaton changes to q_4 it knows that all assumptions of the CD have occurred. It has to find all commitments now. The required change to C has also occurred. The CD requires that C holds for a duration in $]b_3, e_3[$ and then the system has to change to B and remain there for at least b_4 time units. Hence, in state q_4 a change to A is wrong, a change to B too early is wrong, and keeping the state for e_3 time units is wrong. If B arrives in time the system has met the commitment C_1 and now has to check whether C_2 is satisfied, too. The only way to violate this commitment is the arrival of A or C too early.

4.6 More Variables

In Constraint Diagrams it is allowed to constrain the behaviour of several variables (cf. Fig. 1). The approach described in the previous sections can be generalised for several variables as long as *commitments* are given for only one

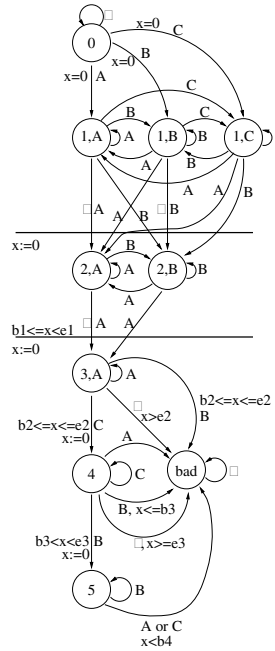


Fig. 6. Test automaton with time requirements.

variable.[6] We construct basically the test automaton for the Constraint Diagram that we get by omitting all variables without commitments. This is described above. The remaining assumptions about all other variables are checked by auxiliary automata which are synchronised appropriately with the main test automaton. Each of those automata has to reach a certain state which is only possible if all corresponding assumptions for the variable were seen.

5 Implementation

A prototypic implementation of the presented test automaton semantics was developed for the model-checker Uppaal [LPW97]. The tool Moby/CD allows the graphical development of Constraint Diagrams. The constructed CDs are translated into a textual representation which serves as the input language for the test automaton compiler. Applied to a textual description `cd` of a CD $\mathcal{C}$ the compiler generates three output files `cd.ta`, `cd.v` and `cd.q`. The file `cd.ta` contains the generated test automata and an additional timed automaton which

[6] If commitments for two or more variables are independent, this is no restriction, because the CD can be split into an equivalent conjunction of CDs which meet the restriction. Otherwise, there is no test automaton semantics possible in the general case (cf. the CD given in App. C).

we denote *demultiplexer*. This automaton serves as an interface to the test automata and interacts with the considered system via a simple protocol. The need for this additional automaton is given by the restricted synchronisation mechanisms in Uppaal. In contrast to the definition of [AD94] there are no direct ways to synchronise more than two automaton simultaneously. The workaround for this problem is as follows: Whenever there are changes of system variables referenced in $\mathcal{C}$ the system signals this via a certain synchronisation `CHANGE_SYNC` to the demultiplexer. Then the demultiplexer controls the values of all relevant system variables and sends to each test automaton synchronisations which indicate the actual values of the variables. Using such an interface decouples the test automata from the target system and allows an easy application because introducing the synchronisation `CHANGE_SYNC` can be done easily for many systems.

The file `cd.v` contains declarations of variables and clocks for the generated test automata. The files `cd.ta`, `cd.v` and the description of the considered system in terms of timed automata must be merged beforehand in order to build the complete system for the model checker. The other necessary input is a suitable reachability question (is q_{bad} reachable) which is contained in `cd.q`. If the Uppaal model checker negates this question the considered system fulfils the property described in $\mathcal{C}$. Otherwise Uppaal generates a trace which is a counterexample to the property in $\mathcal{C}$.

The sizes of the test automata produced from a CD are as follows: For each line of the CD we introduce a clock and an automaton (slightly optimised) with at most $2n + 3$ states where n is the number of phases in the line. For each arrow we need an additional clock. The demultiplexer consists of two states plus a state for each line of the CD.

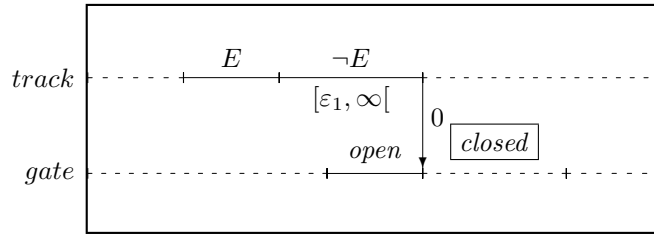
6 GRC Revisited

Consider the CDs in Fig. 2 again. It is obvious that b) and c) belong to the set of CDs for which a test automaton semantics was assigned. Constraint Diagram a) restricts the initial phase of the system, but it does belong neither to CD_1 nor to CD_2 nor to any of the extensions. In [Let00] some particular patterns[7] of CDs were considered and a test automaton semantics for these patterns was defined. Fortunately, Fig. 2 a) belongs to this set of patterns such that we have defined a test automaton semantics for all the CDs given in Sect. 3. Hence, given a Timed Automaton model of the track we would be able to verify with our approach whether this model satisfies the assumptions about the track.

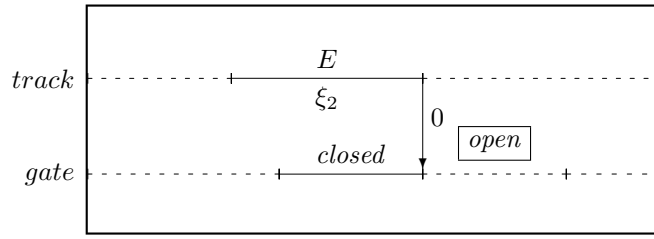
When we consider an implementation of a controller for the GRC we are interested whether the controller satisfies the properties **Safety** and **Utility** given in Sect. 2. Due to the assumption that trains need at least ε_1 time units to approach the crossing (Fig. 2 b)) it is safe when the controller satisfies the

[7] These patterns stem from a sublanguage of Duration Calculus [ZHR91,HZ97] called "Implementables" [Rav95]. This sublanguage consists of frequently used specification patterns for real-time systems.

following property. It requires the gate to become closed whenever the track is not empty for ε_1 time units:



This CD belongs to CD_1 with time extension and the extension for several variables. Hence, a test automaton can be constructed for **Safety**. Similarly, a CD can be found to specify a property of both gate and track to ensure **Utility**:



It requires the gate to open when the track is empty for more than ξ_2 time units. In [DD97] it is shown why these properties in conjunction with the assumptions about the track behaviour meet **Safety** and **Utility**.

7 Related Work Revisited

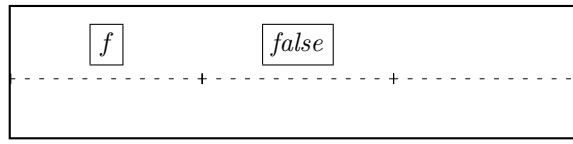
In this we discuss the similar approaches [ABL98,LPW98] in more detail.

[**LPW98**]: In this paper two kinds of formulas are introduced for which test automata are produced. The authors comment the expressiveness as follows:

> "We also noticed that though the logic is so simple, it characterizes the class of logical properties verified in all previous case studies where Uppaal is applied [...]."

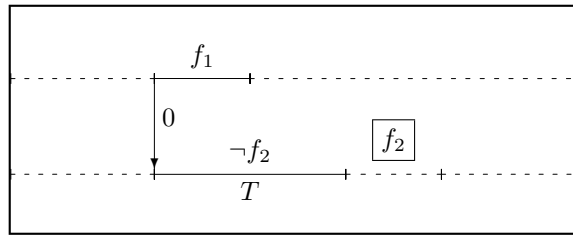
The formulas are either invariants (syntax: $Inv(f)$) where f is a boolean formula over atomic propositions or bounded response formulas (syntax: $f_1 \rightsquigarrow_{\leq T} f_2$) where T is a natural number.[8] Both kinds are expressible by CDs. An invariant $Inv(f)$ is equivalent to

[8] Note that [LPW98] allowed boolean formulas over atomic propositions *and* clock constraints. The clock constraints can be replaced by auxiliary atomic propositions.



This CD belongs to the set of patterns mentioned in the previous section. Thus, it has a test automaton semantics. The meaning is that the system has to fulfill f initially and *false* afterwards. Hence, f must hold forever.

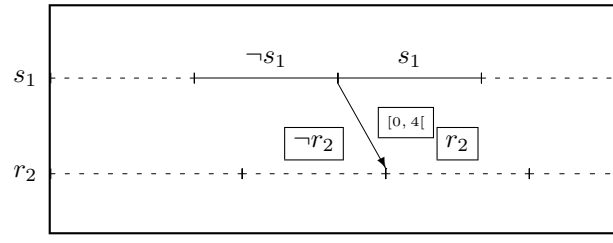
The bounded response formula $f_1 \rightsquigarrow_{\leq T} f_2$ requires the system to ensure the property f_2 within at most T time units when the property f_1 becomes true. This is stronger than the classical until-operator where f_1 has to hold until f_2 becomes true. The bounded response can be expressed by the following CD that belongs to CD_1 (with extensions):

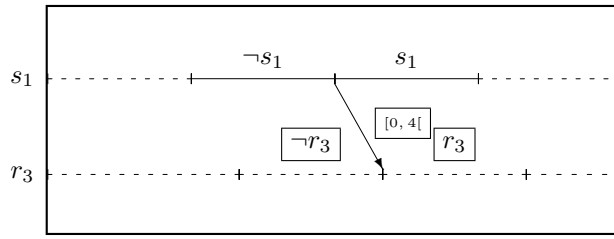


[ABL98]: The approach in this paper is more general and difficult to compare to our approach. The reason is that in [ABL98] a logic called SBLL (safety and bounded liveness) is introduced for which a translation into a test automaton is presented. Instead of discussing the details of SBLL we discuss a SBLL formula in [ABL98]:

$$inv([\mathsf{send}_1!]\ s\ \underline{\text{in}}\ \Cap([\mathsf{recv}_2!](s < 4) \wedge [\mathsf{recv}_3!](s < 4))) \tag{12}$$

This describes a requirement for the CSMA/CD protocol. The informal meaning is that whenever node 1 sends a message then nodes 2 and 3 receive the message within less than 4 time units. Note that SBLL formulas speak about synchronisation labels whereas CDs speak about the states and variables. Assuming appropriate variables s_1, r_2, r_3 we can represent this property by





Note that CDs allow to represent this property by a single CD but the given representation is equivalent and uses commitments only for a single observable. Hence, there is a test automaton semantics available (cf. Sect. 4.6). It depends on the individual skills, experiences and education which representation is more accessible: The formula (12) or the CDs above.

8 Conclusion

Constraint Diagrams have been designed as a graphical language for the requirements capture of real-time system. From the CD's point of view the main contribution of this paper is the new applicability of this specification language as temporal logic for automatic formal verification. From the Timed Automata point of view the main contribution of this paper is the new availability of a graphical temporal logic in assumption/commitment-style for which test automata can be constructed automatically.

Acknowledgements

The authors thank E.-R. Olderog and the members of the "semantics group" in Oldenburg for fruitful discussions on the subject of this paper.

References

ABL98. L. Aceto, A. Burgueño, and K. Larsen. Model Checking via Reachability Testing for Timed Automata. In Steffen [Ste98], pages 263–280.

ACD90. R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time Systems. In *Fifth Annual IEEE Symp. on Logic in Computer Science*, pages 414–425. IEEE Press, 1990.

AD90. R. Alur and D.L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

AD94. R. Alur and D.L. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

DD97. H. Dierks and C. Dietz. Graphical Specification and Reasoning: Case Study "Generalized Railroad Crossing". In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME'97*, volume 1313 of *LNCS*, pages 20–39, Graz, Austria, September 1997. Springer.

DFMV98a. H. Dierks, A. Fehnker, A. Mader, and F.W. Vaandrager. Operational and Logical Semantics for Polling Real-Time Systems. In A.P. Ravn and H. Rischel, editors, *FTRTFT'98*, volume 1486 of *LNCS*, pages 29–40, Lyngby, Denmark, September 1998. Springer. short version of [DFMV98b].

DFMV98b. H. Dierks, A. Fehnker, A. Mader, and F.W. Vaandrager. Operational and Logical Semantics for Polling Real-Time Systems. Technical Report CSI-R9813, Computer Science Institute Nijmegen, Faculty of Mathematics and Informatics, Catholic University of Nijmegen, April 1998. full paper of [DFMV98a].

Die96. C. Dietz. Graphical Formalization of Real-Time Requirements. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 366–385, Uppsala, Sweden, September 1996. Springer.

Die99. H. Dierks. *Specification and Verification of Polling Real-Time Systems.* PhD thesis, University of Oldenburg, July 1999.

FJ97. K. Feyerabend and B. Josko. A Visual Formalism for Real-Time Requirements Specifications. In M. Bertran and T. Rus, editors, *ARTS'97*, volume 1231 of *LNCS*, pages 156–168, Mallorca, Spain, May 1997. Springer.

HL94. C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing. In *IEEE Real-Time Systems Symposium*, 1994.

HLR93. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing. Springer, June 1993.

HM96. C. Heitmeyer and D. Mandrioli, editors. *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*. Wiley, 1996.

HNSY94. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111:193–244, 1994.

HZ97. M.R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283–330, 1997.

Kle00. C. Kleuker. *Constraint Diagrams.* PhD thesis, University of Oldenburg, December 2000.

Let00. M. Lettrari. Eine Testautomatensemantik für Constraint Diagrams und ihre Anwendung. Master's thesis, University of Oldenburg, Department of Computer Science, Oldenburg, Germany, April 2000.

LPW97. K.G. Larsen, P. Petterson, and Wang Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1+2):134–152, December 1997.

LPW98. M. Lindahl, P. Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In Steffen [Ste98], pages 281–297.

Mos85. B. Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10–19, 1985.

Rav95. A.P. Ravn. Design of Embedded Real-Time Computing Systems. Technical Report 1995-170, Technical University of Denmark, 1995.

Sch01. R. Schlör. *Symbolic Timing Diagrams: A Visual Formalism for Model Verification.* PhD thesis, University of Oldenburg, Department of Computer Science, Oldenburg, Germany, 2001.

Ste98. B. Steffen, editor. *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*. Springer, 1998.

Yov97. S. Yovine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.

Zho93. Zhou Chaochen. Duration Calculi: An overview. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Formal Methods in Programming and Their Application*, volume 735 of *LNCS*, pages 256–266. Springer, 1993.

ZHR91. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *IPL*, 40/5:269–276, 1991.

A Duration Calculus Semantics for CDs

In this appendix first we introduce briefly Duration Calculus , a formalism in which the semantics of CDs is given in [Kle00]. After this we explain briefly the ideas of the semantics of CDs in Duration Calculus. A full treatment of the semantics can be found in [Kle00].

Duration Calculus [ZHR91,Zho93,HZ97] (DC for short) is a real-time interval temporal logic extending earlier work on discrete interval temporal logic of [Mos85]. A formal description of a real-time system using DC starts by choosing a number of time-dependent state variables (called "observables") *obs* of a certain type. An interpretation I assigns to each observable a function $obs_I : \mathsf{Time} \longrightarrow D$ where Time is the time domain, here the non-negative reals, and D is the type of *obs*. If D is finite, these functions obs_I are required to be *finitely variable*, which means that any interval $[b, e] \subseteq \mathsf{Time}$ can be divided into finitely many subintervals such that obs_I is constant on the open subintervals.

State assertions P are obtained by applying propositional connectives to elementary assertions of the form $obs = v$ (v for short if *obs* is clear) for a $v \in D$. For a given interpretation I state assertions denote functions $P_I : \mathsf{Time} \longrightarrow \{0, 1\}$.

Duration terms are of type real and their values depend on a given time interval $[b, e]$. The simplest duration term is the symbol ℓ denoting the length $e - b$ of $[b, e]$. For each state assertion P there is a duration term $\int P$ measuring the duration of P, i.e. the accumulated time P holds in the given interval. Semantically, $\int P$ denotes $\int_b^e P_I(t)dt$ on the interval $[b, e]$.

Duration formulas are built from arithmetical relations applied to duration terms, the special symbols true and false, and other terms of type real, and they are closed under propositional connectives and quantification over rigid variables. Their truth values depend on a given interval. We use F for a typical duration formula. true and false evaluate to true resp. false on every given interval. Further basic duration formulas are:

Relation over Durations: For example, $\int P = k$ expresses that the *duration* of the state assertion P in $[b, e]$ is k.

Chop: The composite duration formula $F_1; F_2$ (read as F_1 *chop* F_2) holds in $[b, e]$ if this interval can be divided into an initial subinterval $[b, m]$ where F_1 holds and a final subinterval $[m, e]$ where F_2 holds.

Besides this basic syntax various abbreviations are used:

$$\begin{aligned}
\text{point interval: } \lceil\rceil &\stackrel{\text{df}}{=} \ell = 0 \\
\text{everywhere: } \lceil P \rceil &\stackrel{\text{df}}{=} \textstyle\int P = \ell \ \wedge \ \ell > 0 \\
\text{somewhere: } \Diamond F &\stackrel{\text{df}}{=} \mathsf{true}; F; \mathsf{true} \\
\text{always: } \Box F &\stackrel{\text{df}}{=} \neg\Diamond\neg F
\end{aligned}$$

A duration formula F *holds* in an interpretation I if F evaluates to true in I and every interval of the form $[0, t]$ with $t \in \mathsf{Time}$. If convenient, we use F to describe a set of interpretations namely all interpretations in which F holds.

Semantically, Constraint Diagrams denote an implication between assumptions and commitments of the form

$$\forall \epsilon_1, \ldots, \epsilon_k.\ (\ Assm(\epsilon_1, \ldots, \epsilon_k) \implies \exists \delta_1, \ldots, \delta_l.\ Comm(\epsilon_1, \ldots, \epsilon_k, \delta_1, \ldots, \delta_l))$$

for real variables ϵ_i, δ_j with $i \leq k, j \leq l, k, l \in \mathbb{N}$. Assumptions as well as commitments characterising lines are conjunctions of sequence formulae like

$$(\lceil P_1 \rceil \wedge \ell = \epsilon_1)\,;\ \ldots\ ;\ (\lceil P_n \rceil \wedge \ell = \epsilon_n)$$

for state assertions P_i and real variables $\epsilon_i, i \leq n, n \in \mathbb{N}$. Difference formulae characterising arrows have the form

$$\left(\sum_{i=1}^{n} \epsilon_i - \sum_{j=1}^{m} \delta_j \right) \in Intv$$

for real variables ϵ_i, δ_j, $i \leq n, j \leq m, n, m \in \mathbb{N}$ and an interval $Intv$. They are also needed in length requirements between lengths of phases in assumptions and commitments to assure that they concern the same subintervals.

Consider the CD for the watchdog (Fig. 1). The DC semantics is equivalent to this formula:

$$\begin{aligned}
\forall \epsilon_1, \epsilon_2 : & \ell = \epsilon_1; (\lceil \neg S \rceil \wedge \ell = 10); \ell = \epsilon_2 \\
\Longrightarrow & \exists \delta_1, \delta_2, \delta_3 : \ell \geq \epsilon_1 + \epsilon_2 + 1 \\
\Longrightarrow & \ell = \delta_1; (\lceil A \rceil \wedge \ell = \delta_2); \ell = \delta_3 \wedge \\
& \delta_1 - (\epsilon_1 + \epsilon_2) \in [0, 1]
\end{aligned}$$

Detailed definitions and discussions of the semantics are found in [Kle00].

B Test Automaton Semantics for CD_2

Formally we define the test automaton $\mathcal{T}(\mathcal{C})$ for a CD $\mathcal{C} \in CD_2$ in this way:

$$
\begin{aligned}
\mathcal{T}(\mathcal{C}) &= (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{IV}, S_0) \text{ with} \\
\mathcal{S} &= \{q_0, q_{bad}, q_{good}, q_{seen}, q_1, \ldots, q_m\} \cup \{q_{m+i,\sigma} | 1 \leq i \leq n, \sigma \models A_i\} \\
\mathcal{X} &= \{x\} \\
\mathcal{L} &= \Sigma \cup \{\varepsilon\} \\
\mathcal{IV}(q) &= \mathsf{true} \text{ for all } q \in \mathcal{S} \\
S_0 &= \{q_0\}
\end{aligned}
$$

The transitions $\mathcal{E}$ are defined in Figure 7.

$$
\begin{aligned}
\mathcal{E} = {} & \{(q, \sigma, \mathsf{true}, \emptyset, q_{good}) \mid q \in \mathcal{S}, q \neq q_{bad}, \sigma \in \Sigma\} && (13)\\
& \cup \{(q_i, \sigma, \mathsf{true}, \emptyset, q_{i+1}) \mid 0 \leq i < m, \sigma \models C_{i+1}\} && (14)\\
& \cup \{(q_0, \sigma, \mathsf{true}, \emptyset, q_0) \mid \sigma \not\models C_1\} && (15)\\
& \cup \{(q_i, \sigma, \mathsf{true}, \emptyset, q_i) \mid 1 \leq i \leq m, \sigma \models C_i\} && (16)\\
& \cup \{(q_i, \sigma, \mathsf{true}, \emptyset, q_1) \mid 1 \leq i \leq m, \sigma \models C_1\} && (17)\\
& \cup \{(q_i, \sigma, \mathsf{true}, \emptyset, q_0) \mid 1 \leq i < m, \sigma \not\models C_1 \cup C_i \cup C_{i+1}\} && (18)\\
& \cup \{(q_i, \sigma, \mathsf{true}, \emptyset, q_{m+1,\sigma}) \mid 0 \leq i < m, \sigma \models A_1\} && (19)\\
& \cup \{(q_m, \sigma, \mathsf{true}, \emptyset, q_0) \mid \sigma \not\models C_1 \cup C_m \cup A_1\} && (20)\\
& \cup \{(q_m, \sigma, \mathsf{true}, \emptyset, q_{seen}) \mid \sigma \models A_1\} && (21)\\
& \cup \{(q_{seen}, \sigma, \mathsf{true}, \emptyset, q_{seen}) \mid \sigma \models A_1\} && (22)\\
& \cup \{(q_{seen}, \sigma, \mathsf{true}, \emptyset, q_1) \mid \sigma \models C_1\} && (23)\\
& \cup \{(q_{seen}, \sigma, \mathsf{true}, \emptyset, q_0) \mid \sigma \not\models A_1 \vee C_1\} && (24)\\
& \cup \{(q_{m+i,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{m+i,\sigma'}) \mid 1 \leq i \leq n, \sigma, \sigma' \models A_i\} && (25)\\
& \cup \{(q_{m+i,\sigma}, \sigma', \mathsf{true}, \emptyset, q_{m+i+1,\sigma'}) \mid 1 \leq i < n, \sigma \models A_i, \sigma' \models A_{i+1}\} && (26)\\
& \cup \{(q_{m+i,\sigma}, \varepsilon, \mathsf{true}, \emptyset, q_{m+i+1,\sigma}) \mid 1 \leq i < n, \sigma \models A_i \wedge A_{i+1}\} && (27)\\
& \cup \{(q_{m+n,\sigma}, \varepsilon, \mathsf{true}, \emptyset, q_{bad}) \mid \sigma \models A_n\} && (28)\\
& \cup \{(q_{bad}, \sigma, \mathsf{true}, \emptyset, q_{bad}) \mid \sigma \in \Sigma\} && (29)
\end{aligned}
$$

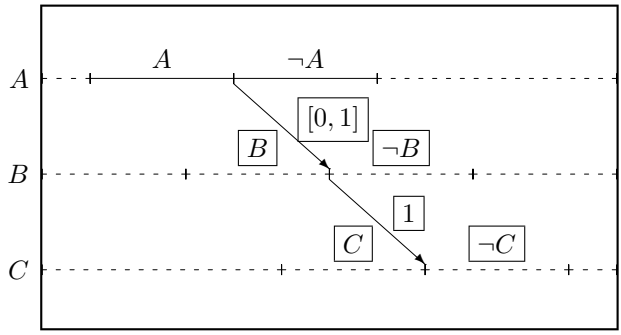
Fig. 7. Definition of $\mathcal{E}$.

The idea of the state space is that $q_0, \ldots, q_m$ represent the number of fulfilled commitments at the moment. Hence, if we are not in state q_m and the assumption can be match, then a counterexample was found. If we are in state q_m and the first assumption is fulfilled, we switch into the special state q_{seen} in order to avoid the detection of a wrong counterexample. All other states have a similar meaning as in the semantics for CD_1.

(13): Transitions to q_{good} are always possible to avoid blocking. (14): In state q_i the next commitment was observed, thus the successor state is q_{i+1}. (15): Idling is possible for q_0 as long the first commitment was not seen. (16)–(19): Transitions for q_i $(1 \leq i < m)$. If C_i holds, it idles (16); if C_1 holds, the first commitment was seen and q_1 is the successor state (17); if C_{i+1} holds, (14) is applicable to proceed in the sequence of the commitments; in all other cases a change to q_0 is allowed (18); if the first assumption is observed it may change to the assumption states (19). Transitions for q_m are similar except for the case of A_1 where it changes into q_{seen} in order to avoid wrong counterexamples (21). Transitions for q_{seen} hold the state as long as A_1 is true (22) and changes otherwise to q_0 or q_1 respectively (23),(24). (25)–(27): Similar to (3)–(5). (28): If all assumptions were seen a counterexample was found. (28): Idling of sink states.

C A CD without TA Semantics

For the following CD it is not possible to find a test automaton semantics, because it would need infinitely many clocks:



The meaning of this CD is that whenever we observe a change from A to $\neg A$ at time t_A the system has to produce a change from B to $\neg B$ at time $t_B \in [t_A, t_A + 1]$ and a change from C to $\neg C$ at time $t_B + 1$. In order to detect a counterexample the test automaton has to verify that all possible instances of t_B do not satisfy the commitment. However, the decision whether an instance of t_B satisfies the commitments depends on the future. Therefore the test automaton has to remember all possible candidates for t_B, i.e. all time points when the system changes from B to $\neg B$. However, this is not possible with a static number of clocks because it is possible that more changes happen as clocks are available.