Synthesizing Modal Sequence Diagram Specifications with UPPAAL-TIGA

Technical Report tr-ri-10-310

Joel Greenyer*
Software Engineering Group, Department of Computer Science, University of Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
jgreen@upb.de
February 10, 2010

Abstract

During the design of software-intensive systems it is crucial to specify requirements formally and to detect inconsistencies early in order to avoid flaws in the final product or costly iterations in the development. Live Sequence Charts (LSCs) and a recent variant, Modal Sequence Diagrams (MSDs) are a rich, visual formalism for specifying requirements on the interaction of components in a system. Because inconsistencies are easily introduced in a scenario-based specification, approaches for their synthesis have been presented previously. But, the synthesis of scenario-based specifications, especially under the consideration of time, is an inherently complex task. This paper presents a novel, efficient technique the synthesis of a timed or untimed MSD-specification by mapping it to the input for UPPAAL TIGA. UPPAAL TIGA is an extension to the UPPAAL model checker which implements an efficient, on-the-fly algorithm for synthesizing winning strategies in dense-time or untimed two-player games. The approach is not complete because a resulting strategy is not always a valid implementation of the specification, but additional measures can be taken for finding a valid one. In many cases, however, the algorithm efficiently calculates useful results and if no strategy can be found, this indicates that the specification is inconsistent. The mapping is formalized and implemented by a set of Triple Graph Grammar rules which map a UML-based MSD specification to an input model for UPPAAL TIGA.

1 Introduction

From household machines to transportation systems, we find software-intensive system everywhere around us today. These systems very often fulfill highly

*supported by the International Graduate School Dynamic Intelligent Systems.
safety-critical tasks, while at the same time there is an increasing demand on the functionality of these systems. In addition, these systems become increasingly interconnected, and, especially in software which controls physical processes, timing aspects become an important concern. To manage the increasing complexity of such systems and to design them in a time- and cost efficient manner, it is essential to provide software and system engineers with intuitive, but precise means for developing these systems. Furthermore, it is important that automated analysis techniques and formal reasoning can be employed already early in the development in order to detect flaws in the system design and to avoid costly iterations.

A vision of software and system engineers is to be able to intuitively specify the requirements of such systems and then to automatically check whether a system with these requirements is realizable and, moreover, to automatically derive an implementation for the controllers of such systems that adhere to these requirements. This will probably not be possible on a grand scale in the foreseeable future, but combining efficient algorithms with precise, but intuitive specification languages for certain kinds of requirements can bring us a step closer to this vision.

Large parts of the requirements of software-intensive systems are requirements on its discrete reactive behavior, which are requirements of the actions of the system in reaction to events in environment. For the reactive behavior, there exist algorithms for deriving controllers from requirements or showing that there cannot be any because of contradictions in the requirements. This task, called the synthesis of controllers is an inherently complex task; Pnueli and Rosner have shown that the general problem of synthesizing a controller from a set of LTL formulae takes double-exponential time in the length of the formulae [PR89].

The requirements for reactive systems can be formally specified for example by using temporal logic formulae, automata, or process calculi. One formalism that is particularly intuitive and appealing because of its graphical syntax is Live Sequence Charts (LSCs), invented by Damm and Harel in 1999 [DH01]. LSCs are a formalism for specifying requirements on the interaction of system and environment components in a graphical way. LCSs are a kind of sequence diagrams that allow us to not only describe sequences of events that may occur, but also such that must or must not occur during the interaction of the system and its environment. To express this, LSCs allow us to distinguish the modality of the charts between existential and universal charts. Existential charts describe sequences of events that must be possible to occur during a run of a system whereas universal charts describe sequences of events that must occur when triggered by a certain event or by a sequence of preceding events. Moreover, the events in universal charts may be cold events, which are events that may occur, and hot events, which must eventually occur. The principle of LSCs was partly adopted by the sequence diagrams in version 2.0 of UML [UML05] by introducing fragments like assert and negate. Harel and Maoz have uncovered ambiguities of the sequence diagram semantics given in UML2.0 and proposed a formalized interpretation of the UML sequence diagrams, called Modal Sequence Diagrams (MSDs) [HM08]. MSDs inherit most concepts of LSCs, but
their syntax is based on UML. The work in this paper is based on a slightly adapted form of MSDs. In the following, I simply use the term diagram for referring to a single MSD.

When the behavior of the system is specified by a set of MSDs, these MSDs define the admissible runs of the system. But, what’s more is that we can operationalize an MSD-specification by interpreting its universal diagrams: on external events, one or more charts may become active, thereby postulating a set of enabled events that the system should (cold) or must (hot) perform next. This algorithm is called play-out and was developed for LSCs by Harel and Marelly \cite{HM03}. During the system design, the play-out algorithm allows the engineer to simulate the interaction of the system and environment components. This is a powerful technique to understand the specification, which typically consists of multiple, cross-cutting requirements that are often contributed by different engineers and stakeholders.

At times, play-out may lead to situations where diagrams require contradicting events to be performed next and the algorithm cannot progress beyond hot messages or conditions in the diagrams. This is called a hot violation. When we cannot assure to always avoid hot violations or, in other words, when there are sequences of environment events where always a hot violations occurs, the specification is inconsistent.

The problem with the play-out algorithm is that the algorithm often has to make non-deterministic choices between concurrently enabled events, and it does so quite naively; hot violations during play-out could often be avoided when the algorithm chose another possible sequence of events. But the play-out algorithm cannot "look ahead" to avoid hot violations. This is a major problem of play-out, because, when hot violations occur, we never know whether the specification is really inconsistent or not.

To help avoid violating steps, an improved play-out algorithm, called smart play-out was developed by Harel et al. \cite{HKMP02}. Smart play-out can look ahead a limited number of steps, called a super step, which is the steps that the system performs until it waits for the next external event to occur. This is, however, still a limitation and smart play-out may still execute violating sequences of system steps while there are admissible system reactions. Thus, ultimately, we desire a method for finding out for sure whether the system can avoid violations under all circumstances (all possible sequences of environment events) or whether that is not possible. This can be achieved by the synthesis of the LSC specification.

The problem of synthesizing LSC requirements has been addressed previously. Early approaches of synthesizing LSCs are based on calculating intersection automata representing all possible runs of the LSC specification and then removing such states for which the system cannot assure to always avoid violations. The specification is inconsistent when the resulting automaton is empty \cite{HK99}. Since the state space can become very large, the approaches that explicitly construct the whole state space are not efficient for bigger LSC specifications. More recent approaches for synthesizing LSC specifications are based on mapping the LSC synthesis problem to the problem of finding winning strategies in parity games \cite{BS03, BH05, KPP09}. The advantage of this
approach is that existing, efficient algorithms for synthesizing game strategies exist that, for example, use symbolic representations of the state space. This makes these approaches very promising to be used also for bigger specifications.

Algorithms have been proposed for two kinds of synthesis that we need to distinguish. Synthesis can be interpreted in the sense of finding a satisfying strategy for the play-out (simulation) of an LSC/MSD specification, as pointed out above. The long term goal, however, and the more general meaning of synthesis is finding a valid implementation for each system component that are subject to the specification. The first kind of synthesis is also called the synthesis of a global controller that can only in some cases be used as an implementation for the system, for example if the system is running on a single hardware processor. The second kind of synthesis is the synthesis of distributed controllers, which requires additional concepts. However, even when the global controller cannot directly serve as an implementation of a distributed system, synthesizing a global controller can result in a controller strategy that can be the basis for the development of a distributed controller or, if it turns out that no global strategy exists, it tells us that the specification is inconsistent an there can neither exists any distributed implementation.

In this paper, I present an approach for synthesizing a global controller from a specification in the form of dense-timed or untimed universal MSDs by mapping them to the input for UPPAAL Tiga $^{\text{CDF+05, BCD+06, BCD+07a}}$. UPPAAL-Tiga is a tool for synthesizing winning strategies for untimed and timed two-player games. For this purpose, UPPAAL-Tiga extends the timed automata of the UPPAAL model checker to Timed Game Automata (TGA). In TGA the set of transitions is divided into such that are controllable by the system and such that are uncontrollable. The uncontrollable transitions are controlled by the opponent, which is the system’s environment in this case. UPPAAL Tiga allows us to check whether there exists a strategy for the system to always satisfy reachability or safety properties, expressible in a form of TCTL. If a winning strategy exist, the algorithm can calculate an arbitrary or even a time-optimal strategy. If a winning strategy does not exist, UPPAAL Tiga calculates a winning strategy for the environment, which helps in understanding how the properties can be violated. The synthesis algorithm of UPPAAL Tiga is an on-the-fly algorithm, which means that it does not necessarily explore the whole state space of the synthesis problem.

In this paper I show how to formulate the problem of synthesizing timed and untimed MSD specification as a problem of finding a winning strategy that is satisfying a certain winning condition in a system of Timed Game Automata. Furthermore, I present an implementation of the approach by providing a formal mapping from MSD specifications to the system of TGA. The mapping is specified by a Triple Graph Grammar (TGG) $^{\text{Sch+94}}$, which is a graphical, rule-based formalism for describing the relation of different kinds of models. The TGG-based mapping can be executed by a transformation engine that interprets the TGG rules $^{\text{Gre+05, GK+10}}$.

An algorithm for synthesizing dense-time MSD specifications has been described previously by Plock $^{\text{Plo08}}$, but, these concepts were not implemented. Also, in addition to the work of Plock, this paper presents basic concepts for
including environment assumptions in the synthesis, which is very important for the practical use of MSD specifications with time.

At the same time of developing the synthesis technique presented here, a similar approach was worked out by Larsen et al. \cite{LLNP10}. Larsen et al. present a mapping from LSCs to a system of Timed Game Automata for consistency checking with Uppaal Tiga. The differences and similarities of that approach with the one presented here are discussed in Sect. 8.1.

The synthesis approach presented here is however not complete due to the restrictions on winning conditions that are expressible in Uppaal Tiga \cite{BCD+07a}. Therefore, due to a simplified winning condition, not each strategy produced by the synthesis algorithm is a valid implementation of the specification. The synthesis problem as formulated by Harel et al. \cite{HKP05} states that a specification is consistent if a strategy exists such that the system can always find a way to eventually terminate all active diagrams, which includes that the system is not “stuck” anywhere because it cannot progress beyond contradicting hot messages or conditions. This winning condition, however, uses a combination of temporal operators (globally and eventually), which cannot be expressed in Uppaal Tiga. In the approach presented here, the winning condition for the system is to always avoid steps which produce hot violations. While this intuitively sounds to be exactly what we want, the synthesis algorithm may find a strategy where the system remains in an infinite cycle which indeed avoids violating steps, but where always MSDs remain active. Such strategies are not valid implementations of the MSD specification.

To avoid such cycles, for example a counter and an upper bound for the number of possible system steps in reaction to an environment event could be added to the encoding of the synthesis problem. This is, however, not a general concept. Instead, because the algorithm implemented by Uppaal Tiga can also produce all possible strategies, an additional analysis of this extended result is possible to retrieve an admissible strategy (one where always eventually a state is reached where environment events are enabled again) or to determine that no such strategy exists, which would reveal that the specification is in fact inconsistent. But, these measures are not elaborated in the scope of this paper.

Even though the approach does not always produce valid strategies, it can be expected that the algorithm efficiently produces useful results in many cases, that is cases where there are no loops of self-triggering MSDs. Then it can be expected that results are produced in a time that is linear to the number of reachable super-cuts. Further, if the algorithm reports that no winning strategy exists, this indicates that the specification is in fact inconsistent.

The paper is structured as follows. Section 2 presents the example case study considered in this paper. A sketch of the synthesis approach is then presented in Section 3. Section 4 introduces the foundations of MSDs, followed by a description of the basic concepts of Uppaal Tiga in Section 5. I then present the mapping from an untimed or a timed MSD specification to a system of TGA Section 6. The formalization and implementation of this mapping is presented in Sect. 7. At last, I conclude and give an outlook on embedding this synthesis approach in a tool suite for the scenario-based design of mechatronic systems.
2 Example

The example considered in this paper is taken from the context of the project “Neue Bahntechnik Paderborn/RailCab” short RailCab. RailCab is an innovative concept for the rail-bound traffic of tomorrow. The vision is that trains no longer run on schedule, but small, autonomous vehicles, called RailCabs, transport passengers and goods on demand. To reduce the energy loss due to wind resistance, the RailCabs may form convoys when traveling in the same direction. Furthermore, each RailCab is driven by a linear drive, which permits to build light vehicles and enables a power supply of the RailCabs without overhead power lines or contact rails. Figure 1 illustrates the vision of the RailCab project. The images on the right show the RailCab test track that is built at a scale of 1:2.5 at the University campus in Paderborn.

![passenger RailCab](image1)
![cargo RailCab](image2)
![convoy formation](image3)

Figure 1: The RailCab system: On-demand transportation of passengers and goods by autonomous rail vehicles

2.1 Example use cases in the RailCab project

A lot of the functionality of such an autonomous transportation system is realized by software. Figure 2 illustrates different use cases in which a RailCab interacts autonomously with other RailCabs and further components of the railway track system. Occurrences of these use cases and the involved system components are depicted by the dashed ellipses and the attached connectors. For example, a RailCab may detect another RailCab driving ahead and may decide to form a convoy with it. RailCabs may also break up the convoy again, for example when one RailCab takes another direction than the other members of the convoy. The principle of the RailCab’s two-phase linear drive \([HVB+05]\) requires that it constantly exchanges information about its current position and its desired speed with a track section control. A track section control is a component that, among other things, controls the phase of the magnetic field that is produced by conductors in the track bed. In Figure 2, the track section controls are illustrated by the antenna attached to a track section. When a RailCab drives from one track section onto another, it is therefore necessary that the

\(^{1}\)“Neue Bahntechnik Paderborn”, http://www-nbp.upb.de
2.2 Two example use cases

RailCab registers at the next track section control. Furthermore, the RailCab has to ask for permission to enter the next track section. It may be that, for example, a hazard occurred on the next track section such that, for safety reasons, the RailCab may not enter. Branching and merging switches are also controlled by section controls. In the case of a merging switch, a switch control must for example ensure that two RailCabs, when unaware of each other, do not enter the switch at the same time. At branching switches, the RailCab has to decide which route to take.

![Diagram of RailCab system](image)

Figure 2: An illustration of different application scenarios in the RailCab system.

2.2 Two example use cases

There are of course many more use cases, but let us consider two complementary use case of a RailCab entering a merging switch in more detail. In the second, also timing aspects become relevant.

I present this example in a simplistic form in order to illustrate how requirements given in the form of natural language text can be modeled formally using timed Modal Sequence Diagrams (MSDs). Furthermore, I show how an MSD specification can be inconsistent and how the developer can modify the MSD specification to resolve this inconsistency. In Sect. 2.3 I argue in which ways smart play-out fails in detecting such inconsistencies, or rather that it fails in showing their absence. This leads to the introduction to the synthesis approach in Section 3.

The initial situation of the use case of a RailCab driving onto a merging switch (“drive onto merging switch”) is illustrated in Fig. 3. This use case actually includes another, more general use case, namely the use case “drive onto next track section”. As explained previously, before a RailCab is allowed to drive onto the next track section, it has to register at the next section control and request the permission to enter. This also holds when the next track section is a switch. Additionally, there are specific requirements when the next track section is a merging switch. It is for example required that the reply of the track section control is not sent too early, because, under certain circumstances, other RailCabs approaching the switch shall be given the right of way. For example, a convoy shall have the right of way over a single RailCab (unless they coordinate for merging the single RailCab into the convoy). Also, emergency RailCabs shall be given the right of way in any case.
In the following, let us consider a simplified form of the “drive onto track section” and “drive onto merging switch” use cases. The first use case is free of time constraints, which are added with the second use case. Informally, the use case “drive onto track section” is described as follows.

**Use Case: Drive onto track section**

When the RailCab approaches the end of the track section, the RailCab must send a request to enter the next track section to the section control responsible for the next track section. Then the section control must reply, stating whether entering the track section is currently allowed or not. The reply must be received before the RailCab reaches the point of the last safe break.

**Environment assumption:** When the RailCab is driving on a track section, it is at some point notified that it approaches the end of the track section. After that, the RailCab is notified about reaching the point of the last safe break and then the point of no return. (Beyond the point of no return, we cannot guarantee that breaking will safely stop the RailCab before it enters the next track section.) At last, the RailCab enters the next track section.

In contrast to the previous use case illustrations, the environment now becomes more explicit: We assume that certain events occur in the environment, namely that the RailCab is reaching certain positions on the track section in a certain order. In Fig. 3, these external events are named endOfTS, lastBreak, noReturn, enterNext. In reality, these events are actually generated by sensors and other protocols at a lower system level. However, from the point of view that we take in this use case, these events are environment events that we cannot control.

The use case structure can be captured by the diagram shown in Fig. 4. On the left, there is the object env, which can send the messages endOfTS, lastBreak, noReturn, and enterNext to the RailCab. On the right, there are the relevant system objects in this use case. These are the RailCab (rc) and the section control that controls the next track section (next). The physical track section that the RailCab is driving on, or the one it plans to enter next, is not actively involved in the described interaction and therefore we do consider them here. For simplicity, we neither consider the section control of the current
Two example use cases

track section and we omit that the RailCab would have to break in case that entering the next track section is not allowed.

In the scope of this paper, only such static object systems are considered; I discuss in Sect. 8 in which way the synthesis of static systems help in engineering dynamic systems.

The messages that the RailCab and the section control objects interchange are requestEnter and enterAllowed. The use case states that the RailCab may be allowed to enter or not. To express this, we may introduce another message, like enterNotAllowed, or we can parameterize the enterAllowed message by a Boolean variable. For the sake of simplicity, however, I do not consider parameterized messages here.

![Figure 4: Structure of the Example Use Case “RailCab drives onto merging switch”](image)

Based on this structure, we can express the interaction described in the use case. We could do that by using UML sequence diagrams. For example, taking the first paragraph from the use case description above, the resulting sequence diagram could be the one shown in Fig. 5. We use the assert fragment to reflect the mandatory requirements that are expressed in the use case. That is, when the environment sends the endOfTS message to the RailCab, the RailCab must send requestEnter to the next section control, which must in turn reply by sending enterAllowed. The assert fragment means that the sequence contained in it “are the only valid continuations. All other continuations result in an invalid trace” of the interaction (see the UML2.2 Superstructure specification on the assert fragment [UML09, p.472]).

![Figure 5: UML sequence diagram: RailCab requests the permission to enter the next track section](image)

Harel and Maoz argue that the UML specification provides an imprecise semantics of a number of sequence diagram constructs. In case of the assert
fragment, for example, it is not exactly clear which traces are valid and which ones are not [HM08]. To resolve this problem, Harel and Maoz propose Modal Sequence Diagrams (MSDs), to provide a precise semantics for UML sequence diagrams, inspired by Live Sequence Charts. I give a more precise summary of MSDs in Section 3.

There is an alternative notation of MSDs, which is for its more concise notation favored by their authors over the UML notation. Figure 6 shows the interaction in the MSD notation. Here messages that may occur, called cold messages, are represented by blue dashed arrows and messages that must occur, called hot messages, are represented by a red solid arrow. Note that all the messages appearing in the charts are synchronous messages, which is expressed by the solid arrowhead. That means that sending and receiving the message is one event, it happens at the same time. In reality, however, the communication of the RailCab and the section controls takes place via an asynchronous channel (WLAN in this case). If we want to express, for example, that time passes or other events may take place between the sending and the receiving of the message, the message should be modeled as an asynchronous messages. But, for simplicity, I only consider synchronous messages in the scope of this paper.

![Modal Sequence Diagram: RailCab requests the permission to enter the next track section.](image)

Figure 6: Modal sequence diagram: RailCab requests the permission to enter the next track section.

The requirements expressed in the second paragraph of the above use case can be modeled by extending the MSD of Fig. 6 or by specifying another MSD. For the sake of introducing MSDs, let us model a second MSD, which is shown in Fig. 7. This MSD partly overlaps with the previous MSD by stating that the message enterAllowed must occur in reply to the message requestEnter. Additionally, we require that the reply must be sent before the environment sends the lastBreak message to the RailCab. This way it is a violation if the RailCab sends requestEnter to the section control and lastBreak happens before enterAllowed is returned to the RailCab.

In addition to the requirements expressed in the use case, environment assumptions are formulated. Environment assumptions can make it easier to specify a valid system behavior, because we can express that the environment cannot do arbitrarily nasty things so that system is forced into an invalid state. Often it is even impossible to fulfill given system requirements without making assumptions about the environment. However, as discussed previously, the environment assumptions are often additional requirements for the lower level system behavior. So, by making too optimistic environment assumptions for one part of the system, we might just defer problems to another part of the
2.2 Two example use cases

Figure 7: MSD specifying that the reply of the track section control must be sent before reaching the point of the last safe break.

The assumption formulated for the above example use case says that there are certain points on the track section that the RailCab passes in a certain order (Fig. 3 above). The use case description says that the events endOfTS, lastBreak, noReturn, and enterNext are sent in this sequence. To capture these environment assumptions formally, we model the automaton shown in Fig. 8. The transitions are displayed as dashed arrows to express that these are environment events that we cannot control. The assumption encoded here is actually an example of a too optimistic assumption: what if the RailCab reverses and then drives forward again? Then it may happen that subsequences of these environment events occur repeatedly. For now, however, in order not to complicate the example, we stick to this over-optimistic environment model.

Figure 8: The automaton representing the environment assumption of the “drive onto track section” use case.
The reader may wonder why I choose to switch the modeling paradigm here from using MSDs to specify the system behavior to automata for describing the environment assumptions. That is because MSDs (or LSCs) are intended for specifying which interactions shall take place in reaction to an environment or system event. It is not possible to describe that something can spontaneously happen. Some ideas of using MSDs for specifying environment assumptions are discussed in Sect. 6.4.

Next, let us consider the use case “drive onto merging switch”.

**Use Case: Drive onto merging switch**

**Requirements:** If the RailCab drives onto a merging switch, the reply of the next section control, stating whether entering the track section is allowed, must not be sent more than 5 seconds before entering the switch. This is because other RailCabs entering the switch from the other direction may have the right of way, e.g. large RailCab convoys or emergency RailCabs.

**Environment assumption:** When the RailCab is notified to be approaching the end of the track section, it takes between 8 to 12 seconds until it effectively enters the next track section. Furthermore, when the RailCab is notified to be approaching the end of the track section, it takes at least 5 seconds until it passes the point of the last safe break.

The MSD in Fig. 9 describes that the reply of the track section control, `enterAllowed`, must not be sent more than 5 seconds before entering the switch (receiving `enterNext`). This kind of MSD, which describes that a certain sequence of events must not happen, is called an *anti-scenario*. Typically, an anti-scenario contains cold messages and conditions that end in a hot condition with the value `false`. That means that a hot violation occurs when the end of the diagram is reached. This way, anti-scenarios allow us to express that the system must have a way to violate a cold fragment of the anti-scenario before reaching its end.

In this case, the system should violate the time constraint. The time constraint in Fig. 9 is modeled as follows (see also [HM02, HM03]): the solid box on the RailCab lifeline with the sand-clock symbol in the upper right corner is a clock reset that resets or, in this case, initializes the clock $c$ with 0. It is defined that such a clock reset is performed at the instant it is enabled, which means that no time passes between receiving `enterAllowed` and the clock reset. The blue, dashed hexagon following the `enterNext` message is a cold time condition that states that the clock $c$ should have a value greater than 5. Such a time condition is also evaluated immediately after it is enabled. That means that there is a cold violation when the `enterNext` message is received less than 5 seconds after the track section control sends the `enterAllowed` message to the RailCab. Thus, violating the cold time constraint this way, the system can avoid violating the use case specification.

But when is the admissible time to send the `enterAllowed` message? To determine this, we must know when the `enterNext` event will occur, which is implied by the time constraints in the environment assumptions. Figure 10
2.2 Two example use cases

shows the automaton from Fig. 8 that additionally contains the timing assumptions of the above use case.

Here, in the manner of Timed Automata [AD90, Alu99], time constraints can be specified by initializing or resetting clocks as well as by guard conditions on the transitions. The automaton in Fig. 10 says that \textit{lastBreak} happens at least 5 seconds after \textit{endOfTS} (our time unit shall be seconds here). \textit{noReturn} happens some time after \textit{lastBreak}, and then, \textit{enterNext} happens within 8 to 12 seconds after \textit{endOfTS}. We could also specify that \textit{lastBreak} and \textit{noReturn} shall not take too long, such that the automaton does not deadlock in the state \textit{noReturn}, but we assume here that the environment does not deadlock—events in the environment simply do not occur in any other way than restricted by the automaton.

With these requirements and environment assumptions, however, we run into problems. We assume that the \textit{lastBreak} can happen as soon as 5 seconds after \textit{endOfTS}. Therefore, the \textit{enterAllowed} reply of the track section control must be sent some time before that happens (see the MSD in Fig. 7). However, when it takes too long (longer than 10 seconds) to enter the next track section (\textit{enterNext}), the requirement of not to allow RailCabs to enter the next track section too early is violated. Figure 11 illustrates the time constraints on the environment events and system messages.

To resolve the inconsistency, we can modify the requirements in a number of ways: first (i), we could loosen the requirement such that the reply of the track section control need not be sent 5 seconds prior to entering the next track, but that this may be up to 7 seconds before entering. But (and this is a bit tricky) the 7 seconds only work if we can assume that \textit{lastBreak} occurs not only \textit{at least} 5 seconds after \textit{endOfTS}, but actually \textit{more} than 5 seconds.

Figure 9: MSD specifying that the reply of the track section control must not be sent too early.

Figure 10: The automaton representing the timed environment assumption of the “drive onto merging switch” use case.
Figure 11: An illustration of the time constraints formulated in the “drive onto merging switch” use case.

after endOfTS (> 5 instead of ≥ 5 seconds). Second (ii), if we cannot change the assumptions, we could loosen the requirements that the reply of the track section control need may be up to 8 seconds before entering. Third (iii), we could skip the requirement that the reply of the track section must be sent before the RailCab reaches the point of the last break (lastBreak). (Then, to be on the safe side, the RailCab would have to break when the track section control does not reply.) Instead of changing the requirements the assumptions could be changed (iv): if we knew that the point of last break (lastBreak) came later, with less time span remaining until entering the next track section (enterNext), the requirements could be fulfilled. That, however, is something that depends on the parameters of the RailCab’s breaking system. Thus, in a real development process, we would need to coordinate with the mechanical engineers whether this can be guaranteed.

2.3 Can Smart Play-out help?

Smart play-out [HKMP02b] is a technique that can help the engineer find inconsistencies in an LSC specification. Smart play-out helps in such a way that it can, to a certain degree, avoid running into avoidable violations of hot fragments which naive play-out can at times not avoid. That means that, when a hot violation occurs during smart play-out, the engineer has more reason to suspect an actual inconsistency than when a hot violation occurs during naive play-out.

So, since the power of smart play-out lies in avoiding avoidable hot violations, the question is whether smart play-out can find an admissible execution of a repaired version of the above example specification. In the following I briefly argue that smart play-out extended with time [HKP04] cannot avoid an avoidable violation in this case. Therefore, violating runs of smart play-out do not necessarily imply that the specification is inconsistent.

The principle of smart play-out is that it can find admissible super-steps. A super-step is a sequence of system steps that shall be performed in reaction to an environment event, until the system waits for the next environment event to occur. To find such admissible super-steps, smart play-out utilizes model
checking: upon every environment event the LSC specification the current state of the objects in the system, and the currently activated LSCs are encoded as an input for a model checker. The model checker then is given the task to prove the property that in this model always at least one of the universal charts remains active. That may be for example because there is a hot message that the system must send, but some other LSC is in a state that does not allow to do that—which conforms to the notion of a hot violation. If, however, the model checker fails to verify this property, it returns a counter-example of a sequence of system steps that at some point renders all LSCs inactive—and this is exactly the desired, admissible super-step.

Let us assume that we have fixed the above specification (the MSDs of Fig. 6, 7, and 9) such that now the `enterAllowed` reply of the track section control can be sent up to 7 seconds before the RailCab is entering the next track section `enterNext` while, second, assuming that the event `lastBreak` occurs more than 5 seconds after `endOfTS`, not at least 5 seconds after (> 5 instead of ≥ 5 seconds), see alternative (i) above. Now we want to use smart play-out to simulate the system. The admissible super-step that smart play-out will return on the `endOfTS` event consists of sending the messages `requestEnter`, `enterAllowed` and setting the clock c to 0. Then, the active MSD of Fig. 6 becomes inactive (terminates). Also the MSDs in Fig. 7 and 9 become inactive, because they wait for the environment messages `lastBreak` and `enterNext` to take place.

In the scope of this super-step, there is no time constraint that would give any reason to delay the sending of any of the system messages, and, therefore, smart play-out would decide to send these messages without any delay. The problem occurs in the next super step, in reaction to the `enterNext` event: having sent the `enterAllowed` message at the time 0, the value of clock c will be at least 8 at the time of receiving `enterNext`. Therefore, a hot violation will occur and will not be avoided by smart play-out. This is an example of the limitations of smart play-out, namely that it cannot sufficiently look into the future to anticipate violation that may occur in subsequent super-steps. For this reason, more powerful means are required to find admissible system reactions where they exist, or, on the other hand, to show clearly that there are none.

In the play-out scenario described here, I assume that active MSDs can become inactive while they wait for an environment event to occur, upon which they are activated again. This is however not supported in the original definition of universal LSCs [HM03]. Universal LSCs consists of a `prechart` and a `main chart`. The prechart acts as a trigger: when a sequence of environment events or system actions fulfills the prechart, the behavior described in the main chart has to take place. Therefore, the life-cycle of a universal LSC is as follows: At first a universal chart is inactive. Then, it can be `pre-active` when a message that is the first in the prechart occurs. When the prechart is fulfilled, the chart becomes active, and when the main-chart is fulfilled, or a violation of a cold fragment takes place, the chart becomes inactive again.

MSDs, however, are more flexible than LSCs in the sense that they can have cold messages at arbitrary locations throughout the chart that are only
monitored and not actively executed. This is for example relevant for modeling environment messages that are expected to occur after certain system steps, as shown in Fig. 7 (a more detailed introduction to MSDs is given in Sect. 4). These cold, monitored messages “serve prechart-like purposes” [HM08]. This means that the diagram, when active, can be come pre-active again, then active, and so on.

Smart play-out could probably be extended to look further into the future, namely beyond the prechart-like cold messages, and try to find subsequent super-steps that eventually render all charts inactive. We then had to assume that certain environment events will eventually occur and will thus help the system to continue in the execution of a chart. But, to the best of my knowledge, this has not been elaborated thus far.

3 Sketch of the approach

In the following, I give a short overview of the synthesis approach in this paper. The approach primarily consists of mapping the MSD specification to a system of Timed Game Automata. In combination with a winning condition, Uppaal Tiga can synthesize a winning strategy for the system. If no winning strategy can be found, the MSD specification is inconsistent. Then Uppaal Tiga produces a counter-strategy that shows how the environment can violate the MSD specification. The approach is not complete, which means that the synthesis may find a winning strategy which is not satisfying the MSD specification. Details are discussed in Sect. 6.

As described in the previous section, an MSD specification consists of an object system and a number of MSDs that describe the behavior of the object system. The objects in the object system are either environment objects or system objects. Note that the object system is static, which means that no additional objects are created and no existing objects are destroyed. The lifelines of the MSDs represent exactly one object in the object system, therefore the MSDs are called concrete MSDs. MSDs as well as LSCs further have the concept of symbolic lifelines, where a lifelines can represent any object of a certain kind. But, this concept is not considered here.

The different parts of the MSD specification, the environment objects, the system objects, and the MSDs are displayed at the top of Fig. 12. The bottom shows the environment automaton, the system automaton, and the MSD automata which are created in Uppaal Tiga for these parts of the specification. The automata are created as follows.

One automaton, the environment automaton, represents the possible environment behavior. It can “produce” the events occurring in the environment. It does so by assigning an integer constant representing an event to a global variable event. Then, it emits over a broadcast channel events. Through this broadcast channel, the MSD automata are synchronized when events are produced, which is explained shortly. This environment automaton can either produce the environment events non-deterministically or, when assumptions for the environment are made, this automaton can be modeled such that it only
Figure 12: Sketch of the principle of encoding an MSD specification as a system of Timed Game Automata

produces events in a restricted way. The environment automaton here shall represent the environment assumption as shown in Fig. 8 and 10. In the scope of this paper, it is required that the environment assumptions are modeled manually in UPPAAL Tiga, following the implementation principle presented in Sect. 6. It is desirable to also specify the environment assumptions in a more intuitive way and generate this automaton for example from an UML statechart model, desirably time-annotated statecharts like real-time statecharts [GB03]. In Sect. 6.4 I discuss some ideas of also using MSDs for that purpose.

The edges in the environment automaton (the arrows are called edges in UPPAAL, not transitions) are represented by dashed arrows, which means that they are controlled by “the opponent”, which is the environment in this case.

The second automaton created for an MSD specification is the system automaton. This automaton can produce system events similar to the environment automaton. The system automaton can choose non-deterministically among the events which are enabled in any of the MSDs. Enabled events are such events where there is an MSD in which the current cut is immediately prior to an occurrence of the event. To be more specific, the system automaton may choose to produce only such enabled events that are marked to be actively executed by the system. Such events are called active, see Sect. 4.

Third, MSD automata are created for each MSD in the specification. Integer variables for each lifeline store the current cut of the MSD and edges in an MSD automaton are created which correspond to events in the MSD. Such an edge is enabled when the corresponding event in the MSD is enabled in the current cut. When the environment or system automaton produces a suitable event, the edges in the MSD automata are synchronized with the edges in the system or environment automaton via the events broadcast channel. The edges, when they are taken, progress the cut by updating the lifeline variables accordingly.
Furthermore, an environment or system event may lead to a cold or hot violation of an MSD. A cold violation results in setting the lifeline variables to 0, and a hot violation leads to a sink state, setting the variable \textit{hotViolation} to true. In this synthesis approach, the winning condition is to always avoid hot violations, \textit{A[] not hotViolation}. Therefore, the system will always try to avoid these violations.

To win in a timed setting, the system will always try to progress the active MSDs before the next environment event occurs, before the environment events produce a hot violation in the MSDs. In an untimed setting, the environment cannot produce events as long as the system is active. To prohibit the system in avoiding hot violations by doing nothing, the system is always in a \textit{committed} state and thus must produce events as long as it is active. As mentioned in the introduction, this synthesis approach is not complete, since a winning strategy could also be that the system takes an infinite cycle where no hot violations occur, but always at least one MSD remains active.

This encoding here is only a sketch and there are a number of further details to be considered. The mapping from the MSD specification to the system of TGA is explained in more detail in Sect. 6. Before that, I give a more detailed introduction to MSDs and the language of \textit{Uppaal Tiga}.

4 Modal Sequence Diagrams

In this section, I give an introduction to Modal Sequence Diagrams. I describe the structure of an MSD specification and its dynamic semantics. Also, I briefly explain the play-out algorithm and the difference in the play-out scheme of timed and untimed MSD specifications. At last, in Sect. 4.5 I explain the abstract syntax of an MSD specification modeled in UML2.

4.1 Introduction to MSDs

MSDs allow us to specify that, during the interaction of objects of a system and its environment, certain sequences of events may, must, or must not occur. To express this, there are \textit{existential} MSDs to describe interaction sequences that must be possible to occur and \textit{universal} MSDs to express restrictions over all possible runs of the system. In universal charts, as we have seen in the previous examples, we further distinguish between \textit{hot} and \textit{cold} messages for representing events that must eventually occur or events that may occur. Using these modalities, MSDs are a convenient formalism for specifying the set of \textit{admissible runs} of a system. By adding a few language constructs, it is possible to operationalize a set of universal MSDs for example via the play-out algorithm. Because the operational interpretation of MSDs is more convenient for explaining and understanding the basic principles of MSDs, the operational semantics is introduced first. I return to the more general interpretation of MSDs in Sect. 4.2.

In LSCs, universal charts are divided into two sections, called the \textit{prechart} and the \textit{main chart}. The play-out algorithm interprets these parts of the chart such that if a sequence of events occurs that satisfy the prechart, play-out
will try to execute the steps described in the main chart. In MSDs, there is no explicit prechart or main chart, but instead messages can be marked as monitored or executed. The monitored messages fulfill the purpose of the prechart [HM08]. Corresponding to the main chart, the monitored messages are followed by messages marked as executed [MH06, HKM07] and the play-out algorithm will try to execute them. See Fig. 13 for a comparison of an LSC and an MSD. MSDs are more flexible than LSCs because monitored messages may occur also after sequences of executed messages.

Figure 13: A Live Sequence Chart (left) and the corresponding Modal Sequence Diagram (right).

For simplicity, the steps described by a sequence diagram, like sending messages or evaluating conditions are called events, even if the play-out algorithm is actively executing them. Events are either visible events or hidden events: visible events are events that take place in the environment or they are reactions of the system, represented in the MSD by messages. Hidden events are such events which are only important for processing the diagrams, like the evaluation of conditions or the assignment of diagram variables.

The first message in the sequence diagram is called the minimal event. On the occurrence of an event in the system which corresponds to this minimal event, an active copy or instance of the diagram is created. On the occurrence of further events in the system or its environment that correspond to the subsequent messages, the active copy of the diagram is progressed. To capture this progress, the passed locations on each lifeline are marked. Locations are the points on the lifeline where messages are sent or received, conditions evaluated, and so forth. The set of already passed locations in the active copy of the diagram is called the current cut of the diagram. Note that there may not only be one first message in the diagram, but there may be multiple messages that have no preceding events. However, I define that only one of the messages can be the minimal event. Furthermore, just as the minimal event in a universal LSC is always in the prechart, the minimal event in an MSD is always cold and monitored.

An LSC is said to be pre-active as long as the cut has not passed all locations in the prechart. Upon reaching the end of the prechart, and therefore progressing to the main chart, the chart is said to be active. Upon reaching the end of the main chart, the chart is discarded. The same principle applies for MSDs, except that the active diagram may become pre-active again if on all lifelines the locations immediately following the current cut are monitored messages.
When a diagram is (pre-)active, there are one or more events that when they occur progress the current cut. These possible next events are called enabled events. If the enabled events are in the main chart or, in MSDs, the events are marked as executed, the possible next steps are called the active events. There may be several concurrently enabled/active events in a (pre-)active diagram, because there is not necessarily a linear sequence of events in a diagram, but the diagram describes a partial order among the events. Furthermore, multiple diagrams can be active or pre-active at the same time; therefore there may be a multitude of enabled/active events at a time. Once there are diagrams active, the play-out algorithm is forced to execute the active events as long as at least one diagram remains active.

In an untimed setting, it is assumed that the system is fast enough to always react to environment events before the next environment event occurs. That means, on the occurrence of an environment event, the play-out algorithm executes active events as long as all active diagrams are discarded or pre-active. Only then, the next environment event occurs. This sequence of steps that the system can take until it waits for the next environment event is called a super step. In the following, the assumption that the environment is inactive while the system is active is called the super step assumption. In Sect. 4.4 I explain that this assumption is not practical when we include time in our specification.

Multiple active diagrams may have the same events enabled at a time. Then, those diagrams progress simultaneously on the occurrence of this event (e.g. when the event is also active in one of the diagrams and the play-out algorithm executes it). When an event occurs that does not correspond to an enabled event in an active diagram, there can be two cases: first, if the event does not occur anywhere in the diagram, it does not affect the active diagram and it remains in the current cut. However, if this event occurs somewhere in the diagram, but it is not enabled in the current cut, there is a violation of the diagram.

If a violation of an active diagrams occurs while there is at least one hot enabled event in the current cut, there is a hot violation of the diagram. A hot violation constitutes forbidden behavior, which must not happen under any circumstances. If a violation occurs in an active diagram while all the enabled events in the current cut are cold events, there is a cold violation. Contrary to hot violations, cold violations may occur since the events violated were not strictly required to occur. However, a cold violation is a violation of an active diagram, which is then discarded. The cut of a diagram is called a hot cut when at least one of the enabled events in the active diagrams is hot. When all the enabled events in the active diagram are cold, this is a cold cut.

Distinguishing the prechart and the mainchart or monitored and executed messages are not only important for the play-out algorithm. The intuition of a cold executed message is “this should happen” whereas the intuition of a cold monitored message is “this may happen”. The event is in both cases not strictly required to occur, but for a software engineer who is going to implement the behavior, it means two different things: A cold executed messages means “try to find an implementation where this happens, but its okay when it doesn’t happen if there are reasons against it”; a cold monitored message means “your
implementation should do that only if it says so somewhere else”. In other words, by cold, executed messages, a stakeholder can express how he wishes the system to behave. Accordingly, the play-out algorithm will try to execute as much of the main chart/executed messages as possible. The play-out algorithm is even implemented to try to avoid cold violations where it can in order to do more of the things that should happen \cite{HM03}. On the other hand, a stakeholder does not wish the system to spontaneously do all the things that are possible happen. For example, take an MSD where the track section control must order all registered RailCabs to break when one of the RailCabs sends a cold monitored message \texttt{hazardOccurred}. Of course, \texttt{hazardOccurred} should not happen spontaneously, but only if there are other requirements that require \texttt{hazardOccurred} to be sent. For this reason, the notion of monitored/executed messages is also included in the synthesis approach. Further details are discussed in Sect. 6.

Note that hot messages strictly require the event to occur and from the perspective of the play-out algorithm there is little reason to mark them as monitored. But, one could think of other use cases where, for example, part of the system is already implemented and one would like use the play-out algorithm to just monitor the behavior of the implementation.

Note further that the more general semantics of MSD, which is explained in the following, does not require to distinguish monitored and executed events at all. In the more general sense, an MSD specification defines which runs of the system are allowed and which runs are not allowed.

### 4.2 The language of an MSD specification

Harel and Maoz define the semantics of MSDs by a mapping from universal and existential MSDs to a kind of Büchi automata, called \textit{Alternating Weak Word Automata} (AWW) \cite{HM08}. A system is defined to satisfy an MSD specification if each infinite run of the system is accepted by each automaton that corresponds to a universal MSD. Furthermore, it is required that each automaton which represents an existential MSD accepts at least one possible infinite run of the system.

The set of \textit{admissible runs} is therefore the language which is formed by the \textit{intersection} of languages described by the MSDs. Further, a system satisfies an MSD specification if the set of its possible runs is a subset of the admissible runs.

The AWW corresponding to the universal MSD \texttt{RequestEnterAtEndOf-TrackSection} (see Fig. 9) is show in Fig. 14. The AWW construction described by Harel and Maoz, but also previous mappings from LSCs to automata \cite{HK99,BH02,KW01} follow the same principle: an automaton is created with states that represent legal (reachable) cuts in the diagram and transitions that correspond to events in the system. In an AWW for an universal chart, there are basically three kinds of states and four kinds of transitions. The first kind of states are the cut-states, which correspond to legal cuts in the MSD. Additionally, there are two sink states, one state \texttt{r} for rejected runs and another \texttt{a} for accepted runs. The first kind of transition are such transitions that can
lead from a state representing one legal cut to another state which represents a subsequent legal cut. In Fig. 14 this is for example the transition labeled by the event requestEnter from state $s_0$ to state $s_1$. The second and third kind of transition are such that represent a hot or cold violation and lead to the sink state $r$ or $a$, respectively. In Fig. 14 for example, the transition labeled by the set of events $M \setminus \text{requestEnter}$ leads to the state $r$. $M$ is the set of events occurring in the MSD. Thus, upon any of the events occurring in the MSD except the expected event requestEnter, the transition is taken to the rejecting state, which corresponds to the notion of a hot violation as explained above. Transitions that represent cold violations have the same label, but lead to the state $a$ instead.

The fourth transition kind is the state’s self-transitions, which correspond to events which are ignored by the MSD. The transition labeled $\Sigma \setminus M$ on state $s_1$ is an example of such a transition. $\Sigma$ is the set of all the events in the system, and therefore $\Sigma \setminus M$ are the events ignored by the diagram. (Note that $\Sigma$ is not only the set of events occurring in the MSD-specification.)

On the occurrence of the minimal event (requestEnter in this case), the AWW takes a transition into both $s_0$ and $s_1$. The $\land$ symbol expresses that both transitions form a universal transition. The runs of alternating automata form a tree and we can imagine that, on the occurrence of the minimal event, two copies of the automation are created: one copy that remains in the initial state and one copy that progresses to the state $s_1$. The latter copy corresponds to the active copy of the MSD and it eventually either ends up in the state $r$ or state $a$. On a cold violation or when reaching the end of the diagram, this copy ends up in state $a$ where it loops infinitely often. Since $a$ is an accepting state, this copy accepts any remaining (infinite) sequence of events. Instead, upon a hot violation, the copy ends up in state $r$, which is not an accepting state. Since there are no outgoing transitions, the copy remains in this state forever and thus does not accept any remaining runs of the system.

![Figure 14: The AWW corresponding to the universal MSD RequestEnterAtEndOfTrackSection in Fig. 6.](image-url)
4.2 The language of an MSD specification

Note that the first copy of the automaton loops in the accepting initial state, and further copies of the of the automaton are created upon the occurrence of the minimal event. This way the AWW especially capture the fact that multiple active copies of the same diagram, referring to the same objects, can be created. This may happen when certain sequences of events that are accepted somewhere along the diagram are also accepted in the beginning of a diagram. Take for example an MSD with a sequence of events endOfTS, requestEnter, enterAllowed, endOfTS, requestEnter: on the occurrence of endOfTS, requestEnter, enterAllowed, endOfTS there will be two active copies of the same MSD.

The synthesis approach in this paper, however, assumes that only one active copy of a diagram can be created at a time. Such restrictions were also made previously, for example by Klose et al. [KTWW06]. This restricted interpretation, that allows only one active copy on an MSD called the iterative interpretation. The interpretation that allows multiple active copies is called the invariant interpretation. The iterative interpretation may not be as elegant as the invariant interpretation given by the mapping to AWW. But, for many examples, and also the example in this paper, there is no difference between the iterative and invariant interpretation. Furthermore, the iterative interpretation makes the semantic mapping presented in this paper much easier. Figure 15 shows a regular Büchi automaton which reflects this iterative interpretation of the universal MSD in Fig. 6.

Because an MSD does not necessarily describe a strict sequence of events, but defines a partial order among the events, the corresponding automata that represent the different possible runs of the diagram typically have a characteristic diamond-shaped form. In larger MSDs, especially when the MSDs have more than three lifelines, or when asynchronous communication is described, the number of cut-states easily becomes larger than the number of all locations of all lifelines.
When sticking to the more simple iterative interpretation of the MSDs, another way to illustrate the possible runs through an MSD is by using a Petri net. A Petri net represents the possible cuts in a way that is quite isomorphic to the structure of an MSD: basically each location of each lifeline is represented by a place and each event in the diagram is represented by a transition. Each reachable configuration of the Petri net represents a legal cut of the MSD. Figure 16 shows a Petri net representation of the MSD in Fig. 4. The inactive diagram is represented by the initial marking of the Petri net at place $s_0$, which corresponds to the initial state $s_0$ of the automata above in Fig. 14 and 15. On an occurrence of the minimal event requestEnter the diagram is activated and reaches the cut $(l_{0,1}, l_{1,1}, l_{2,0})$. Accordingly, the transition is fired in the Petri net, which results in the configuration $[s_{10}, s_{11}, s_{20}]$.

In contrast to the automata-based trace language definition of an MSD, the Petri net representation requires an extra transition which marks the termination of the MSD. In Fig. 16, this is the transition marked by the event requestEnterAtEndOfTrackSectionEND. This transition collects all tokens upon the occurrence of the last enabled event, which results in having the marking $[s_0]$ again. This event is however not an event occurring in the system, but used to manage the life cycle of the MSD. These events are therefore also hidden events. In the Petri net, the hidden events are represented by a transition with a dashed border.

![Petri net diagram](image)

Figure 16: The Petri net corresponding to the universal MSD RequestEnterAtEndOfTrackSection in Fig. 6

The disadvantage is that this Petri net does not capture the exact semantics of hot or cold violations as the above automata do by the transitions to the accepting/rejecting states. But anyway, this Petri net representation shall just illustrate the (non-violating) runs of an MSD. In our mapping from an MSD to the system of TGA (see Sect. 6) the idea is that variables represent the current locations resp. markings and transitions increase or reset these variables. This principle is inspired by the encoding principle used in smart
Assignments and conditions

A whole range of additional constructs were defined for LSCs and MSDs. This paper, however, is not going to consider all of these constructs. In addition to the synchronous messages regarded so far, this section explains conditions and assignments as they are used for example in the MSD shown in Fig. 9. In that example, a special kind of assignment and condition occur which deal with time; the time model used in this paper and the extended semantics of time conditions and clock resets (assignments of clock variables) is explained in the subsequent section (Sect. 4.4).

Assignments are events in an MSD that assign a value to a diagram variable by expressions of the kind \(<\text{diagramVar}> = <\text{expr}>\). Diagram variables are only visible in the scope of an active MSD. Assignments may span one or multiple lifelines of the MSD and they are enabled when the cut is immediately prior to the assignment on all lifelines that it covers. Assignments are executed as soon as they are enabled. When a variable is assigned, it can be used in condition expressions succeeding the assignment. Assignments have no temperature and cannot violate the diagram.

Conditions are events in an MSD with an expression that evaluates to a Boolean value. Like assignments, conditions span one or multiple lifelines in an MSD and they are either hot or cold. A cold condition is enabled as soon as the cut is immediately prior to the condition on all lifelines that it covers, and then it is immediately evaluated. If the evaluation of the cold condition expression results true, the cut progresses. If the result is false, it is a cold violation. A hot condition is enabled only when the cut is immediately prior to the condition on all lifelines that it covers and when its expression evaluates to true. That means that the diagram cannot progress beyond the condition as long as expression evaluates to false, and it is a hot violation if that will never be the case. One may require another or more versatile interpretation of conditions like “evaluate immediately before the next event”, or “must/may be valid between events”. Such interpretations are, however, not considered here. The expressions in conditions can be formed by referring to attributes of the objects that correspond to the lifelines that a condition covers or by using diagram variables, which must be assigned before the evaluation of a condition.

Conditions and assignments are hidden events since their evaluation resp. execution is not directly visible in the system. This is represented by the dashed transitions in Fig. 17. Assignments cannot assign values to object attributes directly. Otherwise their effect would be visible in the system. Harel and Marelly instead suggest a concept where messages may affect an attribute of an object represented by the receiving lifeline. These concepts are, however, beyond the scope of this paper.
4.4 Time

The conditions and assignments in the above example are actually of a certain kind, namely *time conditions* and *clock resets*. Clock resets and time conditions are visualized like assignments and conditions with an additional sand-clock icon in the upper-right corner (see Fig. 9). Clock resets are assignments which reset *clock variables*. Clock variables are a special kind of diagram variables. In this approach, a dense time model is used where clocks are real-valued variables that synchronously and monotonically increase in value over time.

![Figure 17: The Petri net corresponding to the universal MSD ReplyOfSwitchControlNotTooEarly in Fig. 9.](image)

The notion of time was introduced to LSCs by Klose and Wittke [KW01] by mapping LSCs with discrete time constraints to Timed Büchi Automata. Harel and Marely similarly define discrete time constraints for LSCs and describe how these constraints are interpreted by the play-out algorithm [HM02, HM03]. Plock et al. consider LSCs with a dense-time model and describe a mapping from timed LSCs to a timed automata [PGZ05]. Common in these approaches is that only one clock variable *Time* carries the current time. Its value can be assigned to real-valued *time-stamp variables*, which can be used in condition expressions later on. In principle, having one clock-variable *Time* and time-stamp variables or multiple synchronously increasing clock variables makes no difference in expressive power. In order to make the formal analysis of timed automata feasible [AD94], the expressions involving clocks are restricted to the form of \( x_1 \triangleleft c \) or \( x_1 - x_2 \triangleleft c \), where \( x_1, x_2 \) are clock variables, \( c \) is a natural number, and \( \triangleleft \in \{<, \leq, >, \geq, =\} \). (Plock et al. [PGZ05] similarly restrict the expressions to the form *Time* \( \triangleleft tsvar + c \),
where \( tsvar \) is a time-stamp variable. This is an equivalent restriction of the language.)

The semantics of cold conditions, “evaluate as soon as enabled”, applies for cold time conditions as well. Accordingly, hot time conditions are only enabled when their expression evaluates to true. If a maximal time delay evaluates to false, it will never evaluate to true in the future, which is therefore a hot violation.

As defined by Harel and Marelly [HM02], it is assumed that no time passes for any events in the system (called the synchrony hypothesis). Time passes only between events, when the system waits for a hot time condition to become true or when it delays the execution of certain steps. When the system is waiting, time passes and environment events may occur.

In the extension of the play-out algorithm with time [HM02], Harel and Marelly define that active LSCs only wait for a hot, lower-bound time conditions to become true (minimal delays). This means that a super-step terminates with all active MSDs being discarded or with only hot time conditions waiting to become true. This sounds like a reasonable adaption of the super-step assumption for a timed setting. But, in fact, this assumption is too restrictive and leaves no room for the system to delay certain steps when there is no explicit minimal delay to make the system wait. The example shown in Sect. 2.2 is such an example where the rule “do everything as soon as possible” would not allow the system to delay the reply of the track section control such that it is not sent too early.

Therefore, the synthesis approach presented here is more liberal and allows the system to delay its steps. But then, the environment events may occur in many more places than in an untimed setting. Therefore, in most practical cases, the synthesis approach presented here requires to specify explicit environment assumptions such that the MSD specification is not easily violated.

### 4.5 Modal Sequence Diagrams in UML

This section illustrates how an MSD specification is represented using UML2.2 and a profile. The MSD profile used here is a slightly modified variant of the profile proposed by Harel and Maoz [MH06, HM08]. In the following, I first present the stereotypes used in the MSD profile. Second, I illustrate the abstract syntax of an MSD specification in UML by object diagrams. This shall help to understand the structural mapping from MSDs to the system of TGAs in Sect. 6 but especially to understand the mapping definition in Appendix B and C. For details on UML profiles and more information about the UML metaclasses, I refer to the UML2.2 specification [UML09].

The diagram in Fig. 18 shows the stereotypes and data types which extend UML for modeling MSD specifications. First, there is the stereotype **MSD-Specification**, which extends the UML meta-class **Package**. By its attribute **specificationKind**, an MSD specification can be marked as a timed specification or an untimed specification.

Second, there is the stereotype **SpecificationPart**, which extends the UML meta-class **Property**. Specification parts model the environment and system
objects in the MSD specification (see Fig. 4). To distinguish environment and system objects, the SpecificationPart stereotype has the attribute partKind. This attribute is typed over the enumeration PartKind, which defines two literals “Environment” and “System”.

Third, the profile contains the stereotype ModalMessage for modeling hot and cold messages. The stereotype ModalMessage has an attribute temperature, typed over the enumeration TemperatureKind. This enumeration defines the two literals “cold” and “hot”. Each message in an MSD specification must be a modal message, which is expressed by the “required”-annotation on the stereotype extension arrow.

Fourth, the profile contains the stereotype Condition, which extends the metaclass StateInvariant, and, like the modal message, defines an attribute temperature. Conditions are state invariants with the condition stereotype applied, assignments are state invariants with no stereotype applied. To syntactically distinguish time conditions and assignments which specify clock resets, there are the stereotypes TimeCondition and ClockReset.

Figure 18: The UML-Profile for MSD-Specifications

The object diagrams in Fig. [19][20] and [21] illustrate the abstract syntax of an MSD specification modeled in UML and by using the above profile. The first two object diagrams show part of the MSD RequestEnterAtEndOfTrack-Section in Fig. 6 and its object system as shown in Fig. 4. The object diagram in Fig. [21] shows how the time condition of the MSD ReplyOfSwitch-ControlNotTooEarly in Fig. 9 is represented in the abstract syntax.

Figure [19] shows that the root object of an MSD specification is a package with the MSDSpecification stereotype applied. (For simplicity, the name of the applied stereotype is used as the type name of the object.) A MSD specification contains a collaboration, which in turn contains the specification...
parts which represent the environment and system objects of the MSD specification (see Fig. 4). The specification parts are properties of the collaboration with the stereotype `SpecificationPart` applied. As such, they define values for the `partKind` attribute. Furthermore, the specification parts are typed over the classes contained in the package. In addition to the specification parts, the collaboration contains interactions. The interaction shown here represents the MSD `RequestEnterAtEndOfTrackSection` shown in Fig. 6. The lifelines in the MSD interactions represent the environment, resp. system objects by referencing the specification parts contained in the collaboration.

![Diagram](image)

Figure 19: The abstract syntax of an MSD specification: packages, classes, collaborations, parts, interactions, and lifelines

Figure 20 shows further details of the abstract syntax; objects already occurring in Fig. 19 are grayed out. Besides the lifelines, an interaction contains modal messages (messages with the `ModalMessage` stereotype applied) and message occurrence specifications. The diagram shows how the first message in the MSD `RequestEnterAtEndOfTrackSection` is represented in the abstract syntax: a message connects the sending and receiving message occurrence specifications, which represent the sending and receiving locations on the sending and receiving lifelines. The diagram further shows that the message occurrence specifications are contained in the interaction via the ordered reference `fragments`. The position in this list is denoted by the numbers in the square brackets. This ordering reflects the order of events in a UML Sequence Diagram; the partial order of events in the MSD can be deduced from this ordering.
Each message occurrence specification references a send operation event. For synchronous messages, the sending and receiving message occurrence specifications refer to the same event. The event in turn refers to an operation defined by the type class of the property represented by the receiving lifeline. When messages of the same kind occur in different MSDs, or the same MSD, there are multiple send operation events which refer to the same operation, i.e. the operation encodes the type of the message.

![Abstract Syntax Diagram]

Figure 20: The abstract syntax of an MSD specification: messages, occurrence specifications, events, and operations

The abstract syntax representation of a condition, in this case the time condition of the MSD in Fig. 9, is shown in the object diagram of Fig. 21. As an extension to state invariants, a time condition references one or more lifelines that it covers. A state invariant further can have a constraint with an expression which carries the expression string. Just like message occurrence specifications, state invariants are contained in the interaction via the fragments list. Fig. 21 shows that the time condition of the MSD in Fig. 9 is at position 5.

5 Uppaal Tiga

Time is an important aspect for many kinds of systems. Especially in mechatronic systems, there are often time-critical requirements on physical processes
that need to be controlled by software, while the software is running on a platform that supplies a limited processing speed. Therefore, models for reactive systems that consider time were introduced in the past, for example Timed Automata [AD94].

Timed Automata are finite automata with a finite number of clock variables that can take a positive real value which synchronously, monotonically and continuously increases with time. The value of these clock variables can be reset on state transitions. Such a time model is called a continuous-time model or, synonymously, dense-time or real-time model. Continuous-time models have advantages over discrete-time models, for example is it not necessary to a priori decide on adequate time intervals for discretizing the model of a given system.

Alur et al. have shown that by restricting the expressions over clock values, Timed Automata can be described by a finite transition system [ACD93], which makes the automated checking of temporal properties feasible. A tool for modeling networks of Timed Automata and automatically checking their properties is the model checker Uppaal [BLL96]. A number of other analysis tools were implemented on the basis of the Uppaal model checker, such as Uppaal Tiga [CDF05, BCD06, BCD07a], a tool for finding strategies in timed games.

In the following, this section gives a short introduction to the Timed Automata (TA) in Uppaal and the Timed Game Automata (TGA) in Uppaal Tiga. For a more comprehensive introduction to Uppaal and Uppaal Tiga, see the Uppaal tutorial [BDL04] and the Uppaal Tiga manual [BCD07b]. A summary on the principles of the on-the-fly strategy synthesis algorithm implemented by Uppaal Tiga is given in Sect. 5.3. At last, Sect. 5.4 introduces a model of the abstract syntax of a network of TGA in Uppaal Tiga, which is used for the mapping from MSDs to TGA described in Sect. 6.

5.1 Timed Automata in Uppaal

In the Uppaal model checker, real-time systems can be modeled by a system of parallel Timed Automata. Figure 21 shows an example of a light switch modeled
as a system of two parallel TA. The example is taken from the UPPAAL tutorial [BDL04].

A Timed Automaton in UPPAAL is modeled in an template. For clarity, a template is also called automaton template in the following sections. A template defines the automaton and declares bounded integer, Boolean, and clock variables, channels, and functions that are used in the automaton. Additionally, there may be definitions of further data structures, but that is not covered here. A system in UPPAAL consists of a finite number of instances of these templates. Fig. 22 shows a system consisting of two template instances: Lamp1 is an instance of template Lamp and User1 is an instance of template User.

A template automaton consists of a set of locations, edges. Each automaton has one initial location, marked by the double-bordered circle. In a system, each instance of a template can be in one current location at a time. Locations and edges are deliberately not called states and transitions, since the state of a system is defined by the set of current locations of all template instances and the valuation of the integer and clock variables. A transition is a discrete change in the state of the system, which may be one edge or multiple, synchronized edges firing and possibly, in doing so, updating the values of integer variables and resetting clock values.

Each edge has a number of labels. Important here are the guard, update, and synchronization labels. Guard labels are side-effect free expressions over integers and clocks that evaluate to a boolean value. Clock guards are restricted to the form \( x_1 \succ c \) or \( x_1 - x_2 \preceq c \), where \( x_1, x_2 \) are clock variables, \( c \) is an integer expression, and \( \succ \) is a compare operator \( \in \{<, \leq, >, \geq, =\} \). Update expressions are a comma-separated list of assignments that values to variables. Clock variables may only be reset to integer values. An edge is enabled in a template instance, when the source location is the current location of the template instance and when its guard expression evaluates to true. Guards are evaluated before the updates are executed. Synchronization labels are used to synchronize edges via channels. The labels press! and press? in Fig. 22 are examples for such synchronization labels. In this case press is a binary channel, which means that, when the sending edge, labeled press!, fires, then a currently enabled receiving edge, labeled press?, must fire synchronously. If more than one receiving edge is enabled, one of these enabled edges is non-deterministically chosen for synchronization. If no receiving edge is enabled, the sending edge cannot fire. (The two latter cases will, however, never occur in the lamp switch example.)

Channels can further be broadcast channels, which means that a sending edge must synchronize with all enabled receiving edges. If there are multiple enabled receiving edges in a template instance (all going out of the current location), one is chosen non-deterministically. Also, in contrast to the binary channel, the sending edge may fire when no receiving edges are enabled.

In addition to the name label, locations may have an invariant and they may be urgent or committed locations. The location invariant is a side-effect free boolean expression like the guard expression of an edge. The only restriction is that lower bounds on clocks are not allowed. The system can never be in a location when the invariant of the location evaluates to false. Therefore, the
5.2 Timed Game Automata in Uppaal Tiga

system must leave the location before the invariant is false and it must not enter a location when the invariant evaluates to false. This further has implications on the synchronization: for example an enabled edge sending over a broadcast channel cannot fire when an enabled receiving edge leads to a location where the invariant evaluates to false.

If a location is urgent, this means that no time is allowed to pass while the template instance is in this location. If a location is committed, no time is allowed to pass, just as for urgent locations, but yet more strictly, it is required that the next transition leaves the committed location. If the system is in multiple committed locations at once, only transitions can be taken which leave committed locations.

The specification language used in Uppaal is a subset of CTL, which allows to specify constraints over states (expressions $\varphi$ over locations, variables and clocks) and safety properties (of the form $A\Box \varphi$ or $E\Box \varphi$), reachability properties (of the form $E\Diamond \varphi$), and liveness properties (of the form $A\Diamond \varphi$ or $\varphi \sim \psi$, which is equivalent to $A\Box (\varphi \Rightarrow A\Diamond \psi)$) over paths. The restriction is that path formulae (i.e. temporal operators) are not allowed to be nested. See the Uppaal tutorial [BDL04] for details.

5.2 Timed Game Automata in Uppaal Tiga

Uppaal Tiga [CDF+05, BCD+06, BCD+07a] is an extension of Uppaal for synthesizing winning strategies in timed, two-player games. Uppaal Tiga extends the Timed Automata of Uppaal to Timed Game Automata (TGA), which, in addition to the TA introduced above, allow to mark the edges as controllable or uncontrollable. Controllable edges can be fired by the system, uncontrollable edges can be fired by the environment. If controllable and uncontrollable transitions are enabled, the environment has priority over the system in firing the uncontrollable transitions.
Given a network of TGA and a winning condition, UPPAAL Tiga can find a strategy for the system that satisfies the winning conditions or it produces a counter-strategy, which shows how the environment can violate the winning condition. Winning conditions can be safety or reachability formulae, where safety formulae take the form $A \Box \varphi$ or $A[\varphi W \psi]$ and reachability formulae take the form $A \Diamond \varphi$ or $A[\varphi U \psi]$.

A strategy is a function that tells the system resp. environment which transition to take or, more precisely, when to take which transition or if to take any transition at all in a certain state. The strategy does not map all states potentially reachable in the TGA system, but only such which are reachable in a game controlled by the strategy, which means that there may be very small strategies in large systems.

Since UPPAAL Tiga introduces little further syntactic constructs, I omit an extensive example here, but refer to the UPPAAL Tiga manual [BCD+07b] for more details. The following section introduces the key ideas of the strategy synthesis algorithm.

5.3 On-the-fly synthesis of game strategies

The synthesis algorithm of UPPAAL Tiga is restricted to solving timed and untimed reachability and safety games. The strategies are basically determined by a backwards computation of winning states. The principle of the algorithm is briefly explained in the following. For details, see the original presentation of the algorithm by Cassez et al. [CDF+05].

In reachability games, a strategy for the system must be found such that it can always reach a state from a set of goal states. Safety games are the complement of these strategies: if there is no strategy for the environment to always reach a forbidden state, the system wins the safety game.

The algorithm can be explained best when only the reachability game is considered and first an untimed model is assumed. The algorithm for finding winning strategies in untimed reachability games was first described by Liu and Smolka [LS98]: starting from the initial state, the algorithm explores transitions and states until it finds a state which fulfills the winning condition. This is the first winning state. Then, it reevaluates the predecessor states of the winning state: if all states reachable by uncontrollable transitions are also winning states and at least one controllable transition leads to a winning state, then the predecessor state is also a winning state. To determine that, typically further exploration and reevaluation steps are required. Finally, if the reevaluation can include the initial state into the set of winning states, the system can always win the reachability game. If that is not the case, there is no strategy for the system to win.

In the timed case, the algorithm was enhanced by Cassez et al. such that a symbolic representation of states is used, i.e. not the winning status of single states is computed backward, but the winning status of symbolic sets of states. Furthermore, the backward computation of the winning states must also consider the predecessors of of states in different time regions.
For the untimed case, the algorithm for finding winning strategies is shown to have a complexity that is linear with respect to the size of state space (number of states plus number of transitions). In the timed case, the performance of the algorithm is not linear, but experimental results by the authors show that it “keeps up linearly” \cite{CDF+05} with the size of the state space by the help of a number of optimizations.

5.4 The Uppaal Tiga model

System models can be loaded into UPPAAL or UPPAAL Tiga via an XML-based file format. Other tools that want to invoke UPPAAL or UPPAAL Tiga have to produce valid input files. The mapping approach presented in this paper maps an UML-based MSD specification to such an input file. It does so via a two-step transformation that first transforms the UML model into an ECore representation of system of TGA and then transforms this model into a valid UPPAAL Tiga XML file (see Sect. 7). The ECore meta-model for UPPAAL Tiga is presented in Sect. A.

6 Mapping MSDs to Uppaal Tiga

This section describes the mapping from an MSD specification to a system of TGA in UPPAAL Tiga. First, Sect 6.1 describes how an untimed specification is mapped to TGA. Second, Sect. 6.2 presents changes and extensions to that mapping which are necessary in a timed setting.

6.1 Mapping untimed MSD specifications

As explained in Sect. 3, different parts of the MSD specification are each transformed into a TGA in UPPAAL Tiga. Section 6.1.1 first explains the two automaton templates that are created for each MSD specification: the environment automaton template, which can “produce” events that are possible to occur in the environment, and the system automaton template, which can “produce” active events in the system. Then Sect. 6.1.2 explains how the MSDs are each mapped to an MSD automaton template.

6.1.1 Environment and system automata for untimed specifications

Figure 23 shows the environment and system automaton templates corresponding to the object system of the example shown in Sect. 3 (see Fig. 1). The automata refer to a number of global declarations, which are shown in Listing 1. The basic idea is that these automata can choose which events occur in the environment or the system. The MSD automata (see Sect. 6.1.2) synchronize with these automata via the events channel. First, let us assume that no environment assumption restricts the environment behavior.

The environment automaton consists of an initial location environmentInitial and a committed location produceEvent. For each event that can occur in the environment, an edge leads from the initial location to produceEvent. An update
expression on each edge sets the global integer variable \texttt{event} to a value which corresponds to a possible environment event. As shown in Listing 1, \texttt{event} is an integer variable and integer constants with different values are declared which each correspond to a visible event in the system.

From the committed location \texttt{produceEvent}, an edge leads back to the initial location, which emits on the broadcast channel \texttt{events}. This edge synchronizes with all MSD automata, which leads to the activation of diagrams, the progress of cuts, violations, and so on (see Sect. 6.1.2 on how this encoded). Since the location is committed, the edge back to the initial location must be taken immediately, before any other edge from an uncommitted location is taken. This two-step approach, to first update the \texttt{event} variable and to then synchronize the automata, is necessary because in Uppaal, the guard expressions are evaluated before the update expressions and in the MSD automata, edges that synchronize via the \texttt{events} channel have guards to determine which steps to carry out accordingly in the MSDs. To take the right edge in the MSDs, the information on what step to carry out must be available beforehand in the form of the updated \texttt{event} variable.

The committed location of the environment automaton further has an invariant \texttt{not isSystemActive()}. This is a call to a side-effect free Boolean function which determines whether any hidden event or executed system event is enabled in any of the MSDs. That means that the environment cannot produce any events as long as the system is active, which corresponds to the assumptions made in an untimed MSD setting (see Sect. 4). The implementation of that function is explained in Sect. 6.1.2.

When the environment automaton emits over the \texttt{events} channel, the system automaton moves from the initial location \texttt{systemInactive} to the committed location \texttt{systemActive}. From this location, the system automaton can choose to produce certain events similar to the environment automaton. Further it can trigger the handling of hidden events, or it can move back to the initial location when the system it not active anymore. However, because the location is committed, the system automaton must always choose an edge leading from that location. This way, the system is forced to always take steps until the system is inactive again and moves to the location \texttt{systemInactive}. The system automaton can produce system events only when the corresponding event is active in any of the MSDs, which is calculated by the Boolean function \texttt{active(int event)}. (Enabled messages marked as executed are called \textit{active events}, see Sect. 4).

Further, the system can only produce system events when there are no enabled hidden events in any of the MSDs. This is calculated by another Boolean function \texttt{isHiddenEventEnabled()}, which is called from the guard expression of the edges leading to the second committed location \texttt{produceEvent}.

If there are hidden events enabled, the system automaton can take an edge that synchronizes with one of the MSD automata via the channel \texttt{hiddenEvent} in order to progress enabled hidden events in the respective MSDs. A binary channel is used here, since hidden events are only visible in the scope of a single MSD and do not have to be synchronized over multiple active diagrams.
6.1 Mapping untimed MSD specifications

![Diagram of the TGA for the system and environment objects](image)

Figure 23: The TGA for the system and environment objects
The declarations in Listing 1 furthermore show the Boolean variable hotViol-
ation. This variable is not modified by the environment or system automaton, but only by the MSD automata when hot violations occur. How the MSDs are mapped to the MSD automata is explained in the following section.

6.1.2 Mapping the MSDs to TGA

The MSD automaton for the MSD RequestEnterAtEndOfTrackSection is shown in Fig. 24. Listing 3 shows functions declared locally for that automaton template and Listing 4 shows additional details in the global declarations.

The environment automaton consists of an initial location and a committed sink location which represents the occurrence of a hot violation in the MSD. Except the one edge that leads to the sink location, all edges are loops on the initial location. These edges are of certain kinds that are explained in the following.

The basic idea is that this automaton captures the behavior of the Petri net shown in Fig. 16 and complements it with the hot and cold violation handling that is missing in the Petri net. The configuration of the Petri net, respectively the cut of the MSD is represented by integer variables of the form $\langle MSD-name \rangle_{\langle object-name \rangle}$. These variables are called lifeline variables. For example, the configuration (0,0,0) of the lifeline variables for the the MSD RequestEnterAtEndOfTrackSection corresponds to the configuration [s0] of the Petri net in Fig. 16; the configuration (1,1,1) corresponds to the configuration [s01,s11,s20] of the Petri net, and so on. The lifeline variables do not always correspond exactly to the locations on the lifelines (this is only true for for the lifelines involved in the minimal event). However, a configuration of the lifeline variables is also called the cut in the following. Listing 2 shows the globally declared lifeline variables for the MSD RequestEnterAtEndOfTrackSection.

Listing 2: The globally declared lifeline variables

```plaintext
int RequestEnterAtEndOfTrackSection_env = 0;
int RequestEnterAtEndOfTrackSection_rc = 0;
int RequestEnterAtEndOfTrackSection_next = 0;
```
The first kind of edge is the one that corresponds to the minimal event of the MSD. This edge is enabled in the initial cut, which means that all lifelines are at their initial position, in (0,0,0) in this case. Additionally, the edge is enabled when the current event produced by the environment or system automaton is the minimal event. The update expression increased all the lifeline variables to 1.

Second, the MSD automata have edges for each non-minimal synchronous message that occurs in the MSD. The guard of these edges also checks that the current produced event is the one of sending this message and whether the lifeline variables are in a configuration which enables the sending and receiving of the message in the MSD. Instead of determining that directly in the guard expression, this is implemented in the Boolean function \texttt{RequestEnterAtEndOfTrackSection\_enabled(int ev)} which is created locally for each MSD template (see Listing 3). The update expression increments the lifeline variables of the sending and receiving lifeline.

The third kind of edge shown in Fig. 24 is for handling the termination of the MSD, which is a hidden event of the diagram. The edge is enabled when the lifeline configuration has reached its maximal value, (1,3,3) in this case, and it updates the lifeline variables to the initial configuration.

Fourth, two edges are created for each MSD which handle cold violations. One of these two edges is in particular responsible for handling a cold violation that is caused by the occurrence of the diagram’s minimal event. At the occurrence of a cold violation, the active copy of the diagram is discarded, which is expressed here by resetting the lifeline variables to their initial configuration.

When the cold violation is caused by the minimal event, the active copy of the diagram is discarded, but a new active copy is created at the same time. This is expressed here by resetting all the lifeline variables of the violated MSD to 1. (Remember that this approach only considers the iterative interpretation of the MSDs, see the discussion in Sect. 4.2) This behavior is realized by these two edges as follows.

The edge for the general-case cold violation is enabled if the current event is an event in the MSD, but it is not enabled. The former is determined by the Boolean function \texttt{eventInMSD(int event)} (see Listing 3). The latter is determined by the help of the afore-mentioned Boolean function \texttt{enabled(int ev)}. Further, it is important that the current event is not the minimal event of the MSD and that no hot events are enabled in the current cut, because this would mean a hot violation. Whether or not a hot event is enabled in the current cut is determined by the Boolean function \texttt{hotEventEnabled()} (see Listing 3).

The edge for handling a cold violation by a minimal event is only enabled if the current event is the minimal event of the MSD. Furthermore, the event must not be currently enabled. The minimal event is enabled in the initial cut, e.g. in the configuration (0,0,0), but it may also be enabled in other configurations if the same message occurs multiple times in the MSD. Note that I define that the minimal event is enabled in the initial cut of the MSD, but that does of imply that the system is active, since the minimal event is always monitored. As discussed in Sect. 4, the system is only active when there is at least one active (enabled executed) event in any of the MSDs.
Figure 24: The TGA for the MSD RequestEnterAtEndOfTrackSection
In the following, I explain the implementation of the function declarations in Listing 3 and 4 in more detail. Listing 3 shows three Boolean functions created locally for each MSD template. The first function `enabled(int ev)` consists of a return statement that is a disjunction of expressions which encode for each visible event whether it is enabled in the current cut. The function `hotEventEnabled()` returns true if the current cut is hot. Its return statement is a disjunction of expressions over the lifeline variables, which encode for each hot message and condition in the MSD whether it is enabled. The function `eventInMSD(int event)` returns true if the event encoded in the integer parameter `ev` is an event in the MSD. Its return statement consists of a disjunction of expressions, which check for each visible event the equality of the parameter variable `ev` and the integer constant that is corresponding the visible event in the MSD.

Listing 3: The functions declared in the template `RequestEnterAtEndOfTrack-Section`

```cpp
/* return whether the event ev is enabled in the current cut of the MSD */
bool enabled(int ev) {
    return (ev == env_rc_endOfTS) and
            (RequestEnterAtEndOfTrackSection_env == 0 and
             RequestEnterAtEndOfTrackSection_rc == 0 and
             RequestEnterAtEndOfTrackSection_next == 0) or
            (ev == rc_next_requestEnter) and
            (RequestEnterAtEndOfTrackSection_rc == 1 and
             RequestEnterAtEndOfTrackSection_next == 1) or
            (ev == next_rc_enterAllowed) and
            (RequestEnterAtEndOfTrackSection_rc == 2 and
             RequestEnterAtEndOfTrackSection_next == 2));
}
```

```cpp
/* return whether any hot event is enabled in the current cut of the MSD */
bool hotEventEnabled() {
    return (RequestEnterAtEndOfTrackSection_rc == 1 and
             RequestEnterAtEndOfTrackSection_next == 1) or
            (RequestEnterAtEndOfTrackSection_rc == 2 and
             RequestEnterAtEndOfTrackSection_next == 2);
}
```

```cpp
/* return whether the event ev is in the MSD */
bool eventInMSD(int ev) {
    return (ev == env_rc_endOfTS or
             ev == rc_next_requestEnter or
             ev == next_rc_enterAllowed);
}
```

Listing 4 shows globally declared Boolean functions, completing the declarations shown in Listing 1. First, there are the functions of the form

1. `<MSD-name>_active(int ev)`, and
2. `<MSD-name>_hiddenEventEnabled()`
which are created for each MSD in the specification. Additionally, there are the Boolean functions

(4) \text{active}(\text{id}\ ev),

(5) \text{isHiddenEventEnabled}(), and

(6) \text{isSystemActive}()

which are created once per MSD specification. Let us discuss the implementation principle of the function kinds (1) and (2) first: The function RequestEnterAtEndOfTrackSection\_active\(\text{id}\ ev\) is an example of functions of kind (1). These functions consist of a return statement that is a disjunction of expressions for each executed visible event which encode whether the event is active in the current cut. The functions of kind (2) are the functions to determine whether there are hidden events enabled in the current cut of a particular MSD. These functions consists of a return statement which is a disjunction of expressions to determine for each hidden event whether it is enabled in the current cut. Since the MSD RequestEnterAtEndOfTrackSection only has one hidden event, namely the event of its termination, the function RequestEnterAtEndOfTrackSection\_hiddenEventEnabled() only returns whether or not the END event is enabled.

The function active\(\text{id}\ ev\) determines whether the event encoded by the integer parameter is active in any of the MSDs. It implements a return statement which is the disjunction of calls to the functions of kind (1) explained above. Since there is yet only one MSD in this example, just the function RequestEnterAtEndOfTrackSection\_active\(\text{id}\ ev\) is called here.

The function isHiddenEventEnabled() checks whether there is any hidden event enabled in any of the MSDs. It does so by a disjunction of calls to all the functions of the kind (2). Again, since there is yet only one MSD in this example, just a call to the function RequestEnterAtEndOfTrackSection\_hiddenEventEnabled()

The function isSystemActive() determines whether any of the MSDs in the specification is active, which means whether there is any event currently active in any of the MSDs. It implements a return statement that is a disjunction of calls to the function active\(\text{id} ev\) for each system event. Additionally, the system is active as long as there are hidden events enabled in the MSD. Therefore, a call to the function isHiddenEventEnabled() is added to the disjunction.

Listing 4: Global declarations (Part II)

```c
/* return whether the event ev is active in the current cut of MSD RequestEnterAtEndOfTrackSection */
bool RequestEnterAtEndOfTrackSection\_active\(\text{id} ev\)({
  return (\text{id} ev == rc\text{\_next\_requestEnter}) \text{ and }
  (RequestEnterAtEndOfTrackSection\_rc == 1 \text{ and }
   RequestEnterAtEndOfTrackSection\_next == 1) \text{ or }
  (\text{id} ev == next\text{\_rc\_enterAllowed}) \text{ and }
  (RequestEnterAtEndOfTrackSection\_rc == 2 \text{ and }
   RequestEnterAtEndOfTrackSection\_next == 2));
```
6.1 Mapping untimed MSD specifications

The above example only covers a small subset of the language constructs defined for LSCs and MSDs. Encodings for asynchronous messages, combined fragments (sub-diagrams for loops or if-then-else constructs) are tasks for future work. However, in the following I briefly describe the encoding principle for assignments are conditions. In Sect. 6.2 I explain the encoding of clock resets and time conditions in more detail.

6.1.3 Encoding assignments and conditions

Assignments and conditions are hidden events in the MSD and therefore they have to be considered in the implementation of the functions `<MSD-name>.-hiddenEventEnabled()` for the respective MSD they are in. Assignments are represented in the MSD automaton by an edge similar to the edge for the MSD END event as shown in Fig. 24. To this edge, the expression of the assignment is added as an additional update expression. I assume that assignments are restricted to the kind `<var> = <expr>` where `<var>` is a integer variable and `<expr>` is a valid UPPAAL expression which evaluates to an integer value. Furthermore, `<var>` is mapped to an integer variable declaration in the declarations of the respective MSD template automaton. Assignments and conditions can be extended to other data types supported by UPPAAL, but here only integers are considered. The edge of the assignment in the MSD automaton template is further guarded by an expression to make sure that the lifeline variables are at
a position such that the assignment is enabled. (This is the same expression as
the one in the function $<\text{MSD-name}>\_\text{hiddenEventEnabled}()$ mentioned above.)

Assignments and conditions are of course much more interesting when ex-
pressions could be formed over object attributes. Assignments, however, may
not assign values to object attributes directly, but only to variables declared in
the scope of an MSD \cite[pp. 108]{HM03}. Object attributes can only be modified
by messages which affect the attributes of the receiving object, i.e. param-
eterized messages which represent \textit{set}-method calls. The encoding of object
attributes and parameterized messages is however not elaborated in the scope
of this paper.

Cold conditions are each encoded by a pair of edges: one, called the \textit{pro-
gressing edge}, that is taken when the condition is enabled and the expression
evaluates to true, and another, called the \textit{violating edge}, that is taken when the
condition is enabled and the evaluation expression evaluates to false. The latter
edge updates all lifeline variables to 0.

The semantics of hot conditions is a little bit different from cold conditions,
because, when the cut reaches the hot condition, but their expression evaluates
to false, this does not result in a hot violation right away. Rather, this means
that the MSD cannot progress beyond this condition unless it evaluates to true,
and it is a hot violation only when that is never the case. So, hopefully, there
are other steps that the system can carry out that affect variables and object
properties, such that the condition eventually evaluates to true.

However, as long as object properties are not considered by this synthesis
approach, conditions can be formed only over MSD variables that have been
previously declared and assigned a value by assignments executed in preceding
cuts of the MSD. Therefore, once a hot condition is reached which evaluates
to false, there are little means for the system to affect the MSD variables such
that the expression eventually evaluates to true. Only in the rare situation that
other assignments are concurrently enabled with the hot condition in the same
MSD, there is a chance that executing such an assignment will change a variable
that affects a change in the evaluation of the hot condition expression.

Because of these restrictions, it makes sense to encode hot conditions in
such a way that a hot violation occurs right way when the cut reaches the
position of the hot condition, but its expression evaluates to false. To encode
this behavior, a pair of edges is created for each hot condition, similar to the
encoding of cold conditions explained above. In the case of the hot violation,
the violating edge updates the variable \textit{hotViolation} to true, which means that
the winning condition is immediately violated when the violating edge is taken.
If other assignments are concurrently enabled in the MSD that, as discussed
above, could turn the tide for the hot condition, the system automaton can
choose to progress these assignments first before having to take the violating
edge of the hot condition.

This way of immediately “producing” a hot violation is in fact similar to the
way that hot conditions with the expression \textit{false} are handled by the play-out
algorithm \cite[Sect. 9.4]{HM03}; since once the cut reaches such a condition, which
typically occur in anti-scenarios (see Fig. 9), it’s obvious that this implies a hot
violation and it is shown to the user right away.
In the future, when this approach is extended to consider object attributes, two alternative approaches for encoding a hot condition may be discussed. First, the original semantics of a hot condition, having to wait with progressing the cut until it evaluates to true, can be encoded as follows: Only the progressing edge is created for the hot violation, but additionally, the implementation of the function `<MSD-name>_hiddenEventEnabled()` is changed such that the hot condition is only considered enabled when its expression evaluates to true. The problem with this option is, however, that the restricted winning condition that is formulated in this synthesis approach does not capture a hot violation that results from waiting in front of a hot condition infinitely long. Therefore, another possibility is to prohibit the condition expressions to directly refer to object attributes, but instead require that object attribute values must be copied to MSD variables by assignments first. Under these language restrictions, pursuing the two-edge encoding as explained above would be reasonable.

6.1.4 Encoding environment assumptions

The environment automaton shown above in Fig. 24 can choose nondeterministically among all the possible environment events. However, a more restrictive model of the environment was discussed previously in Sect. 2.2 (see Fig. 8). These environment assumptions can be encoded as shown in Fig. 25. Here, the assumption automaton from Fig. 8 is modeled as a TGA, with the difference that the events are produced in the same two-step approach as shown above for the unrestricted environment automaton (assigning the event variable first, then emitting over the broadcast channel events).

![Figure 25: The environment template reflecting the environment assumptions (see Fig. 8)](image)

For now it is required that these restricted environment automata are modeled manually in UPFAAL TIGA.

One problem that yet remains in this synthesis approach with respect to environment assumptions is that the synthesis assumes that the system knows the state of the environment. In reality this is typically not the case, because
a system is informed of the state of the environment only by receiving the environment events. It is, for example, not possible for the RailCab to know that an obstacle is lying on the track section before it detects it through its sensors. Therefore the synthesis has to consider the partial observability of certain states. It is planned to extend Uppaal Tiga with this concept in the future BCD+07a BCD+07b.

6.2 Mapping timed MSD specifications

This section discusses the mapping from timed MSD specification to a model in Uppaal Tiga. The encoding principles for the environment and the system automaton templates change, which is explained in Sect. 6.2.1. The encoding principles for the MSD automata as presented in Sect. 6.1 are still valid. In a timed setting, only the encoding of clock resets and time conditions requires additional mapping definitions. Their encoding is explained in Sect. 6.2.2.

6.2.1 Environment and system automata for timed specifications

In an untimed setting, we assumed that the system was fast enough to perform an arbitrary finite number of steps before the occurrence of the next environment event. In a timed setting, where we specify that events can or must occur after or before certain time delays, it is overly optimistic to assume that no events occur in the environment while the system is taking time to perform certain reactions. Therefore, there are a number of modifications of the environment and system automata in a timed setting. Let us look at the modified environment automaton first.

Figure 26 shows the environment automaton for the timed setting. It is quite similar to the environment automaton for the untimed setting (see Fig. 25). The first difference is there is no invariant not isSystemActive() on the committed states. That means that the environment is not inhibited from producing events while there are reactions remaining to be performed by the system. This may even be overly pessimistic, since, when there are controllable and uncontrollable transitions enabled, Uppaal Tiga gives priority to uncontrollable events. But this issue is discussed later in Sect. 6.3. Second, this automaton includes the time assumptions made for the use case Drive Onto Merging Switch that were discussed previously in Sect. 2.2. see Fig. 10.

The system automaton for the timed setting is shown in Fig. 27. From the initial location systemActive, the system can nondeterministically choose active system events to produce next. It can only do so when there are no hidden events enabled, otherwise it must process these hidden events first. In contrast to the untimed system automaton, the system is not forced to produce system events immediately, but time may pass until a next step is taken. From the committed location produceEvent, the automaton must immediately progress to the committed location handleHiddenEvent and emit over the broadcast channel events while doing so. This is the same two-step principle of producing events as explained previously. In the committed location handleHiddenEvent, the system is forced to process all enabled hidden events immediately before the system
6.2 Mapping timed MSD specifications

can again progress to the location systemActive. Almost all hidden events are processed immediately in the location handleHiddenEvent. This additional edge for processing hidden events in the location systemActive is necessary because the system may not be able to immediately process hot lower-bound time conditions (minimal delays) (see Sect. 6.2.2).

When events are produced by the environment, the system automaton also moves to the location handleHiddenEvent to process hidden events before any other events can be produced by the system or the environment.

6.2.2 Encoding clock resets and time conditions

The encoding of clock resets and time conditions is explained in the following by the help of the example MSD ReplyOfSwitchControlNotTooEarly (see Fig. 9 and its Petri net representation in Fig. 17) that is encoded by the MSD automaton in Fig. 28.
The encoding of the minimal message event and the second message event follow the same principle as explained previously in Sect. 6.1. Also the edges for a hot violation and the two cases for the cold violation are created as explained before. The clock reset is encoded like a normal assignment except that the assigned variable is a clock variable.

The example furthermore shows the encoding of a cold time condition. As explained in Sect. 4.4, when the cut reaches a cold time condition in an MSD, it is immediately evaluated. When it evaluates to true, the cut progresses beyond the condition, when the cut evaluates to false, a cold violation occurs. Thus, a cold time condition is encoded with the same progressing/violating edge pair as an untimed cold condition.

Note that in Uppaal, one cannot simply form negations of time expressions (as it is done in the case of untimed cold conditions). Therefore, to negate the time constraint, the mapping must invert the compare operator, e.g. instead of writing not (c > 5), we must write c <= 5. Further, expression that test for the equality of a clock variable, like c == 5 are prohibited in cold conditions, because expressions like not (c == 5) or c != 5 are not allowed as guard expressions in Uppaal. In essence, this approach requires that the expressions in cold time constraints are restricted to the form c1 ⌧<expr> and c1 - c2 ⌧<expr> where c1 and c2 are clock variables, ⌧ is an operator <, ≤, ≥, >, and <expr> is an expression evaluating to an integer value. Further, the only variables allowed in <expr> are MSD variables defined by assignments prior to the condition.

For each cold time conditions, also the implementation of the functions of the form <MSD-name>_hiddenEventEnabled() is extended such that they return true when the the lifeline variables of the lifelines covered by the time condition are at the position immediately prior to the time condition (see Listing 5).

As explained previously, when the cut reaches a hot time condition, it either progresses beyond this condition immediately when the time constraint expression evaluates to true or, if the expression evaluates to false, the cut cannot progress until the expression renders true. A hot violation occurs when that is never the case. The same holds for hot time conditions. In this approach, I restrict the hot time condition expressions such that only minimal or maximal delays can be expressed. Minimal delays are lower time bounds of the form c ≥ <expr> or c > <expr> and maximal delays are upper time bounds of the form c < <expr> or c ≤ <expr>. Harel and Marelly make a similar restrictions [HM03]. The reason for this restriction is because this way, the play-out algorithm can immediately determine that a hot violation occurs when an upper-bound time condition (maximal delay) is reached and evaluates to false. If conjunctions of time expression were allowed, it would be hard to structurally determine whether they will become true in the future or not.

As discussed previously in the context of untimed hot conditions, because of the restricted winning condition formulated for this synthesis approach, only hot violations can be detected that happen due to violating steps. Hot violations that occur because the system never progresses an active MSD may in some cases not be detected. Due to the above restriction on the condition expressions in hot time conditions, the hot time conditions can be divided into such that,
6.2 Mapping timed MSD specifications

Figure 28: The automaton template for MSD ReplyOfSwitchControlNotToo-Early
when they evaluate to false at some point in time, never become true in the future and such that will in any case become true. For maximal delays it is therefore possible to immediately determine that a hot violation occurs when its expression evaluates to false. Because of this fact, hot maximal delays are encoded similarly to untimed hot conditions by a pair of a progressing and a violating edge, where the violating edge sets the variable hotViolation to true.

Hot minimal delays are conditions that the system can always progress at some point in the future. Therefore, each hot minimal delay is encoded by one (progressing) edge in the MSD automaton. For minimal delays, the implementation of the function `<MSD-name>_hiddenEventEnabled()` must however not evaluate to true when the cut is immediately prior to the hot minimal delay but its expression evaluates to false. (This would force the system to process a hidden event although it cannot yet do so.) It would be desirable to include the delay expression in the implementation of the `<MSD-name>_hiddenEventEnabled()` function. That, however, is not possible since UPPAAL and UPPAAL TIGA do not allow clock variables to be used inside function bodies. Therefore the implementation of the function `<MSD-name>_hiddenEventEnabled()` is not extended at all to return true when the cut is prior to a minimal delay. That means that, as soon as the delay expression evaluates to true, the system can choose to progress the cut beyond the hot minimal delay whenever it pleases. The synthesis algorithm will however try do so soon, because otherwise it will remain in a hot cut where the occurrence of violating events has fatal consequences. To encode that the cut is hot, the implementation of the function hotEventEnabled() for the respective MSD has to be extended such that it returns true when the cut is immediately prior to the minimal delay.

Listing 5 shows the functions which are added to the global declarations for the MSD ReplyOfSwitchControlNotTooEarly. Also, the listing shows how the implementation of existing functions (see Listing 1 and 4) are extended when the MSD ReplyBeforeLastSafeBreak (see Fig. 7) and ReplyOfSwitchControlNotTooEarly are encoded in the UPPAAL TIGA model.

Listing 5: Global declarations (Part III)

```c
int ReplyOfSwitchControlNotTooEarly_env = 0;
int ReplyOfSwitchControlNotTooEarly_rc = 0;
int ReplyOfSwitchControlNotTooEarly_next = 0;

/* return whether the event ev is active in the current cut of MSD RequestEnterAtEndOfTrackSection */
bool ReplyOfSwitchControlNotTooEarly_active(int ev)
{
    return false; // The MSD contains no executed events...
}
```

/* return whether the event ev is active in the...*/
6.2 Mapping timed MSD specifications

```c
bool active(int ev)
{
    return RequestEnterAtEndOfTrackSection_active(ev) or
           ReplyBeforeLastSafeBreak_active(ev) or
           ReplyOfSwitchControlNotTooEarly_active(ev);
}
... /* return whether any hidden event is enabled in the current cut of MSD */

bool ReplyOfSwitchControlNotTooEarly_hiddenEventEnabled()
{
    return (ReplyOfSwitchControlNotTooEarly_rc == 1 or
             ReplyOfSwitchControlNotTooEarly_rc == 3 or
             (ReplyOfSwitchControlNotTooEarly_env == 2 and
              ReplyOfSwitchControlNotTooEarly_rc == 4 and
              ReplyOfSwitchControlNotTooEarly_next == 1) or
             // the end of the MSD will never be reached...
             (ReplyOfSwitchControlNotTooEarly_env == 3 and
              ReplyOfSwitchControlNotTooEarly_rc == 5 and
              ReplyOfSwitchControlNotTooEarly_next == 2));
}
... /* return whether a hidden event is enabled in the current cut of ANY MSD */

bool isHiddenEventEnabled()
{
    return RequestEnterAtEndOfTrackSection_hiddenEventEnabled() or
           ReplyBeforeLastSafeBreak_hiddenEventEnabled() or
           ReplyOfSwitchControlNotTooEarly_hiddenEventEnabled();
}
```

Listing 6 shows the implementations of the functions declared locally for the MSD template automaton. Also, the clock variable declaration of the clock \( c \) is shown.

Listing 6: Global declarations (Part III)

```c
clock c;

bool eventInMSD(int ev)
{
    return (ev == env_rc_enterNext or ev == next_rc_enterAllowed);
}

bool hotEventEnabled()
{
    // there is no hot message nor hot minimal delay in the MSD...
    return false;
}

bool enabled(int ev)
{
    return ( \n```
\[ \begin{align*}
  & (ev \equiv env_{rc}\_enterNext \ \text{and} \ 
  \text{ReplyOfSwitchControlNotTooEarly}_{rc} \equiv 2 \ \text{and} \ 
  \text{ReplyOfSwitchControlNotTooEarly}_{env} \equiv 1) \\
  & \text{or} \ (ev \equiv next_{rc}\_enterAllowed \ \text{and} \ 
  \text{ReplyOfSwitchControlNotTooEarly}_{rc} \equiv 0 \ \text{and} \ 
  \text{ReplyOfSwitchControlNotTooEarly}_{next} \equiv 0 \ \text{and} \ 
  \text{ReplyOfSwitchControlNotTooEarly}_{env} \equiv 0)
\end{align*} \]

\}

6.3 Discussion

This section explained how the MSD synthesis problem can be mapped to an input for UPPAAL Tiga. The mapping showed how to encode the basic constructs of MSDs (messages, assignments, and conditions) in an untimed setting and in a timed setting (clock resets, time conditions). The encoding principles will have to be completed by elaborating the mapping of further MSD constructs like parameterized messages, asynchronous messages, and combined fragments (alternatives, loops). Mapping these additional constructs should, however, not pose a general problem.

Further, I discussed in this section that for timed MSD specifications, the super-step assumption is not practical. The super-step assumption means that the system takes all activated steps immediately and the environment events can only occur when the system is inactive or waiting for minimal delays. Instead, the setting for timed MSD specifications is chosen such that both system and environment events can occur when they are active/enabled.

The setting assumed here is even a little too pessimistic, because in UPPAAL Tiga, uncontrolled transitions have priority over controlled transitions. We could change the system and environment automaton so that the system can always take steps before problematic environment events occur. For example, the environment could be blocked from doing anything unless the system is waiting. If such a more optimistic setting is more practical and necessary will have to be discussed based on further practical examples.

The remaining problem is that the winning conditions which can be expressed in UPPAAL Tiga are not expressive enough to express that always eventually all active MSDs become inactive again [HKP05]. Therefore, a simpler winning condition is chosen, stating that never events occur which cause a hot violation, A [] not hotViolation. But, in the untimed case, this means that the algorithm can find a strategy which keeps the system active infinitely long, for example by a series of self-triggering MSDs. In other words, the system performs an infinite super-step and the environment is always blocked from producing any events. In the timed case, the synthesis could also find a strategy where the system is in an infinite loop. Then the environment is not explicitly blocked from producing events, but if environment assumptions are formulated which delay events in the environment, the system may get stuck in a time-less infinite loop (called zeno loop).

Thus, a winning strategy returned by the synthesis algorithm may not always be a valid implementation of the specification and therefore does not imply
that the specification is really consistent. Only when the algorithm cannot find any valid strategy, it is known the specification is inconsistent.

However, because UPAAAL TIGA can not only produce a single winning strategy, but can also generate all winning strategies for safety games, a second analysis step can be employed on this super-strategy to eliminate cycles that never again involve environment transitions. This, however, is not elaborated in the scope of this paper.

Even if we could formulate the winning condition suggested by Harel et al., “always eventually no MSD is active” [HKP05], it is questionable if this will always produce the desired result in a dense-timed setting as assumed here. Imagine a system that upon some environment event, for example “switch on” ends up in a loop that it only exits when another particular environment event occurs, like “switch off”. Then, whether the system will manage to eventually terminate all its active MSDs is at the mercy of the environment to let the “switch off” event occur. Thus, at least some further assumption on the environment has to be included, for example that always eventually each possible environment event occurs (fairness assumption).

### 6.4 Specifying environment assumptions with MSDs

It can be quite difficult to think of a single automaton that reflects all the environment assumptions. Instead it seems reasonable to specify the environment assumptions using the same paradigm that is used for the specification of the system requirements. Figure 29 shows an example of an MSD which states that a the lastBreak must not occur less than 5 seconds after endOfTS, as assumed in the use case Drive onto merging switch (see p. 12). This assumption is modeled as an anti-scenario: a clock \( c \) is reset to 0 upon the occurrence of endOfTS. Then, if this clock is not greater than 5 seconds upon the occurrence of lastBreak, a hot violation occurs because the cut reaches a hot false condition beyond which it can never progress. A hot violation in the environment means that this is something that cannot happen in the environment. This way, the language of the environment can be conveniently described. In some examples it may additionally be useful to specify a set of possible initial environment events.

![Figure 29: An example for an MSD which restricts the possible environment behavior](image-url)
The encoding of these assumption-MSDs would require little additional concepts. Primarily, hot violations must be encoded such that not the variable hotViolation is set to true, but another variable, for example envViolation is set to true. Then, the winning condition could be extended to $A[] not hotViolation or envViolation$, which means that the environment can never win when it performs a violating step. Furthermore, the messages must not be marked as executed by the system.

The environment assumptions should however be specified carefully, because it can lead to problems when the assumptions are too optimistic as discussed in Sect. 2. Also, the problem discussed in Sect. 6.1.4 remains, which is that the synthesis assumes that the system always knows the current state of the environment, which is typically not the case.

The details of including MSD-based environment assumptions will be elaborated in the future.

7 Realization of the mapping

The mapping of the MSD specification to a system of TGA is realized by a two-step transformation approach. First the MSD specification UML model is transformed into an ECore model of a TGA system (see Sect. A) via a TGG-based model-to-model (M2M) transformation (see Sect. B and C). Then this model is transformed to a valid input file for Uppaal Tiga via an XPAND model-to-text (M2T) transformation. Figure 30 illustrates this process. Between the TGG and XPAND transformations, there are two further steps involved that are not shown in the figure. First, a simple Java component sorts the function declarations in the TGA ECore model such that a function only has function calls to functions declared before its own declaration. This cannot be ensured by the TGG transformation and in Uppaal and Uppaal Tiga there must not be any functions with calls to functions declared after the calling function. Second, the negation of time conditions expressions (replacing the operator $<$ by $\ge$, or $\le$ by $>$, and vice versa) is carried out by another small Java component, since this cannot be conveniently achieved in TGGs or OCL. These two Java components and the XPAND transformation are assembled in an oAW (openArchitectureWare) workflow that can be executed in Eclipse.

![Figure 30: The two step transformation approach](http://wiki.eclipse.org/Xpand)

The remainder of this section briefly explains the principles and benefits of TGG transformations in Sect. 7.1 and 7.2. Section 7.3 presents describes
7.1 The principles of Triple Graph Grammars

Triple Graph Grammars (TGGs) are a formalism for defining sets of corresponding graphs. An element of this set is typically a triple consisting of two independent graphs that are linked via a third graph, called the correspondence graph. These graphs may be typed over different type graphs. Transferred to the “modeling world”, TGGs define sets of corresponding models where the independent models, in the following called domain models, are instances of different meta-models. The domain models are linked via a correspondence model that is an instance of a correspondence meta-model. In the following explanations the terms model and meta-model are used instead of graph and type graph.

A TGG defines a set of corresponding models by a number of TGG rules and an axiom. The TGG rules are graph production rules which describe how a triple of corresponding models (two domain models connected via a correspondence model) can be extended to form another, extended triple of corresponding models. The axiom represents a smallest triple of corresponding models that is the starting point for applying TGG rules.

Each TGG rule is a non-deleting graph grammar rule with nodes typed over classes in the meta-models of the domain- or correspondence model and edges that are typed over references in these meta-models. The nodes and edges can either be context nodes resp. context edges or produced nodes resp. produced edges. Context nodes are displayed as a white box with a black border, context edges are represented by black arrows. The produced nodes are displayed as green nodes with a “++” label. Similarly, the produced edges are shown as green arrows with a “++” label. If the context pattern can be matched in the instance model, the produced pattern can be created. Matching here means to find a pattern in the model with an isomorphic mapping to the context pattern of the rule where the type classes/type references of the nodes/links in the model pattern.

TGGs can not only be used to produce arbitrary corresponding graphs or models, but they can used in a number of practical application scenarios. For example, when one domain model is given, a TGG can very often be operationalized to find a model of the opposite domain and a fitting correspondence model such that they form a triple that is valid with respect to the TGG. This application scenario is called forward transformation where one domain is a source domain and the other domain is the target domain. Sometimes this works in the backward direction, too. If two domain models are given, a TGG can further be interpreted in order to find out whether they are corresponding.
with respect to the TGG. There are a number of further application scenarios, but here TGGs are used only to perform a forward transformation from a UML2-based MSD-specification to an ECore-based UPPAAL Tiga model. In the following, the principle of a forward transformation is explained. See Kindler and Wagner [KW07] and Greenyer and Kindler [GK10] for more comprehensive explanations and examples of TGGs and their application scenarios.

A forward transformation works as follows. Let us assume that a source model is given and shall be transformed into a target model according to a given TGG. First, one instance of the axiom has to be created. That means that a target model pattern and a correspondence model pattern have to be created and connected to the appropriate objects in the source model such that the structure matches the axiom. To mark this occurrence of the axiom, bindings are created between the objects in the model to the nodes in the axiom. Next, the TGG rules are applied in the following way. First the context pattern and the source domain pattern are matched in the triple model such that context nodes are matched only to already bound objects and the produced source domain nodes are matched only to unbound objects. If such a match can be found for a rule, the produced target and correspondence patterns are created in the model and additional bindings are created for these newly matched resp. created objects. This forward rule application schema is illustrated in Fig. 31.

The transformation finishes when no rule is applicable anymore. It is successful when no unbound objects remain in the source model. In practice often only parts of the model have to be transformed. Then, of course, the transformation can be considered successful when all the elements in the regarded view are bound.

7.2 The benefit of TGG transformations

TGGs have a number of benefits over other model transformation approaches. First, the transformation rules are relatively easy to comprehend and to maintain. One reason for this is the graphical syntax, but that is very often a matter of personal taste. Mainly, the rules are easy to understand because they just describe the relation of model patterns without any control structure. Controlled graph transformations easily become complex programs that are difficult to maintain.
Very similar to TGGs are the declarative QVT languages QVT-Relations and QVT-Core. The main idea in these languages is also to declaratively describe relations of model patterns by rules. We have intensively studied model transformation technologies in the past, especially comparing declarative QVT with TGGs and uncovered a number of ambiguities in the specification as well as some flaws in tools implementing QVT-Relations \cite{Gre06, GK07, GK10}. In the course of these and other projects a TGG transformation tool that interprets TGGs for transforming ECore models in Eclipse has been developed \cite{Gre06}. It has been constantly improved and extended in a number of projects at the Software Engineering group in Paderborn. This tool is used to implement the MSD-to-TGA mapping described here.

The benefits of a declarative model transformation language come with some drawbacks. First of all there is the transformation speed: performing a TGG transformation involves traversing and backtracking graph patterns and sometimes unsuccessfully trying match a number of rules. These operations are costly and controlled (graph) transformations are usually faster. On an Intel Core Duo 5500 (1.66 GHz) system, transforming the example specification with the two MSDs RequestEnterAtEndOfTrackSection (Fig. 6) and ReplyBeforeLastSafeBreak (Fig. 7) takes about 4.5 seconds. The example extended by the MSD ReplyOfSwitchControlNotTooEarly (Fig. 9) takes about 5.5 seconds. The time for matching a TGG rule may grow exponentially with the size of the rule and the size of the model, but the effect is not too dramatic in the most cases encountered so far. Plus, there are effective heuristics, e.g. search-plans, to contain the problem. Further, the transformation time may increase linearly with the number of rules, but there are also heuristics to limit the number of unsuccessful rule matches. Some simple heuristics are currently implemented in the TGG interpreter.

The speed of the transformation could be increased by not interpreting the rules, but by compiling code from the TGG rules for certain application scenarios. Other TGG engines exits, for example the Fujaba TGG engine \cite{Wag06} or the MOFLON TGG engine \cite{AKRS09}, which compile executable Java code from TGGs.

One further problem is that there is yet little convenient support for debugging and testing the transformations. Therefore finding mistakes in the TGG rules often requires reading log messages or debugging the interpreter implementation code.

However, despite the transformation speed, which can be improved in the future, the EMF TGG engine implements a number of features which makes it a flexible and convenient transformation tool for the mapping elaborated in this paper. Some of these features are described in the following section.

### 7.3 OCL constraints, generalization, and other extensions

The TGG mapping elaborated in this paper uses a number of features of the TGG interpreter that have not been documented before. These features are OCL constraints, stereotype constraints, and generalization. Also, in addition to context and produced nodes and edges, it is possible to specify reusable nodes
and reusable edges. The latter concept has been discussed before [GK10], but has not been thoroughly implemented thus far. These concepts are briefly explained in the following.

A TGG rule may contain a number of OCL attribute constraints, represented by yellow rounded rectangles. An attribute constraint contains an OCL expression and has a slot node and slot attribute. See for example the constraint in the TGG axiom PackageToNta in Fig. 37, the slot node is determined by the arrow with the double arrow hat, the name of the slot attribute occurs in the top label of the yellow rectangle. The OCL expression occurs in the bottom label. Their semantics is that a binding of a TGG rule is only valid if for each constraint the object bound to its slot node holds a value for the slot attribute that equals the evaluation of the constraint expression. The constraint expression can be any valid OCL expression and it is possible to use the names of the nodes in the rule as variables. These variables are initialized to hold the objects that the respective node is bound to. Operationally, the constraints on nodes that are matched in the current application scenario are only checked. The constraints on created nodes are enforced, which, adopting the term from QVT, means that the slot attribute value of the newly created object is set to the result of evaluating the constraint expression. Constraints are checked or enforced as soon as they can be evaluated, which is when all nodes referred to by variables in the expression are bound.

A TGG may furthermore reference a text file that contains a number of additional attribute or function declarations that can be used in the constraint expressions in the rules. Listing 7 shows an example of such definitions. There can also be constraints without any slot attribute that must have a constraint evaluating to a Boolean value. A binding of a TGG rule is valid only if all such constraints evaluate to true.

When UML models are involved in the transformation, it is possible to annotate the nodes with stereotypes that must or must not be applied to objects bound the nodes. Also, the slot attributes of attribute constraints may reference attributes of the stereotype definition as the slot attribute. The TGG rule HotCondition in Fig. 58 shows an example.

In addition to the produced nodes and the context nodes, a TGG rule may also contain reusable nodes, which are displayed as a gray box with a “##” label. Reusable nodes are nodes that can be interpreted as produced or context nodes. That means they may be bound to already bound nodes or that they may be bound to unbound or newly created objects. See [GK10] for a more detailed discussion on reusable nodes. This concept also applies to edges, which is possible since the TGG interpreter not only keeps track of node-object bindings, but also of edge-link bindings.

Operationally, in a forward transformation, such reusable patterns are useful in the source domain to say “match something that may or may not have been matched before”. In the target domain, it is sometimes desirable to create patterns only if they do not already exist. For example, the MSD-to-TGA transformation maps messages, among other things, to integer constants which represent their message type. In this case, this integer constant should be reused when multiple messages of the same message type (determined by the sending
and receiving instance and the message/operation name) are transformed. See the Message rule in Fig. 47.

The TGG interpreter implements a “priority on reuse”-semantics, which works as follows: first, as described above, the context and source-produced patterns are matched. When reusable nodes are encountered in this matching process, it is simply ignored whether a fitting object is already bound or not. Then, before creating the target patterns, reusable target and correspondence patterns are matched in the model. Whenever a valid match can be found of any connected reusable sub-pattern in the target or correspondence domain, then these model structures are reused. In other words, if there is a reusable pattern in the target/correspondence domain of the rule that is not entirely matched, then none of the matched objects are reused. If another reusable pattern that is not connected via reusable edges to the first can be entirely matched, then those objects are reused.

One concept that may significantly reduce the number and complexity of the rules in a TGG transformation is rule refinement or generalization as the concept is commonly called in object orientation. In many transformations there are rules that just handle a special case of another rule or there are rules that all have some parts of their patterns in common. In such cases it is desirable to reuse existing rules. In TGGs without any generalization concept, one would need to specify the same patterns in many rules redundantly, which has a negative impact on the maintainability of the transformation. Even worse, one would need to add constraints to the TGG rules for the more general case such that these rules are not applied when the more specialized rules can be applied.

To resolve this problem, it is possible to specify that one TGG rule is the refinement of another rule. The major concepts of refinement of TGG rules have been elaborated by Klar et al. [KKS07]. The basic idea is that in all application scenarios, wherever the refining rule is applicable, the refined rule would also be applicable. Further semantic details shall not be discussed at this point. If a TGG rule refines another rule, then the refining rule inherits the refined rule and may extend it in a number of ways. Klar et al. have defined a number of syntactic restrictions on the refining rules [KKS07, Sect. 4.2] that we partly adopt: first, the refining rule is allowed to specialize the type of nodes in the refined rule. More specifically, a representative of an inherited node in the refining rule, called refining node, must have the same type as the type of the refined node or the type must be a sub-type of the refined node’s type. Second, there may be additional constraints on the refining nodes and, if the node’s domain is UML, the refining node may require additional stereotype applications or stereotype applications of more specialized stereotypes than those required by the refined node. Third, the refining rule may contain additional patterns that must be connected to refining nodes. Last, if the refined node is a reusable node, the refining node may also be a context or produced node.

Operationally, in a forward transformation scenario, the refining rules are applied with priority over the refined rules, which means that a refined rule is only applied when all attempts of applying all rules refining it were unsuccess-
fully. Further, a TGG rule can be abstract, in which case it is never attempted to apply the rule.

7.4 Overview of the MSD-to-TGA mapping

This section explains the TGG transformation which formalizes and implements the MSD-to-TGA mapping as explained in Sect. 6. An overview of the rule structure is shown in Fig. 32 and 33. The actual rules are shown in the Appendix C and a semi-formal textual description of the UPPAAL Tiga model patterns produced by the rules is shown in the tables of Appendix B. Some of the explanations below refer to the domain meta-models, namely UML as explained in Sect. 4.5 and the UPPAAL Tiga ECore model as presented in Appendix A. The explanations are given with the forward transformation scenario in mind.

Given a UML model that contains a package with an MSD specification, the transformation starts out with applying the axiom PackageToNta to the MSD specification package and creating a corresponding NTA instance (the root model object for the UPPAAL Tiga ECore model). Based on this occurrence of the axiom, the rule MSDSpecification is applied. In Fig. 32 the rule dependencies are illustrated by dashed arrows. The MSDSpecification rule creates the basic structure of the environment automaton template and the system automaton template as well as their instantiations in the system declaration. Further, variables (event, hotViolation), channels (events, hiddenEvent), and function declarations (active(), isHiddenEventEnabled(), isSystemActive()) are created. Then, based on the occurrence of the MSDSpecification rule, the MSD rule can be applied for each MSD in the specification. This rule creates the basic structure of the MSD automaton template as well as its instantiation in the system declarations. Further, it creates some global function declarations (\langle MSD-name\rangle.active(int ev), \langle MSD-name\rangle.isHiddenEventEnabled(int ev)) as well as some function declarations local to the MSD automaton template (enabled(int ev), hotEventEnabled(), eventInMSD(int ev)).

Based on bindings of the rules MSDSpecification and MSD, the rule for mapping the lifelines, Lifeline, can be applied. This rule adds the lifeline variable declarations to the global declarations and adds some guard and update expressions to edges in the MSD automation template.

Messages can be transformed in the context of bindings of MSDSpecification and MSD. There are a number of different kinds of messages that are transformed differently. But, there are also some structures in the target model that have to be created for each such message. One way to avoid too many redundant patterns is the rule refinement structure shown in Fig. 32. In this figure, abstract rules are shown with an italic label.

The idea is that for all messages, the message type constants (const int <msg-type-name>) are created (or reused, see above) and a number of statements are added to functions declarations. Then, it has to be distinguished whether the message is a cold message, a hot message or a minimal message (which is assumed to be always cold). In these cases, for example different statements for progressing the lifeline variables and handling violating events need to
Figure 32: The relationship between the TGG rules for the mapping of the MSD specification, MSDs, lifelines, and the different kinds of messages
be considered. Further, messages that originate from an environment object are mapped to edges for “producing” events in the environment automaton template and messages originating from system objects are mapped to edges for “producing” events in the system automaton template. When messages are marked as executed, then the return statement of the function \(<\text{MSD-name}>\_\text{active}()\) has to be extended to return true when the respective message is enabled.

Similarly, there are different cases for mapping state invariants, since they can be assignments, clock resets, or a cold or hot untimed conditions, or a hot or cold minimal or maximal delay. Fig. 33 shows how these cases are handled by a refinement hierarchy of TGG rules. The rules for mapping constructs relevant in a timed mapping are highlighted with a thick bolder. Note also the rule \text{TimedMSDSpecification}, which specifies different structure for the system automaton template for timed MSD specifications. Each state invariant (and message) can be mapped in the context of \text{MSDSpecification} or \text{TimedMSDSpecification}.

![Figure 33: The relationship between the TGG rules for the mapping of the MSD specification, MSDs, assignments, clock resets, and the different kinds of conditions](image)

For each state invariant, a expression whether it is enabled is added to the global function \(\text{isHiddenEventEnabled}()\). Further, an edge in the MSD automaton is created which progresses the lifeline variables of the lifelines covered by the state invariant. For each assignment and clock reset, an integer resp. clock variable is created or reused. Additionally, the assignment or reset expression is added as an update statement to the progressing edge. For all conditions, except for the minimal delay, an additional edge for handling violations of the condition is added in the \text{Condition} rule. Depending on whether the condition is a hot or cold condition, update expressions are added that either set the \text{hotViolation} variable to true or expressions that reset the lifeline variables of the MSD. An exception here are hot minimal delays. The rule \text{HotMinimalDelay}
7.5 Discussion

OCL constraints and the concept of generalization on the level of TGG rules make TGGs a powerful and convenient language for defining executable mappings between models. Engineering this mapping presented here was a process that involved a fair amount of iterations and, in my opinion, re-thinking and changing the transformation would have been a lot harder using other state-of-the-art model transformation languages. In the future, however, further concepts for the systematic engineering of such transformations and especially means for debugging and testing the transformation are necessary. Also, even despite the generalization concept, not all redundancies in the TGG rules were eliminated. Therefore, additional concepts for the reuse of TGG rule parts should be investigated in the future, for example inspired by the “where” construct in QVT-Relations [QVT08].

8 Conclusion and outlook

During the design of increasingly complex systems it is important to provide engineers with intuitive, but precise means for describing the requirements on the system. Furthermore, automated analysis techniques are important in order to detect inconsistencies in the system design as early as possible in order to avoid costly iterations or safety-critical flaws in final product.

This paper presented a technique for finding inconsistencies in timed or untimed MSD specifications by mapping MSD specifications to the input for UPPAAL TIGA. UPPAAL TIGA implements efficient algorithms for finding winning strategies in timed or untimed two-player games. By the mapping presented in this paper, these efficient algorithms can be applied for synthesizing an MSD specification. It was discussed that the approach is incomplete due to the restrictions on the winning conditions that can be formulated in UPPAAL TIGA, but also measures were explained to remedy this deficiency in the future. It is expected, however, that the presented technique efficiently produces useful results in many cases, for example when there is no loop of self-triggering MSDs.

The mapping was formally defined and implemented by a set of graphical Triple Graph Grammar rules which map a UML-based MSD specification to a TGA system model. This TGA system model can be transformed to a valid input for UPPAAL TIGA by a model-to-text transformation. The whole transformation process can be run inside an eclipse modeling environment. The mapping was elaborated for a number of essential constructs in timed and untimed MSD specifications: hot or cold synchronous messages, assignments, clock resets, and hot or cold, timed or untimed conditions.

This paper also demonstrated the use of some advanced TGG concepts: generalization of TGG rules and OCL constraints, which make TGGs a powerful and convenient language for defining executable mappings between models.
8.1 Related work

At the same time of developing the synthesis technique presented here, a similar approach was worked out by Larsen et al. \cite{LLNP09,LLNP10}. Larsen et al. present a mapping from LSCs to a system of Timed Game Automata for consistency checking with UPPAAL Tiga. Their approach is to map each lifeline of each LSC in a specification to a timed automaton, which is similar to an encoding scheme pursued in earlier prototypes of the technique presented here \cite{Gre09}. Additional automata realize the synchronization of these lifeline automata, which makes the synchronization principle a little more complicated, but essentially the same runs can be described this way. As in the approach presented here, the winning condition is for the system to always avoid hot violations.

In contrast to the approach of Larsen et al., the approach presented here is based on MSDs instead of LSCs. This makes the approach more flexible, because universal diagrams are not subdivided into the prechart and mainchart, but monitored and executed messages may be arbitrarily mixed. One difference is furthermore that in the mapping by Larsen et al. it is not considered that a cold violation of a chart by the minimal event resets the chart to the cut after the minimal event (“initial+1”) instead of resetting it to the initial cut. It is of course legitimate to define the semantics of cold violations this way, but the mapping presented here seems closer to the actual invariant semantics as described by Harel et al. \cite{HM08}.

Furthermore, the mapping described by Larsen et al. is based on a self-defined LSC model whereas the mapping described here is based on a UML model that is the basis of or supported by a multitude of modern, industry-standard modeling tools today. Also, the mapping in this paper is thoroughly formalized and implemented by graphical mapping rules that are, compared to a hand-coded mapping, easy to understand and extensible in order to support further MSD language features in the future.

8.2 Outlook

In order to support the modeling and synthesis of further practical examples, the concepts for encoding parameterized messages, object attributes, and combined fragments need to be worked out in the future. Further mapping rules can be easily added to the mapping presented here. It could also be reasonable to split the mapping such that a first mapping defines a translation from MSDs to an intermediate model, for example a Petri net-like model as it was used for illustration purposes in Sect. 4.2. A second mapping could then map this model to a TGA system. This way it could be possible to better separate semantic concerns and implementation concerns.

This paper also discussed some ideas of using the scenario-based modeling paradigm not only for specifying the reactive behavior of the system, but also to model environment assumptions. These ideas will have to be refined and incorporated into the mapping in the future, but it seems that this will allow engineers to more intuitively design and compose scenario-based specifications.
The vision is to integrate the synthesis approach and the extensions described above into a convenient modeling tool suite for the modeling, simulation and synthesis of scenario-based specifications. This vision of SCENARIO TOOLS is illustrated in Fig. 34. The basis for SCENARIO TOOLS is currently developed in a project with a number of students at the Software Engineering Group in Paderborn. Some extensions to TOPCASED editors are developed to support the modeling of MSDs. Further, a play-out algorithm is developed for the simulation of the MSD specification. The play-out requires that the classes in the UML model are transformed into an ECore model. Then, an instance model can be created via generated EMF editors. The instance model can be a dynamic system, for example a track system with a number of RailCabs and track section controls, or it can be a static system like an ATM, with a fixed configuration of components. The simulation takes place by interpreting the MSD-Diagrams in the UML model upon environment events that are induced by the user or another component that simulates the environment. Where environment events trigger certain MSDs, the MSDs are dynamically bound to the objects in the system.

Figure 34: The vision of the SCENARIO TOOLS project

The idea is to synthesize only the MSDs that describe single use cases or subsets of use cases with respect to a static object system. The simulation can then be guided by the resulting strategies when occurrences of the use cases need to be simulated in different places of a dynamic system. Probably it will not be feasible to synthesize large specifications and even less will it be feasible to synthesize a specification referring to a dynamic object system. The latter will at least require additional concepts, since the synthesis approach described

5see http://www.topcased.org
here only considers a static object system. However, the play-out algorithm can be used for the simulation or systematic testing of specifications even in a dynamic setting. When the play-out is guided by strategies that avoid avoidable violations in certain places, the engineer has the more reason to suspect inconsistencies when hot violations do occur in the course or a simulation run.

If just one possible winning strategy is synthesized for part of the MSD specification, it is of course possible that exactly this strategy is inconsistent with the rest of the specification whereas there exists another strategy that would blend in consistently. Therefore, UPPAAL TIGA should be used such that it synthesizes a strategy that contains all possible winning actions in a certain state and not just one. Then play-out, possibly guided by other strategies for other parts of the specification, has a choice of actions that may avoid hot violations in other places. But of course the general problem remains, that the specification as a whole may be inconsistent, or that play-out just fails to find consistent runs, even though parts of the specification are consistent.

What’s more is that this synthesis approach does not anticipate all the admissible runs of the system, but only those that are expected to happen due to the monitored/executed modes of the messages (see Sect. 4 and 4.2). When for example an MSD specification is extended by further MSDs, it may be that events which were only monitored before are now executed in some runs of the play-out. These runs are then not covered by a synthesized strategy. This could be changed by encoding also monitored messages such that the system can actively execute them. But when can we expect the system to be inactive again? This question will have to be discussed in the future.

If no winning strategy is returned by the synthesis algorithm for the MSD specification or parts thereof, this implies that the specification is inconsistent. A valuable feature of UPPAAL TIGA is that it can then return a counter-strategy for showing how the environment can violate the specification. However, from simulating the counter-strategy inside UPPAAL TIGA, on the basis of the TGA system, it is very hard to gain insights on the level of the MSDs. Therefore, the vision is to also have an environment simulation that can be guided by a counter-strategy. Then, the engineer could observe how the specification can be violated in terms of the MSDs and an appealing graphical representation of the system.

Acknowledgment

I would like to thank Ekkart Kindler, Shahar Maoz, Oliver Sudmann, Jan Rieke, Dietrich Travkin, Matthias Meyer, Björn Axenath, and Stefan Henkler for helpful discussions. I also thank Sascha Burdick, Nils Diekmann, Jens Frieben, Markus Fockel, Mathias Höckelmann, Lilija Klassen, Daniel Löhelt, and Daniel Simon for their work on the SCENARIOTOOLS project.
References


REFERENCES

(SCESM’02), 24th International Conference on Software Engineering (ICSE’02), May 2002. ACM, 2002.


REFERENCES


[KW01] Jochen Klose and Hartmut Wittke. An Automata Based Interpretation of Live Sequence Charts. In Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’01), held as Part of the Joint European Conferences on Theory and Practice of Software,
REFERENCES


REFERENCES


A The meta-model for Uppaal Tiga

The UPPAAL TIGA ECore meta-model captures the parts of the UPPAAL TIGA language that are used in this synthesis approach. A brief explanation of the meta-model is given in the following; see the tutorial on UPPAAL [BDL04] or the online help of the UPPAAL TIGA tool (version 4.13) for a more comprehensive description of the language.

The first class diagram in Fig. 35 shows the meta-classes that are used for describing the automaton templates, the global resp. template-specific declarations, and the declarations of TGA systems (i.e. the instantiation of automaton templates). The root of each model in UPPAAL TIGA is called NTA (network of Timed Automata). It contains a number of automaton templates and a system declaration that defines a number of instantiations of these templates. Each template consists of locations and edges that have one source and one target location. The NTA and each template may have declarations that consist of declarations of functions, channels, clocks, and Boolean and integer variables. Functions may themselves have parameter variable declarations. In this model, to declare a Boolean variable for example, a BooleanDeclaration has to be paired with a VariableID that determines the name and, optionally, an initial value for that variable. When multiple variable IDs are contained in a boolean declaration, this leads to declarations like bool a = true, b = false;.

Note that not each possible instance of the meta-model leads to valid input for UPPAAL TIGA. For example, according to the meta-model, a channel may be initialized with a String expression, or the parameter variable declarations for functions may have an initializer, which is not possible in UPPAAL TIGA. The aim of this model is primarily to support the MSD-to-TGA mapping in a convenient way. The TGG transformation uses this model in such a way that the XPAND transformation produces valid input files.

Most properties of the locations and edges are encoded by String or Boolean attributes. For the invariant expressions of locations and the guard expressions of edges, the model defines the class BooleanExpression. Further, the update expression of edges is a Statement. Figure 36 shows the meta-classes for functions, statements, and expressions. An expression can either be a plain text expression, an integer literal, or a Boolean expression. Again, this is not intended to represent a valid grammar for UPPAAL TIGA expressions, but shall just serve the MSD mapping in a convenient way. A Boolean expression can either simply be an expression given in the form of a text string, but it can also be a conjunction or disjunction of other boolean expressions, or a binary compare expression between two other expressions. The compare expressions can use a number of compare operators given in the enumeration CompareOperator. A boolean expression can further be the call to a boolean function with zero or more arguments. A function in this model may have a number of statements. In the scope if this paper, however, only Boolean functions are created with a single return statement that is a Boolean expression.
Figure 35: An ECore meta-model for UPPAL Tiga (automaton templates, global/template declarations, system declarations)

Figure 36: An ECore meta-model for UPPAL Tiga (functions, statements, expressions)
B Mapping tables

Each row in the following tables gives an abstract, text-based summary of the structures mapped by a transformation rule in the MSD-to-TGA mapping. The left column shows the name of the transformation rule and, if it is a refining rule, states which other rule that rule is a refinement of. Further, the left column shows references to rule diagrams that show the TGG rule or parts of the TGG rule in Sect. C.2.

The right column describes the patterns that are to be created in the TGA system. Since the rule names closely resemble which structures of the MSD specification are to be mapped to the TGA system, there is no extra column which describes the MSD specifications patterns more closely.

To distinguish model parts that are required as context or parts that are produces or reused, different colors are used. Black, underlined text is used to describe context elements or patterns. Green text is used to describe produced elements or patterns, and gray text with a wavy underline is used to describe reused patterns or elements. A typewriter font is used to represent concrete expressions or names of elements. Expressions in angle brackets, for example <MSD-name>, represent String functions that are used in attribute constraints in the TGG rules. Very often these functions correspond to OCL functions that are listed in Listing 7.

Table 1 lists all rules shown in Fig. 32. Table 2 lists all rules for mapping untimed assignments and conditions, Table 3 lists the rules for mapping clock resets and time conditions and it lists the changes in the mapping of the MSD specification to the system automaton template in a timed setting.

B.1 Untimed MSD specifications

Table 1: MSD to TGA Mapping: MSD specification, MSDs, lifelines, and messages

<table>
<thead>
<tr>
<th>MSD specification (see Fig. 38)</th>
<th>• environment automaton template Env</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o initial location environmentInitial</td>
</tr>
<tr>
<td></td>
<td>o committed location produceEvent (produceEvent, environmentInitial)</td>
</tr>
<tr>
<td></td>
<td>- invariant: not isSystemActive()</td>
</tr>
<tr>
<td></td>
<td>o uncontrollable edge (produceEvent, environmentInitial)</td>
</tr>
<tr>
<td></td>
<td>- synch: events!</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

| (see Fig. [39]) | • system automaton template Sys  
|                 |   o initial location systemInactive  
|                 |   o committed location systemActive  
|                 |   o committed location produceEvent  
|                 |   o edge (systemInactive, systemActive)  
|                 |     - synch: events?  
|                 |   o edge (systemActive, systemInactive)  
|                 |     - guard: not isSystemActive()  
|                 |   o edge (produceEvent, systemActive)  
|                 |     - synch: events!  
|                 |   o edge (systemActive, systemActive)  
|                 |     - synch: hiddenEvent!  
| (see Fig. [40]) | • global declarations  
|                 |   o broadcast chan events;  
|                 |   o chan hiddenEvent;  
|                 |   o bool hotViolation = false;  
|                 |   o int event = 0;  
|                 |   o bool active(int ev){return (...)}  
|                 |   o bool isHiddenEventEnabled(){  
|                 |     return(...);}  
|                 |   o bool isSystemActive(){  
|                 |     return(isHiddenEventEnabled() ...);}  
|                 | (Here (... represents a disjunction to which  
|                 | boolean expressions are added later on.)

Continued on next page
### B.1 Untimed MSD specifications

#### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>System Declarations</th>
<th>MSD Automaton Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>(see Fig. 41)</td>
<td>(see Fig. 42)</td>
</tr>
<tr>
<td>• system declarations (instantiate system and environment automata templates)</td>
<td>• MSD automaton template <code>&lt;MSD-name&gt;</code></td>
</tr>
<tr>
<td>◦ sys = Sys();</td>
<td>◦ initial location <code>initial</code></td>
</tr>
<tr>
<td>◦ env = Env();</td>
<td>◦ committed location <code>violation</code></td>
</tr>
<tr>
<td>◦ system = sys, env, ...;</td>
<td>◦ edge for the hot violation (<code>initial</code>, <code>violation</code>)</td>
</tr>
<tr>
<td></td>
<td>- synch: <code>events?</code></td>
</tr>
<tr>
<td></td>
<td>- guard: <code>eventInMSD(event)</code> and not <code>enabled(event)</code> and <code>hotEventEnabled()</code></td>
</tr>
<tr>
<td></td>
<td>- update: <code>hotViolation=true</code></td>
</tr>
<tr>
<td></td>
<td>◦ edge for the cold violation (<code>initial</code>, <code>initial</code>)</td>
</tr>
<tr>
<td></td>
<td>- synch: <code>events?</code></td>
</tr>
<tr>
<td></td>
<td>◦ edge for the cold violation by the minimal event (<code>initial</code>, <code>initial</code>)</td>
</tr>
<tr>
<td></td>
<td>- synch: <code>events?</code></td>
</tr>
<tr>
<td></td>
<td>◦ edge for the MSD END event (<code>initial</code>, <code>initial</code>)</td>
</tr>
<tr>
<td></td>
<td>- synch: <code>hiddenEvent?</code></td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 1 – continued from previous page

| (see Fig. [13]) | • global declarations  
|                 |   o <MSD-name>_active(int ev){ return (...)OR;}  
|                 |   o <MSD-name>_hiddenEventEnabled(){ return ((...)AND or ...)OR;}  
|                 |   o bool active(int ev){ return (... or <MSD-name>_active(ev))OR;}  
|                 |   o bool isHiddenEventEnabled(){ return (... or <MSD-name>_hiddenEventEnabled())OR;}  
| (see Fig. [14]) | • template declarations  
|                 |   o bool enabled(int ev){  
|                 |     return (...OR;}  
|                 |   o bool hotEventEnabled(){  
|                 |     return (...OR;}  
|                 |   o bool eventInMSD(int ev){  
|                 |     return (...OR;}  
| (see Fig. [12]) | • system declarations (instantiate MSD automaton template)  
|                 |   o <MSD-name>_inst = <MSD-name>();  
|                 |   o system = sys, env, ..., <MSD-name>_inst;  
| Lifeline (see Fig. [15]) | • global declarations  
|                 |   o int <lifeline-var-name> = 0;  
|                 |     (declare lifeline variable, where  
|                 |     <lifeline-var-name> is a string  
|                 |     concatenation of the name of the MSD and  
|                 |     the name of the object represented by the  
|                 |     lifeline, <MSD-name>_<object-name>)  
|                 |   o <MSD-name>_hiddenEventEnabled(){  
|                 |     return ((... and <lifeline-var-name>  
|                 |     == <lifeline-max>)AND or ...)OR;}  

Continued on next page
### B.1 Untimed MSD specifications

#### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>MSD automaton template</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>• edge for the cold violation</td>
<td>• global declarations</td>
</tr>
<tr>
<td>- update: ..., $\langle\text{lifeline-var-name}\rangle = 0$</td>
<td>- $\text{const int } \langle\text{msg-type name}\rangle = \langle\text{position in constant list}\rangle$;</td>
</tr>
<tr>
<td>- edge for the cold violation by the minimal event</td>
<td>(constant declaration per message type)</td>
</tr>
<tr>
<td>- update: ..., $\langle\text{lifeline-var-name}\rangle = 1$</td>
<td>• MSD automaton template</td>
</tr>
<tr>
<td>- guard: ... and $\langle\text{lifeline-var-name}\rangle == \langle\text{lifeline-max}\rangle$</td>
<td>- edge for the message event (initial, initial)</td>
</tr>
<tr>
<td>- update: ..., $\langle\text{lifeline-var-name}\rangle = 0$</td>
<td>- synch: events?</td>
</tr>
</tbody>
</table>

(the return statement of the function is a disjunction of expressions which evaluate to true when a hidden event is enabled. One expression is a conjunction that evaluates whether all lifeline variables have reached their maximal value. The expression above is added to that conjunction. $\langle\text{lifeline-max}\rangle$ is the maximal value of the lifeline variable, i.e. (=number of locations - 1) when the lifeline is the source or target of the minimal message, (=number of locations) otherwise.)

$\langle\text{see Fig. 46}\rangle$
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Minimal Message (see Fig. [48])</th>
<th>Minimal Environment Message (see Fig. [49])</th>
<th>Minimal System Message (see Fig. [50])</th>
<th>Non-Minimal Message (see Fig. [51])</th>
</tr>
</thead>
<tbody>
<tr>
<td>• template declarations</td>
<td>• MSD automaton template</td>
<td>• system automaton template</td>
<td>• MSD automaton template</td>
</tr>
<tr>
<td>◦ bool eventInMSD(int ev){return (... or ev == &lt;msg-type name&gt;) on;}</td>
<td>◦ edge for the message event</td>
<td>◦ edge (systemActive, produceEvent)</td>
<td>◦ edge for the message event</td>
</tr>
<tr>
<td></td>
<td>- update: &lt;increase all MSD’s lifeline variables&gt;</td>
<td>- guard: active(&lt;msg-type name&gt;) and not isHiddenEventEnabled()</td>
<td>- update: &lt;increase the variables of the sending and receiving lifeline&gt;</td>
</tr>
<tr>
<td>◦ edge for the cold violation</td>
<td>- guard: eventInMSD(event) and not enabled(event) and not event == &lt;msg-type name&gt; and not hotEventEnabled()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>◦ edge for the cold violation by the minimal event</td>
<td>- guard: event == &lt;msg-type name&gt; and not enabled(event)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• template declarations</td>
<td>◦ bool enabled(int ev){return (... or (ev == &lt;msg-type name&gt; and &lt;all MSD’s lifeline variables equal 0&gt;) on);}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### B.1 Untimed MSD specifications

#### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Description</th>
<th>Example Code</th>
</tr>
</thead>
</table>
| **Cold Message** | (contains just a constraint that the temperature of the message is “cold”) | ```
bool enabled(int ev){
    return (... or (ev == <msg-type name>
and <variable of sending lifeline at sending location and <variable of receiving lifeline at receiving location ) OR ;}
``` |
| **Cold Environment Message** | see Minimal Environment Message | |
| **Cold System Message** | see Minimal System Message | |
| **Cold Executed System Message** (see Fig. 52) | | ```
bool <MSD-name>_active(in ev){
    return ( ... or (ev == <msg-type name>
and <variable of sending lifeline at sending location and <variable of receiving lifeline at receiving location ) OR ;}

bool isSystemActive(int ev){ return (...
or active(<msg-type name>) ) OR ;}
``` |
| **Hot Message** (see Fig. 53) | | ```
bool hotEventEnabled(){ return (... or
<variable of sending lifeline at sending location and <variable of receiving lifeline at receiving location ) ) OR ;}
``` |
| **Hot System Message** | see Minimal System Message | |
| **Hot Executed Message** | see Cold Executed System Message | |
Table 2: MSD to TGA Mapping: Assignments and Conditions

<table>
<thead>
<tr>
<th>State Invariant</th>
<th>Assignments and Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(see Fig. 54)</td>
<td>• global declarations</td>
</tr>
</tbody>
</table>
|                 |   o `<MSD-name>_hiddenEventEnabled(){
|                 |     return (... or
|                 |     `<condition-enabled-expr>`
|                 | }` |
|                 | (Here `<condition-enabled-expr>` is an expression that is checking whether the lifeline variables of all lifelines covered by the state invariant are at the position immediately before the state invariant.) |
|                 | • MSD automaton template   |
|                 |   o progress edge          |
|                 |     - guard: (`<condition-enabled-expr>` AND `hiddenEvent?`) |
|                 |     - synch: `hiddenEvent?` |
|                 |     - update: `<increment-lifeline-vars>` |
|                 | (Here `<increment-lifeline-vars>` is a set of statements incrementing the lifeline variables of each lifeline covered by the state invariant.) |

Continued on next page
Table 2 – continued from previous page

| Assignment (refines State Invariant) (see Fig. 59) | • template declarations  
  ◦ variable int <assignment-variable>  
• MSD automaton template  
  ◦ progress edge  
  - update: ..., <assignment-statement>  
  (The expression of the state invariant shall be an assignment statement of the form \( \text{<var>} = \text{<expr>} \) (see Sect. 6.1.3). This statement is split up such that \( \text{<assignment-variable>} \) is the variable expression (string before the equals sign) and \( \text{<assignment-statement>} \) is the complete statement expression.) |

| Condition (refines State Invariant) (see Fig. 56) | • MSD automaton template  
  ◦ edge  
  - guard: (<condition-enabled-expr> and <condition-expr>) AND  
  ◦ violating edge  
  - guard: (<condition-enabled-expr>) AND  
  - synch: hiddenEvent? |

| Cold Condition (refines Condition) (see Fig. 57) | • MSD automaton template  
  ◦ violating edge  
  - update: <reset-lifeline-vars-expr>  
  (Here <reset-lifeline-vars-expr> is an expression that resets the lifeline variable of each lifeline covered by the condition to 0.) |

| Hot Condition (refines Condition) (see Fig. 58) | • MSD automaton template  
  ◦ violating edge  
  - update: hotViolation = true |
B.2 Timed MSD specifications

<table>
<thead>
<tr>
<th>Timed MSD Specification (see Fig. 59)</th>
<th>System automaton template Sys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• environment automaton (same as for an untimed MSD specification, except that location produceEvent has no invariant.)</td>
</tr>
<tr>
<td></td>
<td>(In the scope of this paper, I do not consider the automatic generation of environment assumption automata.)</td>
</tr>
<tr>
<td></td>
<td>• system automaton template Sys</td>
</tr>
<tr>
<td></td>
<td>• initial location systemActive</td>
</tr>
<tr>
<td></td>
<td>• committed location produceEvent</td>
</tr>
<tr>
<td></td>
<td>• committed location handleHiddenEvent</td>
</tr>
<tr>
<td></td>
<td>• edge (for handling hidden events 1/2) (systemActive, systemActive)</td>
</tr>
<tr>
<td></td>
<td>• synch: hiddenEvent!</td>
</tr>
<tr>
<td></td>
<td>• edge (systemActive, handleHiddenEvent)</td>
</tr>
<tr>
<td></td>
<td>• synch: events?</td>
</tr>
<tr>
<td></td>
<td>• edge (handleHiddenEvent, systemActive)</td>
</tr>
<tr>
<td></td>
<td>• guard: not isHiddenEventEnabled()</td>
</tr>
<tr>
<td></td>
<td>• edge (for handling hidden events 2/2) (handleHiddenEvent, handleHiddenEvent)</td>
</tr>
<tr>
<td></td>
<td>• synch: hiddenEvent!</td>
</tr>
<tr>
<td></td>
<td>• edge (produceEvent, handleHiddenEvent)</td>
</tr>
<tr>
<td></td>
<td>• synch: events!</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th><strong>Hot Minimal Delay</strong> (see Fig. 61)</th>
<th><strong>Clock reset</strong> (refines State Invariant) (see Fig. 62)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• template declarations&lt;br&gt;  ◦ bool hotEventEnabled(){ return (... or&lt;br&gt;  &lt;condition-enabled-expr&gt; ) or; }&lt;br&gt;  • MSD automaton template&lt;br&gt;  ◦ progress edge&lt;br&gt;    - guard: &lt;condition-enabled-expr&gt; and&lt;br&gt;    &lt;condition-expression&gt; &lt;br&gt;    - update: &lt;increment-lifeline-vars&gt;</td>
<td>• template declarations&lt;br&gt;  ◦ clock &lt;clock-variable&gt;.&lt;br&gt;  • MSD automaton template&lt;br&gt;  ◦ progress edge&lt;br&gt;    - update: ..., &lt;reset-statement&gt;</td>
</tr>
<tr>
<td>(The expression of the state invariant shall be an assignment statement of the form &lt;clock-var&gt; = &lt;expr&gt; This statement is split up such that &lt;clock-variable&gt; is the variable expression (string before the equals sign) and &lt;reset-statement&gt; is the complete statement expression.)</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Condition</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold Time</td>
<td>(see Cold Condition, Tab. 2)</td>
</tr>
<tr>
<td>Hot Maximal Delay</td>
<td>(see Hot Condition, Tab. 2)</td>
</tr>
</tbody>
</table>
C TGG transformation

Section C.2 lists all TGG rules in the MSD-to-TGA mapping. Section C.1 lists the OCL definitions that are utilized in the TGG attribute constraints.

C.1 OCL attribute definitions

Listing 7: Custom OCL definitions

```ocl
package uml
context Lifeline
  def: varName : String =
    self.interaction.name.concat('_')
    .concat(self.represent.name)
  def: minimalEventOnLifeline : Boolean =
    self.interaction.fragment->at(1).covered
    ->includes(self) or
    self.interaction.fragment->at(2).covered
    ->includes(self)
  def: allFragments : OrderedSet(InteractionFragment) =
    self.interaction.fragment
    ->select(iff.covered->includes(self))
  def: maxVarValue : Integer =
    self.allFragments->size()
    + (if self.minimalEventOnLifeline then 0 else 1 endif)
  def: getPositionBefore(f:InteractionFragment) : Integer =
     self.allFragments->indexOf(f)
     - (if self.minimalEventOnLifeline then 1 else 0 endif)

  // taken http://wiki.eclipse.org/OCLSnippets:
  def: toString(i:Integer) : String =
    OrderedSet{1000000, 10000, 1000, 100, 10, 1}->iterate(
      denominator : Integer;
      s : String = ' ' |
      let numberAsString : String = OrderedSet{
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }
        ->at(i.div(denominator).mod(10) + 1)
      in
        if s=' ' and numberAsString = '0' then
          s
        else
          s.concat(numberAsString)
        endif
    )
  def: getPositionBeforeString(f:InteractionFragment) :
     String = self.toString(self.getPositionBefore(f))

context Message
  def: typeName : String =
    self.sendEvent
    .oclAsType(uml::MessageOccurrenceSpecification).covered
    ->any(true).represents.name.concat('_')
    .concat(self.receiveEvent
    .oclAsType(uml::MessageOccurrenceSpecification).covered
    ->any(true).represents.name.concat('_')
    .concat(self.sendEvent
    .oclAsType(uml::MessageOccurrenceSpecification).event
```
context Interaction

def: increaseAllLifelineVariablesExpr : String =
  self.lifeline.iterate(ll : Lifeline; expr : String = '') |
  if expr = '' then ll.varName.concat('++')
  else expr.concat('').concat(ll.varName).
    concat('++').endif

def: allLifelineVariablesZeroExpr : String =
  self.lifeline.iterate(ll : Lifeline; expr : String = '') |
  if expr = '' then ll.varName.concat('==0')
  else expr.concat('').concat(ll.varName).
    concat('==0').endif

def: resetAllLifelineVariablesExpr : String =
  self.lifeline.iterate(ll : Lifeline; expr : String = '') |
  if expr = '' then ll.varName.concat('==0')
  else expr.concat('').concat(ll.varName).
    concat('==0').endif

context StateInvariant

def: enabledExpr : String =
  self.covered.iterate(ll : Lifeline; expr : String = '') |
  if expr = '' then ll.varName.concat('==').
    concat(ll.getPositionBeforeString(self)).
  else expr.concat('==').concat(ll.varName).
    concat('==').
    concat(ll.getPositionBeforeString(self)).endif

def: increaseAllCoveredLifelineVariablesExpr : String =
  self.covered.iterate(ll : Lifeline; expr : String = '') |
C.1 OCL attribute definitions

```ocl
context OpaqueExpression

def resetAllCoveredLifelineVariablesExpr : String =
self.covered->iterate(ll : Lifeline; expr : String = '' |
if expr = '' then ll.varName.concat('++')
else expr.concat(',').concat(ll.varName)
.concat('++') endif
)
def: helperSequence : Sequence(Integer) =
Sequence{1..self.body->any(true).size()}
/* this is a little tricky, because OCL does not support
many string operations. Probably it would be better
to extend OCL: */
def: firstOccOfEquals : Integer = self.helperSequence
->iterate(counter : Integer; result : Integer = 0 |
if ((self.at(counter) = '=' or self.at(counter) = '<')
and result = 0) then counter else result endif)
/* Just checks whether the expression contains an '<': */
def: isMaximalDelay : Boolean = self.helperSequence
->iterate(counter : Integer; result : Integer = 0 |
if ((self.at(counter) = '<')
and result = 0) then counter else result endif) > 0
def: varName : String =
self.body->any(true)
.substrings(1, self.firstOccOfEquals-1)
```

endpackage
C.2 Rules for the untimed MSD specifications

Figure 37: TGG Axiom PackageToNta
Figure 38: TGG Rule \texttt{MSDSpecification} (environment automaton template)
Figure 39: TGG Rule `MDSpecification` (system automaton template)
Figure 40: TGG Rule **MSDSpecification** (global declarations)
Figure 41: TGG Rule **MSDSpecification** (system declarations)
Figure 42: TGG Rule MSD (MSD automaton template and its instantiation)
Figure 43: TGG Rule MSD (global declarations)
Figure 44: TGG Rule MSD (template declarations)
Figure 45: TGG Rule Lifeline (global declarations)
C.2 Rules for the untimed MSD specifications

Figure 46: TGG Rule Lifeline (update and guard expressions in the edges of the MSD automaton template)
Figure 47: TGG Rule Message
Figure 48: TGG Rule MinimalMessage
Figure 49: TGG Rule MinimalEnvironmentMessage
Figure 50: TGG Rule MinimalSystemMessage
Figure 51: TGG Rule NonMinimalMessage
Figure 52: TGG Rule ColdExecutedMessage
Figure 53: TGG Rule HotMessage
C.2 Rules for the untimed MSD specifications

Figure 54: TGG Rule StateInvariant
Figure 55: TGG Rule Assignment
Figure 56: TGG Rule Condition
Figure 57: TGG Rule ColdCondition
Figure 58: TGG Rule HotCondition
C.3 Rules for the timed MSD specifications

Figure 59: TGG Rule `TimedMSDSpecification` (environment automaton template)
Figure 60: TGG Rule TimedMSDSpecification (system automaton template)
Figure 61: TGG Rule HotMinimalDelay
C.3  Rules for the timed MSD specifications

Figure 62: TGG Rule ClockReset
Figure 63: TGG Rule ColdTimeCondition
Figure 64: TGG Rule HotMaximalDelay