Durative Graph Transformation Rules

Technical Report
tr-ri-13-329

Steffen Ziegert
Specification and Modelling of Software Systems
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
steffen.ziegert@uni-paderborn.de

Christian Heinzemann
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Zukunftsmeile 1
D-33102 Paderborn, Germany
c.heinzemann@uni-paderborn.de

Version: 1.0

Paderborn, March 5, 2013

1This work was developed in the course of the Collaborative Research Centre 614 “Self-optimizing Concepts and Structures in Mechanical Engineering” and funded by the German Research Foundation (DFG).
## Contents

1 Introduction 1

2 Fundamentals 2
   2.1 Graph Transformations 2
   2.2 Timed Graphs and Clock Instances 3

3 Approach 4

4 Semantics 6
   4.1 Timed GT Rules 7
   4.2 Clock Instance and Invariant Rules 11
   4.3 Operational Semantics 13

5 Properties of the Semantics 14

6 Related Work 18

7 Conclusion 19

References 19
1 Introduction

Advanced mechatronic systems, like smart cars or smart trains, operate autonomously in unknown and frequently changing environments. During operations, they need to cooperate with other systems in their environment, e.g., for organizing the passage of a crossing. In order to perform these tasks efficiently, i.e., with minimal (hardware) resources, these systems apply reconfiguration for adapting their behaviour to their changing environment \cite{CdLG09}. Technically, reconfiguration is achieved by changing the software architecture of the system at runtime.

The reconfiguration behaviour of a mechatronic system, however, is safety-critical, because a wrong reconfiguration may lead to an erroneous behaviour in a certain situation. In case of a smart car trying to pass a crossing such erroneous behaviour may lead to a crash. As a consequence, we need to apply formal verification for guaranteeing that the reconfiguration behaviour is correct and may not cause such accidents.

In previous publications \cite{BBG06, EHH13}, we have shown that graph transformations \cite{EEPT06} are a suitable formalism for specifying and verifying reconfiguration behaviour. Our approaches and other related approaches like Real-Time Maude \cite{OM07} or GROOVE \cite{KR06}, however, assume that reconfiguration operations are executed atomically in zero time. In reality, reconfiguration operations obviously need time for being executed and, hence, reconfiguration operations of different systems may be executed concurrently. The concurrent execution, however, may cause reconfiguration operations to interfere with each other. The approach by Rivera et al. \cite{RDV09} considers such situations and proposes to cancel one of the involved operations. This, however, has the result that a reconfiguration that has been started will not be finished correctly. This may lead, again, to an erroneous behaviour and, in the worst case, to an accident. As a result, we need to take the duration of a reconfiguration into account upon verification and ensure that each reconfiguration operation can be finished if it has been started.

In this paper, we extend our graph-based specification approach of \cite{EHH13} by so-called durative rules. A durative rule extends a normal graph transformation rule by a duration that it needs for being executed. We define how durative rules can be mapped to our existing timed graph verification framework \cite{EHH13}. In particular, we present a locking approach that guarantees that after starting the execution of a durative rule, it can always be finished correctly. This, in turn, guarantees that reconfiguration operations can only be executed concurrently if they do not interfere with each other. We show this property as part of our contribution.

We illustrate our approach using a smart train system which is developed at the University of Paderborn. In this system, small trains, called RailCabs, drive autonomously on the track system. A particular feature of the RailCab system\footnote{http://www.railcab.de} is the convoy mode where RailCabs drive at very small distances to reduce their energy consumption.
The paper is structured as follows. In Section 2, we introduce graph transformation systems as well as timed graphs that are used for representing a state of the system in our timed graph verification framework. In Section 3, we outline our approach of durative rules for graph transformation systems in more detail before introducing the semantics of durative rules and the mapping to our timed graph verification framework in Section 4. We show that our mapping ensures the intended properties for concurrent execution in Section 5. Section 6 discusses related approaches before we conclude the paper in Section 7.

2 Fundamentals

2.1 Graph Transformations

A graph transformation system (GTS) consists of a set of graph transformation rules (GT rules) and an initial graph. The GT rules can be applied to the initial graph and its resulting successor graphs to construct the state space of the GTS. The underlying theory of GTS is based on graphs and graph morphisms.

Definition 2.1 (Graph, Graph Morphism)
A graph $G = (V_G, E_G, src_G, tgt_G)$ consists of a set of nodes $V_G$, a set of edges $E_G$, and source and target functions $src_G, tgt_G : E_G \rightarrow V_G$. A graph morphism $f : G \rightarrow H$ between two graphs is a pair of mappings $f = (f_E, f_V)$ with $f_E : E_G \rightarrow E_H$ and $f_V : V_G \rightarrow V_H$ such that $f_V \circ src_G = src_H \circ f_E$ and $f_V \circ tgt_G = tgt_H \circ f_E$. A graph morphism $f = (f_E, f_V)$ is injective if $f_E$ and $f_V$ are injective.

A graph morphism is a mapping of nodes and edges of one graph to nodes and edges of another graph such that the source and target nodes of edges are preserved. Such morphisms are used in GT rules to define which nodes and edges are created, deleted, or preserved when the rule is applied to a graph.

Definition 2.2 (Graph Transformation Rule)
A graph transformation rule $p = (L, R, r)$ consists of two graphs $L$ and $R$, called left-hand side (LHS) and right-hand side (RHS), and an injective partial graph morphism $r : L \rightarrow R$, called rule morphism. Given a graph transformation rule $p$ and a match $m : L \rightarrow G$ of its LHS into a host graph $G$, the direct derivation from $G$ with $p$ at $m$, written $G \xrightarrow{m \times p} H$, is the pushout of $r$ and $m$ in Graph$^P$, the category of graphs and partial graph morphisms, as shown below. [EHK+97]

$$
\begin{array}{c}
L \xrightarrow{r} R \\
\downarrow m \quad (PO) \quad \downarrow m' \\
G \xrightarrow{r'} H
\end{array}
$$

Whether a GT rule can be applied to a graph depends on whether a match of its LHS to the graph can be found. To further restrict the applicability of a rule, negative application
conditions (NAC) can be used that forbid specific graph structures from being present in the graph.

**Definition 2.3 (Negative Application Condition)**

Let \( p = (L, R, r) \) be a graph transformation rule, \( G \) a graph, and \( m : L \to G \) a match. A **negative application condition** is a tuple \((N, n)\) with \( n : L \to N \) and \( n \) being injective. If \( \neg \exists q : N \to G \) such that \( q \circ n = m \), then \( m \) satisfies \((N, n)\), written \( m \models (N, n) \).

### 2.2 Timed Graphs and Clock Instances

Timed GTS operate on timed graphs which are an extension of typed graphs [EEPT06]. In addition to the normal graph nodes, a timed graph contains a set of clock instances which measure the progress of time. As in timed automata [AD94, BY03], the values of all clock instances increase continuously and synchronously with the same rate. A clock instance always applies to a subgraph of the timed graph and has edges to all nodes of the subgraph but no other edges.

**Definition 2.4 (Timed Graph)**

Let \( TG \) be a type graph. A **timed graph** \( TiG = (H, \text{type}) \) consists of a graph \( H = (V_G, V_{CI}, E_G, E_{CI}, (src,j, tgt,j))_{j \in \{G, CI\}} \), where

- \( V_G \) and \( V_{CI} \) are called graph nodes and clock instances, respectively;
- \( E_G \) and \( E_{CI} \) are called graph edges and clock instance edges, respectively;
- \( src_G : E_G \to V_G, tgt_G : E_G \to V_G \) are the source and target function for graph edges;
- \( src_{CI} : E_{CI} \to V_{CI}, tgt_{CI} : E_{CI} \to V_G \) are the source and target function for clock instance edges;
- \( \text{type} : H \to TG \) is a graph morphism. \([SHS11, EEPT06]\)

To define the different kinds of rules of the timed GTS formalism, we need the definition of a morphism on timed graphs. A timed graph morphism preserves the source and target nodes of edges, the types of nodes, and the values assigned to attributes.

**Definition 2.5 (Timed Graph Morphism)**

A (timed graph) morphism \( f \) between two timed graphs \( TiG_i, i = 1, 2 \), is a partial graph morphism \( f = (f_{V_G}, f_{V_{CI}}, f_{E_G}, f_{E_{CI}}) \) with \( f_{V_G} : V_{j,1} \to V_{j,2}, j \in \{G, CI\} \), and \( f_{E_{k}} : E_{k,1} \to E_{k,2}, k \in \{G, CI\} \). \( f \) commutes for all source and target functions and preserves types, i.e. \( \text{type}_2 \circ f = \text{type}_1 \). \([SHS11]\)

In addition to timed graph morphisms, we need a definition of clock instance constraints which are used by different rules of the timed GTS formalism. They are used to restrict the values of a clock instance to a specific interval (either as application condition or as invariant).
3 Approach

To illustrate our approach of durative reconfigurations for GTS, we use a convoy scenario of the aforementioned RailCab system as running example. Figure 1 shows the simplified software architecture of the RailCab system in a class diagram. It consists of track segments that are connected to each other via next links. A RailCab can occupy such a track segment. Furthermore, RailCabs can form a convoy. Such a convoy operation is represented by a Convoy object and member links to each participating RailCab. In addition, there are first and last links that represent the head and tail of the convoy, respectively.

To model the reconfigurations of a system’s software architecture, we use story patterns which have a formal semantics based on GTS. Story patterns represent the LHS, RHS, and NACs of a rule in a single graph by using stereotypes. Elements going to be deleted (LHS only) are equipped with the stereotype «\(-\rightarrow\)»; elements to be created (RHS only) with «\(+\rightarrow\)»;

3 Story patterns follow the single pushout approach to graph transformation.

Figure 1: Class diagram of the RailCab system
Ordinary GTS do not include a notion of time. However, as system reconfigurations usually consume time, time should be reflected in the semantics: a path in the state space of a GTS should clearly indicate when a time-consuming reconfiguration starts and when it ends. Therefore, in our approach, a rule has an annotated value for its duration. For these durative rules we developed a semantics based on timed graphs and timed GTS. This enables us to use our existing verification framework [EHH+13] for the verification of durative rules.

In our semantics durative reconfigurations are supported by translating them into two discrete reconfigurations that are temporally linked to each other. One reconfiguration represents the start of the durative reconfiguration; a second one represents its end. Since reconfigurations have an application interval now, it is possible to apply them concurrently. A naive approach that allows concurrent rule applications to overlap arbitrarily might lead to conflicts between two reconfigurations.

Consider for example the story patterns in Figures 2 and 3 and a configuration where two RailCabs are driving in a convoy at a given time. If the breakConvoy reconfiguration is in the process of execution but the actual change of the state will not happen until its execution ends, then a joinConvoy reconfiguration could be scheduled in the meantime. However, as breakConvoy ends before joinConvoy ends, there will be no Convoy object anymore that the joining RailCab object can be linked to. If such a sequence of reconfigurations was to be executed, a rear-end collision might occur in the worst case.

To avert such conflicts we integrate a locking mechanism into the semantics of durative rules. The application of a start rule adds locks into the host graph. There are read locks for objects or links that are preserved and write locks for their creation and deletion. These locks are attached directly to the timed graph as locking edges when the start rule is applied. The application of the end rule removes these locking edges again. Conflicts cannot occur anymore because start rules require the non-existence of locking edges according to the changes they specify. Such a use of locking edges prevents inconsistent configurations from occurring and
ensures that reconfigurations are either carried out completely or not at all. While a concurrent read is allowed, a concurrent write or read-write is not allowed.

Figure 4 shows a configuration of the system where locking edges have been created by the application of breakConvoy’s start rule. Locks on objects are incorporated into the configuration as self-loops. Locks on links are realized as locking edges with same source and target as the link itself. To correctly correlate locking edges to the links they are supposed to restrict the access to, there is one pair of locking edge types (for read and write locks) for every link type. In the figure, locking edges are depicted using dashed arrows.

Figure 4: Configuration of the system where the breakConvoy rule is in ongoing execution

4 Semantics

Our notion of durative rules is inspired by the fact that reconfigurations in software systems require time. On the syntactic level, a durative rule is merely a GT rule with an annotated name and duration value. The idea is that the execution of a durative graph transformation cannot be aborted once it has been started. A modeller who uses durative rules for specification does not need to know about the complicated timed behaviour that happens under the hood.

Definition 4.1 (Durative Graph Transformation Rule)

A durative graph transformation rule $D = (DL, DR, dr, name, d)$ consists of

- two typed graphs, a left-hand side $DL$ and a right-hand side $DR$,
- a partial graph morphism $dr : DL \rightarrow DR$,
- a distinct name $name$, and
- a duration $d \in \mathbb{N}^{>0}$.
The semantics of durative GT rules are defined on top of the rules already established by the timed GTS formalism. A durative rule induces a pair of timed rules (i.e. its start and end rule), a so-called clock instance rule, and an invariant rule. Intuitively, the application of the start and end rule indicate the interval of the durative rule’s execution, the clock instance rule triggers the measuring of time, and the time invariant rule enforces the application of the end rule after $d$ time units have passed since the application of the start rule.

Before giving the definitions of the induced rules, we explain how we realize the locks in timed graphs. Locking of nodes and edges is done via the creation and deletion of additional edges, called locking edges. The types of locking edges are defined in the type graph of the timed GTS. Every node type $t$ has two locking edge types, $rlnode(t)$ and $wlnode(t)$, as self-loops. For every edge type $t$ (that is no locking edge type itself), there are locking edges types $rledge(t)$ and $wledge(t)$ adjacent to the same source and target node types. An object of type $rlnode(t)$ depicts an obtained read lock on the object; an object of type $wlnode(t)$ a write lock. Similarly, a link of type $rledge(t)$ depicts an obtained read lock on the link between its source and target that has the type $t$, $wledge(t)$ a write lock. In addition to the locking edges, there is a node type $name$ for every durative rule and an edge type $underApplication$ from $name$ to every other node type. Objects of type $name$ are called application indicator and indicate the ongoing execution of a durative reconfiguration if available in a configuration. Their outgoing links mark the matching of the durative rule that has been applied, i.e., the subgraph of the configuration that is changed by the reconfiguration.

### 4.1 Timed GT Rules

A timed GT rule basically works like a normal GT rule. It searches for a match of the left-hand side (LHS) in the host graph and transforms the respective subgraph such that it is isomorphic to the right-hand side (RHS). In addition, a timed GT rule specifies a timed guard and resets. The time guard is a clock instance constraint based on the clock instances contained in the LHS. It needs to be evaluated to true for the rule to be applicable. Upon application, all clock instances contained in the resets are set back to 0.

**Definition 4.2 (Timed Graph Transformation Rule)**

A timed graph transformation rule $tr = (L, R, r, \mathcal{N}, z, V_{res})$ consists of two timed graphs $L$ and $R$, a rule morphism $r : L \rightarrow R$ with $r(V_{CI,L}) = V_{CI,R}$ and $|V_{CI,L}| = |V_{CI,R}|$, a set of NACs $\mathcal{N}$, where each NAC is a tuple $(N, n) \in \mathcal{N}$ with $n : L \rightarrow N$, $z \in Z(V_{CI,L})$ is a clock instance constraint, the time guard and $V_{res} \subseteq V_{CI,R}$ is a set of clock instances called resets. \[SHS11, EHH^+13\]

Both, the induced start rule and the induced end rule are defined on top of this timed GT rule. Intuitively, the end rule realizes the reconfiguration syntactically defined by the durative rule. The start rule serves two purposes:

- It adds information about the execution of the durative rule into the host graph. This is needed for the annotation of time and for the end rule to find a match that corresponds to
the match of the start rule. Finding a corresponding match is important because together both rules are supposed to represent the application interval of a durative transformation. With different matches there would be no meaningful interpretation for the application of a durative rule.

- It adds locking edges into the host graph such that subsequent rules do not match if they access the same elements in a conflicting manner. These locking edges are removed again by the end rule.

Figure 5 shows an example of a durative rule called Simple and its induced start and end rule. The durative rule (a) specifies the removal of a link $x$ during an interval of $5$ time units. The induced start rule (b) has a LHS that corresponds to the LHS of the durative rule. Negative application conditions allow its application only if the preserved nodes are not write-locked and the edge to be removed is neither read- nor write-locked. When the rule is applied, an application indicator Simple is created. For each preserved node a read lock $rl$ is created; the edge to be deleted also obtains a write lock $wl(x)$. The LHS of the induced end rule (c) includes the application indicator node Simple and its adjacent edges. This ensures the correct matching of the LHS to the host graph. Locking edges are removed again when the rule is applied. The guard $ci \geq 5$ is an application condition of the end rule. The consumption of $5$ time units is guaranteed by the invariant rule which is formally explained in Section 4.2.

**Definition 4.3 (Induced Start Rule)**

Let $D = (DL, DR, dr, name, d)$ be a durative rule. The induced start rule of $D$ is a timed rule $sr = (L, R, r, N, z, V_{res})$ with

1. $V_{G,L} = V_{DL} \land E_{G,L} = E_{DL}$,
2. $V_{G,R} = V_{DL} \cup \{ai\} \land type(ai) = name \land E_{G,R} = E_{DL} \cup \{e|src(e) = ai \land tgt(e) \in V_{G,R} \land type(e) = underApplication\} \cup E_{NODELOCKS,R} \cup E_{EDGELOCKS,R}$.
4.1 TIMED GT RULES

3. \( V_{CI,L} = V_{CI,R} = \emptyset \land E_{CI,L} = E_{CI,R} = \emptyset \),

4. \( i_L : DL \rightarrow L \) is an isomorphism \( \land r \mid_{\{V_{G,L},E_{G,L}\}} \) is total and injective,

5. \( N = N_{NODELOCKS} \cup N_{EDGELOCKS}, \)

6. \( z = \emptyset \land V_{res} = \emptyset, \)

7. \( N_{NODELOCKS} = \{(N,n)\mid \exists v \in V_{G,L} : V_{G,N} = V_{G,L} \land E_{G,N} = E_{G,L} \cup \{ne\} \land src(ne) = tgt(ne) = n(v) \land type(ne) = wlnode \circ type(v) \land n \) is injective\} \cup \{(N,n)\mid \forall v \in V_{G,L} \backslash i_L \circ \text{Dom}(dr) : V_{G,N} = V_{G,L} \land E_{G,N} = E_{G,L} \cup \{ne\} \land src(ne) = tgt(ne) = n(v) \land type(ne) = rlnode \circ type(v) \land n \) is injective\},

8. \( N_{EDGELOCKS} = \{(N,n)\mid \exists e \in E_{G,L} : V_{G,N} = V_{G,L} \land E_{G,N} = E_{G,L} \cup \{ne\} \land src(ne) = src \circ n(e) \land tgt(e) = tgt \circ n(e) \land type(ne) = wledge \circ type(e) \land n \) is injective\} \cup \{(N,n)\mid \forall e \in E_{G,L} \backslash i_L \circ \text{Dom}(dr) : V_{G,N} = V_{G,L} \land E_{G,N} = E_{G,L} \cup \{ne\} \land src(ne) = src \circ n(e) \land tgt(e) = tgt \circ n(e) \land type(ne) = rledge \circ type(e) \land n \) is injective\},

9. \( E_{NODELOCKS,R} = \{le\mid \exists v \in V_{G,L} : src(le) = tgt(le) = r(v) \land type(le) = rlnode \circ type(v)\} \cup \{le\mid \exists v \in V_{G,L} \backslash i_L \circ \text{Dom}(dr) : src(le) = tgt(le) = r(v) \land type(le) = wlnode \circ type(v)\}, \)

10. \( E_{EDGELOCKS,R} = \{le\mid \exists e \in E_{G,L} : src(le) = src \circ r(e) \land tgt(le) = tgt \circ r(e) \land type(le) = rledge \circ type(e)\} \cup \{le\mid \exists e \in E_{G,L} \backslash i_L \circ \text{Dom}(dr) : src(le) = src \circ r(e) \land tgt(le) = tgt \circ r(e) \land type(le) = wledge \circ type(e)\}. \)

We describe every condition separately. The LHS of the induced start rule corresponds to the LHS of the durative rule (Condition 1). The RHS of the start rule corresponds to its LHS with an additional node \( a_i \), called application indicator, additional edges from \( a_i \) to all other nodes in the RHS, and additional edges \( E_{NODELOCKS,R} \) and \( E_{EDGELOCKS,R} \) that denote the set of locking edges that are created by the start rule (Condition 2). Intuitively, \( a_i \) indicates the application of the durative rule and the sets \( E_{NODELOCKS,R} \) and \( E_{EDGELOCKS,R} \) indicate whether read or write access to specific nodes and edges is locked. Aside from adding \( a_i \), the start rule applies changes only by creating locking edges. Therefore, the rule morphism \( r \) restricted to graph nodes and edges is total (Condition 4) – \( r \) is also unique (up to isomorphism). The sets of clock instances, clock instance edges, time guards, and clock resets are empty (Conditions 3 and 6) because the start rule does not add a clock instance measuring the execution time itself. Instead, the addition of a clock instance for the execution of a durative rule is done by a clock instance rule.

The remainder conditions implement the locking functionality. Conditions 7 and 8 realize application conditions on the locks by defining NACs on locking edges and Conditions 9 and 10 realize changes to the locking state by creating locking edges. The start rule may not be applied if there is a write lock for a required node or edge; it may further not be applied if there is a read lock for a node or edge that is going to be deleted according to the syntax of the
durative rule, i.e. the node or edge is not contained in $i_L \circ \text{Dom}(dr)$ (Conditions 7 and 8). The start rule creates a read lock for every required node or edge; it creates a write lock if the node or edge is deleted according to the syntax of the durative rule, i.e. if the node or edge is not contained in $i_L \circ \text{Dom}(dr)$ (Conditions 9 and 10).

**Definition 4.4 (Induced End Rule)**

Let $D = (DL, DR, dr, name, d)$ be a durative rule. The induced end rule of $D$ is a timed rule $er = (L, R, r, N, z, V_{res})$ with

1. $V_{G,L} = V_{DL} \cup \{ai\} \land \text{type}(ai) = \text{name} \land E_{G,L} = E_{DL} \cup \{e|\text{src}(e) = ei \land \text{tgt}(e) \in V_{G,L} \land \text{type}(e) = \text{underApplication}\} \cup E_{\text{NODELOCKS},L} \cup E_{\text{EDGELOCKS},L}$
2. $V_{G,R} = V_{DR} \land E_{G,R} = E_{DR}$
3. $V_{CI,L} = V_{CI,R} = \{ci\} \land E_{CI,L} = \{(ci,ai)\} \land E_{CI,R} = \emptyset$
4. $i_L : DL \rightarrow L$ is a total injective morphism $\land i_R : DR \rightarrow R$ is an isomorphism $\land r|_{\{V_{G,L},E_{G,L}\}} = i_R \circ dr \circ i_L^{-1} \land r|_{\{V_{CI,L}\}}$ is total and injective
5. $N = \emptyset$
6. $z = \{ci \geq d\} \land V_{res} = \emptyset$
7. $E_{\text{NODELOCKS},L} = \{le|\exists v \in V_{G,L} : \text{src}(le) = \text{tgt}(le) = v \land \text{type}(le) = \text{rlnode} \circ \text{type}(v)\} \cup \{le|\exists v \in V_{G,L} \setminus \text{Dom}(r) : \text{src}(le) = \text{tgt}(le) = v \land \text{type}(le) = \text{wlnode} \circ \text{type}(v)\}$
8. $E_{\text{EDGELOCKS},L} = \{le|\exists e \in E_{G,L} : \text{src}(le) = \text{src}(e) \land \text{tgt}(le) = \text{tgt}(e) \land \text{type}(le) = \text{rledge} \circ \text{type}(e)\} \cup \{le|\exists e \in E_{G,L} \setminus \text{Dom}(r) : \text{src}(le) = \text{src}(e) \land \text{tgt}(le) = \text{tgt}(e) \land \text{type}(le) = \text{wledge} \circ \text{type}(e)\}$.

The LHS of the induced end rule corresponds to the LHS of the durative rule plus the application indicator node $ai$ and the locking edges (Condition 1). The RHS of the end rule simply corresponds to the RHS of the durative rule (Condition 2). Thus, the application of the end rule removes the application indicator $ai$ and the locking edges that were created when the start rule was applied. The rule morphism $r$ indicates that the end rule realizes the actual graph transformation syntactically defined by the durative rule (Condition 4). Additionally, the end rule includes a time guard on the clock value of $ci$ (Condition 6) to guarantee that the proper amount of time is consumed before being applied. The clock instance $ci$ that is connected to $ai$ is not removed by the end rule (Condition 3) because timed GT rules may neither add nor remove clock instances to or from a timed graph. Adding clock instances is subject to clock instance rules and removing them is subject to a singleton clock instance removal rule. Both are covered in the next section.


### 4.2 Clock Instance and Invariant Rules

Besides timed GT rules that execute reconfigurations, we need rules to create and delete clock instances and to restrict the interval of allowed clock instance values as an invariant condition. This is subject to clock instance rules and invariant rules.

A clock instance rule specifies the subgraph that the clock instance applies to as its LHS. The RHS adds the respective clock instance. The rule specifies a NAC which is the same as the RHS. This prevents that infinitely many clock instances are added to the same subgraph.

**Definition 4.5 (Clock Instance Rule)**

A clock instance rule \( cr = (L, R, r, N) \) consists of two timed graphs \( L \) and \( R \) with a rule morphism \( r : L \to R \) and a negative application condition \( (N, n) \in N \) fulfilling the conditions:

- \( V_{CI,L} = \emptyset \)
- \( |V_{CI,R}| = 1 \land |E_{CI,R}| \geq 1 \)
- \( r(V_{G,L}) = V_{G,R} \land r(E_{G,L}) = E_{G,R} \)
- \( N = R \land n = r \land |N| = 1 \). \([SHS11, EHH+13]\)

The induced clock instance rule has only the application indicator node in its LHS. Thus, it attaches a clock instance only if a start rule has been applied before. Since the application indicator is typed via the name of the durative rule, there is one induced clock instance rule for each durative rule. Figure 6 shows the induced clock instance rule for the durative rule of Figure 5a.

**Definition 4.6 (Induced Clock Instance Rule)**

Let \( D = (DL, DR, dr, name, d) \) be a durative rule. The induced clock instance rule of \( D \), \( cr = (L, R, r, N) \), fulfils the following conditions:

- \( V_{G,L} = V_{G,R} = \{ai\} \land type(ai) = name \land E_{G,L} = E_{G,R} = \emptyset \)
- \( V_{CI,L} = \emptyset \land V_{CI,R} = \{ci\} \land E_{CI,L} = \emptyset \land E_{CI,R} = \{(ci, ai)\} \)
- \( N = R \land n = r \land |N| = 1 \).

Multiple applications of a start rule create multiple application indicator nodes. A clock instance can be attached to each of these nodes. If the subgraph that the clock instance applies to is no longer present in the host graph, the clock instance needs to be removed as well. This is
the case when an end rule is applied as each application of an end rule removes an application indicator. Removing the clock instance is subject to a clock instance removal rule. For a given set of clock instance rules, a clock instance removal rule can be deduced automatically. It has a single clock instance as its LHS and an empty RHS. In addition, it specifies the RHSs of all clock instance rules as NACs. This means that the clock instance removal rule deletes a clock instance if the subgraph that the clock instance applies to is no longer present in the host graph. There is only one clock instance removal rule for the whole timed GTS.

**Definition 4.7 (Clock Instance Removal Rule)**

Let $CR$ be a set of clock instance rules with $cr = (L_{cr}, R_{cr}, r_{cr}, N_{cr}, n_{cr}) \in CR$. A clock instance removal rule $cr_{rem}(CR) = (L, R, r, N)$ is defined by the conditions:

- $V_{G,L} = \emptyset$, $V_{CI,L} = \{ci\}$ for a clock instance $ci$,
- $E_{G,L} = E_{CI,L} = \emptyset$,
- $V_{G,R} = V_{CI,R} = \emptyset$,
- $E_{G,R} = E_{CI,R} = \emptyset$,
- $N = \{(N, n) | \exists cr \in CR : N = R_{cr} \land n : L \rightarrow N \text{ with } ci \mapsto ci_N \text{ and } ci_N \in V_{CI,N}\}$.

Invariant rules forbid the existence of a subgraph beyond a specific point in time. They specify a LHS containing a clock instance and a clock constraint. Whenever the LHS is matched to the graph, the clock constraint must be fulfilled for the clock instance. If the subgraph cannot be destroyed by applying a timed GT rule and time cannot elapse without violating the invariant rule, a deadlock occurs.

**Definition 4.8 (Invariant Rule)**

An invariant rule $ir = (L, z)$ consists of a timed graph $L$ with $|V_{CI,L}| = 1$ and a clock constraint $z \in \mathcal{Z}(V_{CI,L})$. [SHS11, EHH+13]

In Section 4.1 we stated that the induced end rule specifies a time guard $z = \{ci \geq d\}$ on the clock value of its clock instance $ci$. To guarantee that the end rule is indeed applied after $d$ time units instead of being postponed arbitrarily – the time guard only guarantees that it is not applied earlier – the durative rule also induces an invariant rule. Figure 7 shows such an induced invariant rule for the durative rule of Figure 5a.

**Definition 4.9 (Induced Invariant Rule)**

Let $D = (DL, DR, dr, name, d)$ be a durative rule. The induced invariant rule of $D$, $ir = (L, z)$, fulfills the following conditions:

- $V_{G,L} = \{ai\} \land \text{type}(ai) = name \land E_{G,L} = \emptyset$,
- $V_{CI,L} = \{ci\} \land E_{CI,L} = \{(ci, ai)\}$,
- $z = \{ci \leq d\}$.
The induced invariant rule specifies an application indicator $a_i$ and a clock instance node $c_i$ as its only nodes and $c_i \leq d$ as the constraint to be fulfilled whenever the LHS is matched. At every match there is a distinct application indicator that was created by a start rule and a clock instance measuring the elapsed time since the application of its start rule. Intuitively, each match of the LHS indicates that an application of a durative rule is taking place. The invariant rule forbids the existence of an LHS match such that the constraint is unfulfilled. Thus, the invariant rule enforces a timed GT rule that destroys its LHS match to be applied no later than the instant the constraint gets unfulfilled. Destroying the LHS match can only be done by deleting the application indicator $a_i$. This in turn can only be done by an application of the end rule that corresponds to the same durative rule as the invariant rule (since the types of the application indicator nodes have to match). Therefore, the invariant rule guarantees that the end rule is indeed applied after $d$ time units.

4.3 Operational Semantics

We define the semantics of durative GTS, which is simply a standard GTS using durative rules, by a mapping to timed GTS. The definition of the operational semantics of a timed GTS is based on the definitions of the timed GT rule, the clock instance rule, the clock instance removal rule, and the invariant rule. However, before defining the operational semantics, we need a definition of the timed GTS itself.

**Definition 4.10 (Timed Graph Transformation System)**

A timed graph transformation system $G_t$ is a tuple $(G_0, TG, TR, IR, CR)$, where $G_0$ is a timed graph, the initial graph, $TG$ is a type graph, $TR$ is a set of timed GT rules, $IR$ is a set of invariant rules, and $CR$ is a set of clock instance rules. [SHS11, EHH+13]

Note that the definition of a timed GTS does not include a clock instance removal rule in the tuple. The clock instance removal rule $c_{i_{rem}}(C I)$ is implied by the set of clock instances $C I$ according to Def. 4.7.

A durative GTS is an initial graph and a set of durative rules. Its semantics is given by a timed GTS whose timed GT rules $TR$, invariant rules $IR$, and clock instance rules $CR$ are all induced by durative rules. We spare us its formal definition here.

As a basis for our operational semantics we define a configuration of a timed GTS. Intuitively, a configuration consists of a timed graph and an assignment of values to the clock instances of the timed graph.

**Definition 4.11 (Configuration, Initial Configuration)**

A configuration is a tuple $\langle G, \nu \rangle$ where $G$ is a timed graph and $\nu$ is a clock instance value assignment. The initial configuration is the tuple $\langle G_0, \nu_0 \rangle$, where

- $V_{0,CI} = \emptyset$ where $V_{0,CI}$ is the set of clock instances in $G_0$ and
- $\nu_0$ is an empty function.
Based on a configuration of a timed GTS, we can now define the operational semantics of a timed GTS.

**Definition 4.12 (Operational Semantics of a Timed GTS)**

Let $G$ be a timed graph and $G = (G_0, TG, TR, IR, CR)$ a timed graph transformation system. We define

$$I(G) = \bigwedge_{ir \in IR} I_{ir}(G)$$

where for an invariant rule $ir = (L_{ir}, z) \in IR$ and its matchings $m_1, \ldots, m_k$ from $L_{ir}$ to $G$ the function $I_{ir}(G)$ is defined as

$$I_{ir}(G) = z[m_1(ci)/ci] \wedge \ldots \wedge z[m_k(ci)/ci]$$

for all $i = 1, \ldots, k$ and for all $ci \in V_{CI,L_{ir}}$. The operational semantics is defined by a transition system where states are configurations $\langle G, \nu \rangle$. The execution starts in the initial configuration $\langle G_0, \nu_0 \rangle$ and transitions are defined by the following rules:

1. $\langle G, \nu \rangle \xrightarrow{\delta} \langle G, \nu + \delta \rangle$ if $(\nu + \delta) \models I(G)$ for $\delta \in \mathbb{R}^+$. (delay transition)

2. $\langle G, \nu \rangle \xrightarrow{tr,m} \langle G'', \nu' \rangle$ for a timed GT rule $tr = (L, R, r, N, z, V_{res}) \in TR$ and an injective matching $m$ from $L$ to $G$ if $\nu \models z[m(ci)/ci]$ for $ci \in V_{CI,L}$, where
   - $G'$ has been derived by applying $tr$ at $m$ to $G$,
   - $G''$ has been derived by applying $cr_{rem}(CR)$ to $G'$,
   - $G'''$ has been derived by applying all $cr \in CR$ in any order to $G''$, and
   - $\nu' = \nu[V_{res} \mapsto 0]$ with $\nu' \in I(G''')$. (action transition)

The operational semantics defines two kinds of transitions: delay transitions and action transitions. This follows the standard approach as defined for UPPAAL timed automata [BY03]. Delay transitions do not apply rules. Instead, they increase the values of all clock instances synchronously. As a condition for the transition the new clock instance value assignment has to satisfy $I(G)$ which is the conjunction of all invariant clock instance constraints. While firing a delay transition in a configuration with no clock instance is possible, it has no effect, i.e. it produces a self-loop in the state space. Action transitions are defined by the application of timed GT rules. With each application of a timed GT rule, clock instances are created and destroyed according to the clock instance rules to create a successor configuration. In the presence of durative rules, a clock instance is created when applying an induced start rule and destroyed when applying an induced end rule.

### 5 Properties of the Semantics

In this section we argue that our semantics of durative rules is well-defined and possesses properties that one might expect. Specifically, we cover two properties. The first property states that we can project the application of a timed action transition to the untimed case, i.e. the semantics of durative rules is a conservative extension of the semantics of untimed GTS.
The second property states that each durative rule application terminates properly. This is ensured by the local confluence of the involved start and end transformations.

Before formalizing the first property, we define a lemma stating that, after executing the start transformation of a durative rule and letting its duration pass, its end transformation can be applied. This lemma is used in the proofs of both properties.

**Lemma 5.1 (Applicability of Induced End Rule)**

Given a durative rule \( \mathcal{D} = (DL, DR, dr, name, d) \), a timed graph \( TiG = (G_{TiG}, type_{TiG}) \) with \( G_{TiG} = (V_G, \emptyset, E_G, \emptyset) \), and a valuation \( \nu = \emptyset \). Let \( sr \) denote the induced start rule of \( \mathcal{D} \) and \( er \) its end rule.

If there is a match \( m_{sr} : L_{sr} \rightarrow TiG \) and a direct derivation \( \langle TiG, \nu \rangle \xrightarrow{st,max} \langle TiG', \nu' \rangle \), then there is a unique (up to isomorphism) match \( m_{er} : L_{er} \rightarrow TiG' \) and direct derivations \( \langle TiG', \nu' \rangle \xrightarrow{d} \langle TiG'', \nu'' \rangle \xrightarrow{st,max} \langle TiH, \nu \rangle \).

**Proof.** According to Def. 4.12, the application of \( \langle TiG, \nu \rangle \xrightarrow{st,max} \langle TiG', \nu' \rangle \) creates a clock instance \( ci \) and \( \nu'(ci) = 0 \). The induced invariant rule \( ir = (L_{ir}, z_{ir}) \) now has a match \( m_{ir} : L_{ir} \rightarrow TiG' \) with \( m_{ir}(ci_{ir}) = ci \) to the current host graph \( TiG' \). Since this is the only invariant rule with a match to the host graph, \( I(G) = z_{ir}[ci/ci_{ir}] = \{ci \geq d\} \). Since \( (\nu' + d) \models I(G) \), \( \langle TiG', \nu' \rangle \xrightarrow{d} \langle TiG'', \nu'' \rangle \) is applicable.

By applying the derivation, we obtain \( \nu'' \) with \( \nu''(ci) = d \) and thus fulfil the time guard \( ci \geq d \) of \( \langle TiG'', \nu'' \rangle \xrightarrow{st,max} \langle TiH, \nu \rangle \). Since \( L_{er} = R_{sr} \) and \( R_{sr} \) is embedded into \( TiG' \), \( m_{er} \) can be defined; due to the application indicator \( ai \), its adjacent edges, and the locking edges, \( m_{er} \) is unique (up to isomorphism). Then we can construct \( TiH \) according to Def. 4.12. Since the \( ci \) is removed by an application of the clock instance removal rule, the new valuation is \( \nu = \emptyset \). \( \square \)

Next, we formalize the first property as a theorem. More precisely speaking, the theorem says that the execution of a durative graph transformation \( \langle TiG, \nu \rangle \xrightarrow{D,m} \langle TiH, \nu \rangle \) results in a graph that is structurally identical to the graph we receive by executing an untimed graph transformation that is defined by the same rule morphism and matching.

**Theorem 5.1 (Conservative Extension of GTS)**

Let \( \mathcal{D} = (DL, DR, dr, name, d) \) be a durative rule, \( TiG = (G_{TiG}, type_{TiG}) \) a timed graph with \( G_{TiG} = (V_G, \emptyset, E_G, \emptyset) \), and \( \nu = \emptyset \) a valuation. Let \( sr \) denote the induced start rule of \( \mathcal{D} \) and \( er \) its end rule. Further, let the GT rule \( r = (DL, DR, dr) \) and the graph \( G = (V_G, E_G) \) be a projection of \( \mathcal{D} \) and \( TiG \) to the untimed case.

If and only if there is a match \( m : DL \rightarrow G \) and a direct graph transformation \( G \xrightarrow{r,m} H \), then there are matches \( m_{sr} : L_{sr} \rightarrow TiG \) and \( m_{er} : L_{er} \rightarrow TiG' \) and direct derivations \( \langle TiG, \nu \rangle \xrightarrow{st,max} \langle TiG', \nu' \rangle \xrightarrow{d} \langle TiG'', \nu'' \rangle \xrightarrow{st,max} \langle TiH, \nu \rangle \) such that

\[
H = (V_H, E_H) \quad \text{and} \quad TiH = (H_{TiH}, type_{TiH}) \quad \text{with} \quad H_{TiH} = (V_H, \emptyset, E_H, \emptyset).
\]

First we show that, given the match $m$, we can define the matches $m_{sr}$ and $m_{er}$ such that $H = TiH|_{\{V_H, E_H\}}$. Since $L_{sr} = DL$ and $TiG|_{\{V_G, E_G\}} = G$, we can set $m_{sr} = i_G \circ m \circ i^{-1}_{sr,L}$. Then we can construct $TiG'$ and $\nu'$ according to Def. 4.12. $TiG'$ now contains one clock instance $ci$ and $\nu'(ci) = 0$.

According to Lemma 5.1, there is a unique (up to isomorphism) match $m_{er}$ and direct derivations $(TiG', \nu') \xrightarrow{er,m_{er}} (TiH, \nu)$. Since $m_{sr} = i_G \circ m \circ i^{-1}_{sr,L}$ and $m_{er}$ is unique, $m_{er} = i_G \circ m \circ i^{-1}_{er,L}$. Hence, $H = TiH|_{\{V_H, E_H\}}$. Furthermore, since $L_{er} = R_{sr}$ and $r_{er}|_{\{V_G, E_G, L\}} = i_{er,R} \circ dr \circ i^{-1}_{er,L}$, the derivation removes the locking edges, the application indicator node $ai$, and the only clock instance $ci$ to construct $TiH$.

Now we show that, given the matches $m_{sr}$ and $m_{er}$, we can define the match $m$ such that $H = TiH|_{\{V_H, E_H\}}$. Since $DL \subseteq L_{er}$, we can define $m = i_G^{-1} \circ m_{er} \circ i_{er,L}$. Then we can construct $H$ according to the SPO approach. Since $dr = i_{sr,R}^{-1} \circ r_{er}|_{\{V_G, E_G, L\}} \circ i_{er,L}$ and $m = i_G^{-1} \circ m_{er} \circ i_{er,L}, H = TiH|_{\{V_H, E_H\}}$. \hfill □

Here, we assume that no other durative graph transformation is executed concurrently during the time span of $(TiG, \nu) \xrightarrow{D,m} (TiH, \nu)$. Intuitively, the resulting graphs are identical because

- executing the start transformation leaves the essential parts of the graph unchanged,
- all locking elements and special nodes that are created by executing the start transformation are deleted again by executing the end transformation, and
- the end transformation realizes a graph transformation that conforms to the untimed graph transformation; in fact, their RHSs are the same.

Next, we formalize the second property which states that each durative rule application terminates properly. In other words, reconfigurations can only be executed concurrently if they do not interfere with each other.

Theorem 5.2 (Termination of Rule Application)
Let $D = (DL, DR, dr, name, d)$ be a durative rule, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, \emptyset, E_G, \emptyset)$, and $\nu = \emptyset$ a valuation. Let $sr$ denote the induced start rule of $D$ and $er$ its end rule.

If there is a match $m_{sr} : L_{sr} \rightarrow TiG$, a direct derivation $(TiG, \nu) \xrightarrow{sr,m_{sr}} (TiG', \nu')$ with $\nu'(ci) = 0$, and a derivation $(TiG', \nu') \xrightarrow{seq} (TiG'', \nu'')$ with $\nu''(ci) = d$ and $(er, _) \notin seq$, then there is a unique (up to isomorphism) match $m_{er} : L_{er} \rightarrow TiG''$ and a direct derivation $(TiG'', \nu'') \xrightarrow{er,m_{er}} (TiG'''', \nu'''')$.

Proof. We show this property by induction over the number of action transformations in $seq$, i.e. delay transformations are excluded.
Basis step. We start with $n = 0$, i.e. $seq$ does not contain any action transformation. The match $m_{er}$ and the direct derivation $\langle TiG'', \nu'' \rangle \xrightarrow{er,m_{er}} \langle TiG'', \nu'' \rangle$ directly result from Lemma 5.1.

Induction step. We assume that there is a unique (up to isomorphism) match $m_{er} : L_{er} \rightarrow TiG''$ and a direct derivation $\langle TiG'', \nu'' \rangle \xrightarrow{er,m_{er}} \langle TiG'', \nu'' \rangle$ and have to show that if there is a derivation $\langle TiG'', \nu'' \rangle \xrightarrow{seq'} \langle TiH, \xi \rangle$ with $\xi(\epsilon) = d$ and $(er, \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ ..
• another start transformation that is applied during the interval only adds elements to the
host graph but deletes none, thus cannot conflict with the applicability of the end rule,

• another end transformation that is applied during the interval and consumes locking
elements would also imply the execution of another start transformation that created
such locking elements earlier in the interval, thus providing exactly the same (up to
isomorphism) locking elements as if none of the two transformations were applied, and

• elements supposed to be deleted by the application of the end rule cannot be deleted by
another durative graph transformation because the start transformation attached locking
elements to them.

Note, that Theorem 5.2 requires that no clock instance exist in the run-time state \( \langle T_iG, \nu \rangle \)
i.e. \( \nu = \emptyset \). In other words, for every start transformation of a durative rule that has been
applied, its end transformation also has been applied to arrive at \( \langle T_iG, \nu' \rangle \). However, we
can use the Local Confluence Theorem for GTS with NACs, cf. Habel et al. [HHT95], to
easily gain properties where durative rules are allowed to be in ongoing execution. The Local
Confluence Theorem states that two parallel independent (direct) graph transformations can
be applied in any order and both orderings result in the same graph. By showing that the start
transformation \( \langle T_iG, \nu \rangle \xrightarrow{sr,max} \langle T_iG', \nu' \rangle \) and a succeeding start transformation are sequential
independent, we can thus prove that they may be reordered. This relaxes Theorem 5.2 in the
sense that the first start transformation in \( \text{seq} \) may be pulled before \( \langle T_iG, \nu \rangle \xrightarrow{sr,max} \langle T_iG', \nu' \rangle \),
i.e. durative rules are allowed to be in ongoing execution in \( \langle T_iG, \nu \rangle \).

6 Related Work

In literature, there exist other approaches supporting durations or time in graph transforma-
tions. The approach by Rivera et al. [RDV09] provides timed graph transformations where
rules specify a duration and periodicity for their execution. If a match is found, the effect of
the rule is established at the end of the duration like in our approach. In contrast to our ap-
proach, they do not explicitly lock the match of a rule, i.e., the matches of rules which are cur-
cently executing may be changed and rules may be cancelled. Real-Time Maude [ÖM07] is a
model-checker for object-oriented graph rewrite rules in a textual syntax. It provides support
for discrete and dense real-time models, although verification is only possible for discrete time
models. In contrast to our approach, they do not consider durations of rules. The MOMENT2
framework [BÖ10] provides model transformations based on MOF meta-models in a textual
syntax. The approach supports one unresetable clock per object, timers that count down to 0
and timed values that can be increased or decreased at a certain, fixed rate based on the elapsed
time. The model transformations can be simulated and verified using Real-Time Maude. The
approach by Michelon et. al. [MdCR07] specifies a graph formalism for specifying message
exchange between objects. Each message has a time information on when it is delivered and
handled. Rules have no duration and their modelling formalism relies on their application domain, because each rule needs to consume at least one message. De Lara et al. [dLV10] add timing information to attributed graph transformation rules. Each rule specifies a time interval in which it may be applied, but no duration. For analysis purposes, the system is translated to timed Petri nets which support durations for transitions. In [dLGB+12], a discrete event simulation based on timed GTS is presented where events encode the point in time when the graph transformation rule is executed. Rules, however, can be grouped in uninterruptable activities where the execution of one rule requires the execution of another one at later point in time. This is similar to the induced timed rules of a durative rule. In contrast to our approach, they ensure the execution of the second rule in their simulation framework rather than in the GTS itself and provide no rule durations. GROOVE [KR06], which is probably the most well-known graph-based verification tool, does neither support time nor durations for rules.

7 Conclusion

In this paper, we presented an extension of graph transformation systems by durations for rules. In addition, we introduced a locking approach that allow for safe concurrent execution of durative rules. In particular, we have proven that a durative rule can always be finished correctly once it has been started. We enable the formal verification of graph transformation systems with durative rules by defining a mapping to timed graph transformation systems. Timed graph transformation systems may serve as an input to our timed graph verification framework [EHH+13].

The extension of graph transformation rules by durations enables developers to provide a more precise specification of reconfiguration behaviour by considering that such behaviour needs time for being executed. In addition, GTS with durative rules provide an intuitive and easy-to-use modelling approach to specify concurrent behaviour. Besides formal verification, we have shown in [ZW13] that durative rules can also be translated into a planning specification. That enables using the same specification for planning reconfiguration orders.

In our future works, we plan to extend the semantics to support negative application conditions on the level of durative rules. While simple forbidden links can be locked by the same locking edges than preserved links, a NAC that forbids the link of an object to any other object of a certain type requires the definition of new kinds of locks to block their concurrent creation. Another idea for future research involves positive influences between durative rules: instead of blocking other rules from being applicable, a durative rule could enforce another rule to be applied during its own application interval.

References


Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. Fundamenta Informaticae, 26:287–313, 1995.


