Graph Transformation Planning with Time and Concurrency

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
an der
Fakultät für Elektrotechnik, Informatik und Mathematik
der
Universität Paderborn

vorgelegt von

Steffen Ziegert, M.Sc.

Paderborn 2016

Abstract

The increasing complexity of technical systems inspires software and systems engineering scientists to improve the state of the art in designing such systems. Among these improvements is the integration of cognitive functions into approaches to model-driven software development. Such cognitive functions enable an autonomous operation of the system, e.g., by planning reconfiguration behavior affecting the software architecture of the system. In this context, this thesis is concerned with the automated generation of reconfiguration plans.

By providing a formal framework for the rule-based modification of graphs or graph-like structures, graph transformation systems enable to model the dynamics of structures. As a consequence, they are particularly convenient for modeling reconfiguration behavior of software architectures. However, graph transformation systems have only rarely been employed as system models for planning techniques.

Motivated by different requirements arising from two fundamentally different application examples, two approaches for graph transformation planning have been developed in this thesis.

The first approach preserves the expressiveness of graph transformation systems by directly working on a graph transformation system's state space. As a result, it can handle system models with an infinite state space. It employs a domain-specific heuristic function that uses the solution length of a relaxed planning problem as heuristic estimate. Taking both the structure of graphs and applicable graph transformations into account, this is a considerable improvement over related work.

The second approach puts its focus on timing aspects and concurrency. It comes with a new formalism for the specification of *durative* graph transformations. This formalism ensures that multiple durative graph transformations with conflicting behavior cannot be executed concurrently. Furthermore, it enables the explicit, rule-based specification of requirements regarding their concurrent and urgent execution. By being based on *timed* graph transformation systems, it also allows to employ available verification procedures. System models that have been designed in this formalism can be translated into planning domains, for which problem instances can be solved by employing off-the-shelf planning systems. Evaluation results give insight on how to decide between different translation variants and convoy an idea how certain aspects of planning domains influence planning performance.

Zusammenfassung

Die zunehmende Komplexität von technischen Systemen motiviert Forscher im Software und Systems Engineering den Stand der Technik der Entwicklung solcher Systeme zu verbessern. Zu diesen Verbesserungen gehört die Integration kognitiver Funktionen in Ansätze der modellgetriebenen Softwareentwicklung. Solche kognitive Funktionen ermöglichen einen autonomen Betrieb des Systems, z.B. durch eine Planung von Rekonfigurationen, die die Softwarearchitektur des Systems beeinflussen. In diesem Zusammenhang beschäftigt sich diese Arbeit mit der automatischen Erstellung von Plänen solcher Rekonfigurationen.

Indem sie ein formales Framework für die regelbasierte Modifikation von Graphen und Graph-ähnlichen Strukturen zur Verfügung stellen, ermöglichen es Graphtransformationssysteme, die Dynamik von Strukturen zu modellieren. Sie sind damit besonders zur Modellierung von Rekonfigurationen einer Softwarearchitektur geeignet. Bisher wurden Graphtransformationssysteme jedoch nur selten als Modelle für Planungsverfahren eingesetzt.

Motiviert durch die unterschiedlichen Anforderungen zweier grundverschiedener Anwendungsbeispiele, wurden in dieser Arbeit zwei Verfahren zur Planung mit Graphtransformationen entwickelt.

Das erste Verfahren erhält die Ausdruckskraft von Graphtransformationssystemen, indem es direkt auf dem Zustandsraum eines Graphtransformationssystems arbeitet. Aus diesem Grund kann es mit Modellen umgehen, die einen unendlichen Zustandsraum aufspannen. Es verwendet eine domänenunabhängige Heuristik, die die Länge der Lösung eines relaxierten Planungsproblems als Schätzwert liefert. Sie berücksichtigt sowohl die Struktur des Graphen als auch die anwendbaren Graphtransformationen, was eine deutliche Verbesserung gegenüber verwandten Arbeiten darstellt.

Das zweite Verfahren legt seinen Fokus auf Zeitaspekte und Nebenläufigkeit. Es bringt einen neuen Formalismus zur Spezifikation von *zeitkonsumierenden* Graphtransformationen mit sich. Dieser Formalismus stellt sicher, dass mehrere zueinander im Konflikt stehende zeitkonsumierende Graphtransformationen nicht nebenläufig ausgeführt werden können. Des Weiteren ermöglicht er die explizite, regelbasierte Spezifikation von Anforderungen bezüglich ihrer nebenläufigen und eiligen Ausführung. Indem er auf *zeitbehafteten* Graphtransformationen aufsetzt, ermöglicht er außerdem die Verwendung bereits verfügbarer Verifikationsverfahren. In diesem Formalismus entwickelte Modelle können in Planungsdomänen übersetzt werden, dessen Probleminstanzen mit Standard-Planungssystemen gelöst werden können. Auswertungsergebnisse helfen zwischen unterschiedlichen Varianten dieser Übersetzung zu entscheiden und vermitteln eine Idee, inwiefern die Performanz der Planung durch verschiedene Aspekte der Planungsdomäne beeinflusst wird.

Acknowledgments

First of all, I would like to thank my PhD advisor Prof. Dr. Heike Wehrheim for her support and guidance during these past five years and the opportunity to write this PhD thesis. I would further like to thank Prof. Dr. Wilhelm Schäfer for this time and interest in the evalation of my thesis and Dr. Theo Lettmann, Prof. Dr. Leena Suhl, and Prof. Dr. Christian Plessl for serving as thesis comittee members.

Special thanks go to Dr. Dominik Steenken, Dr. Claudia Priesterjahn, Dr. Christian Heinzemann, Oliver Sudmann, Tobias Meyer, Christoph Rasche, and Prof. Dr. Matthias Tichy for the productive and enjoyable collaboration within the Collaborative Research Centre 614 and overlapping research interests.

I would also like to thank the (former) members of our research group Dr. Thomas Ruhroth, Dr. Nils Timm, Dr. Galina Besova, Daniel Wonisch, Sven Walther, Alexander Schremmer, Oleg Travkin, Tobias Isenberg, Marie-Christine Jakobs, Manuel Töws, Julia Krämer, and Elisabeth Schlatt for providing a warm and inspiring atmosphere. A similar contribution has been made by various colleagues from across the floor. Thank you for making coffee breaks more enjoyable!

Additionally, I would like to thank my student assistants Shayan Ahmadian and Johannes Geismann for supporting me in developing parts of the thesis implementation and my thesis advisees Marcel Friedrich, Thomas Hauck, and Johannes Heil for interesting discussions in research themes related to this PhD thesis.

Finally, I would like to thank my wife and parents for their support and patience during these years of research (and all those years before) and my brother for sparking my interest in computer science in the first place.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

In today's economy, more and more technical systems contain large amounts of software. The increasing complexity of these systems gave rise to the use of modeling languages, which allow to create a model of the system that is to be developed. Such a model usually abstracts details of the system away or allows to hide them in different views, thus making the model easier to understand. However, the opportunities of employing modeling languages during the development of software systems go far beyond that of abstractly representing systems for easier discussion and documentation.

Model-Driven Software Development (MDSD) [SV06] tries to benefit from the existence of models by considering them as first-class artifacts during the development of software systems. The aim of MDSD is to enable the generation of code from models, e.g., via model transformation techniques [OMG11], making a lengthy and error-prone direct implementation unnecessary. A related goal is to enable the analysis of the same models, e.g., to verify their correctness or to ensure a certain level of quality. The benefits of a well-functioning MDSD approach are obvious: software systems are much easier to develop and errors can be found earlier in the development process.

For an MDSD approach to function properly, its system models need to have a formal foundation, i.e., a mathematical basis that unambiguously defines a model's meaning. Development techniques in the area of software engineering and hardware design that provide such a formal foundation are called *formal methods*. Examples for formal methods include process calculi, like Hoare's *Communicating Sequential Processes (CSP)* [Hoa78] and Milner's *Calculus of Communicating Systems (CCS)* [Mil80], formal specification languages, like the *Z notation* [Spi92; ISO02] and *Alloy* [Jac06], and automata theory.

MDSD approaches, like MechatronicUML [Bec+12], are likely to combine multiple domain-specific modeling languages, each specialized to a certain kind of modeling task. Examples for such modeling tasks include modeling the *structural*

relationship of software components, *communication behavior* between different components, and *reconfiguration behavior*. The latter states how the structural relationship of software components may change over time. Because reconfiguration impacts the software architecture of a system, it is usually treated separately from other behavior.

The increasing complexity of technical systems also inspired software and systems engineering scientists to look into different fields, like control theory, optimization, and artificial intelligence, to improve the state of the art in designing those systems, cf. [GRS14]. Among these improvements is the integration of cognitive functions into MDSD approaches. This enables subsystems of the system under consideration to operate autonomously and thus ensures that they require only low maintenance. These cognitive functions allow to perceive situations and add some kind of partial intelligence to the technical system.

To enable a technical system to operate autonomously, one has to integrate a means of making decisions into this system. For each decision, there may be a large set of alternatives. Selecting which alternative to put into practice should not be done in isolation from other decisions. Systems operating autonomously usually have goals that are supposed to be reached during operation, like optimizing the consumption of time or resources, or achieving user-specified objectives. These goals have to be taken into account when deciding which alternatives to realize. However, recognizing those alternatives that are likely to help in achieving the goal can be a complex task. If these decisions were to be made by humans, the response-time requirements of many technical systems would not be met. As a consequence, the system needs a software component that plans which alternatives to take.

Executing some of the chosen alternatives may involve reconfigurations to the system's software architecture, such as the creation and deletion of software component instances or communication links between them. Systems that autonomously decide when and how to perform these reconfigurations, are said to have a *self-organizing* [GMK02] or *self-managing* [Bra+04] architecture. Multiple architectural models have been proposed for the development of such systems, e.g., the *Operator-Controller Module (OCM)* [HOG04], which was developed as part of the *Collaborative Research Centre "Self-Optimizing Concepts and Structures in Mechanical Engineering" (CRC 614)*, or Kramer and Magee's *reference model for self-managing systems* [KM07]. A schematic representation of the OCM is given in Figure 1.1.

Both architectural models consist of three layers. The bottom layer, called *controller* (in the OCM) or *component control* (in the reference model), accomplishes the most basic tasks of the system. It essentially provides the implementation of primitive features related to sensors and actuators. The middle layer, called *reflective operator* (in the OCM) or *change management* (in the reference model), has the capability to modify the system's architecture, e.g., it selects operating parameters for the bottom layer or executes software architecture reconfigurations. The top layer, called *cognitive operator* (in the OCM) or *goal management* (in the reference model), accomplishes time-consuming tasks, like the computation of a plan that determines which decision alternatives to realize. In a system with a self-managing architecture,

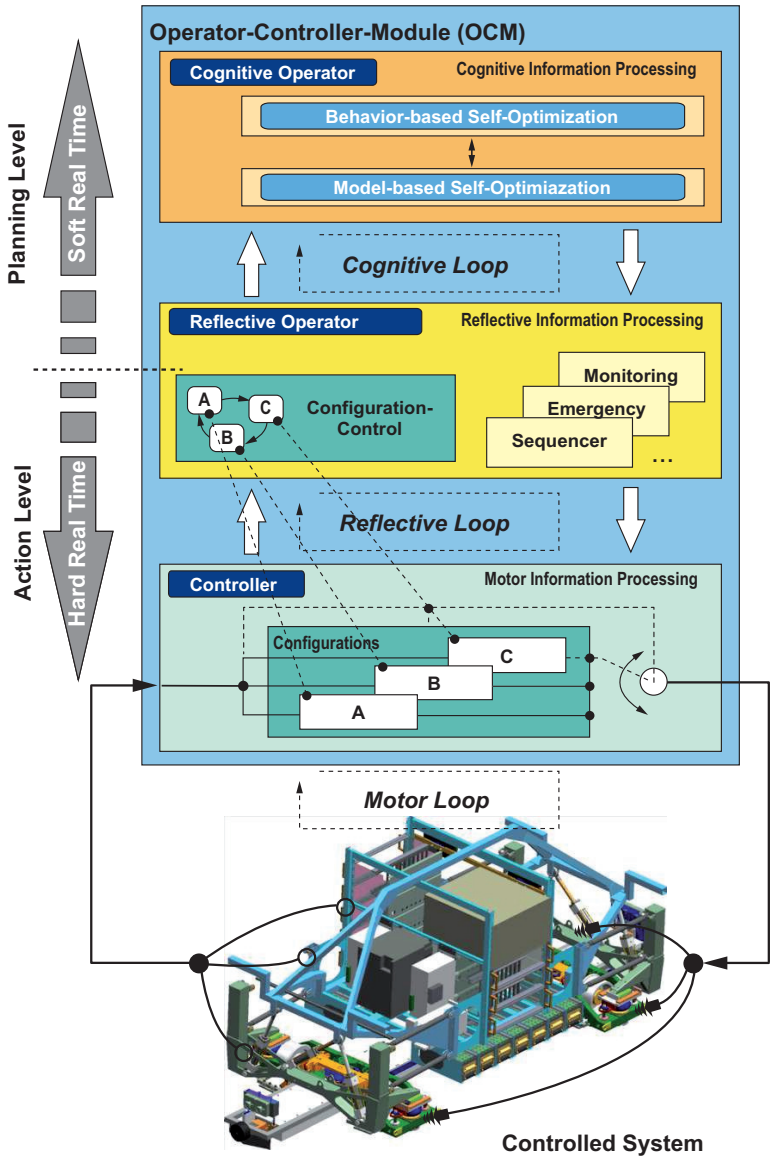


Figure 1.1: Structure of the Operator-Controller Module

such a plan states which architecture reconfigurations to perform and when. This thesis is specifically concerned with this last layer of those architectural models, i.e., with the automated generation of reconfiguration plans.

1.1 Automated Planning

Automated planning is a discipline in the area of artificial intelligence, coming along in many different variants. In most of these variants, some kind of agent has to choose among some set of activities which one to perform. Usually, there is a notion

of a state or configuration, and each activity defines a transition between two such states. States and state transitions can be represented in almost any kind of form.

Independently of the manner chosen to represent system states, a planning task always has some kind of initial state and a goal specification. The goal specification determines whether a state of the state space is a valid end state for the purpose of the planning task. If a planning system finds such an end state in the state space originating from the initial state of a planning task, then the path from the initial state to the end state constitutes a valid plan. Usually, there is also some kind of objective involved, e.g., state changes can have costs, which are to be minimized. In the most simple case, these costs are distributed uniformly, i.e., the objective is to reach the goal in as few steps as possible. If time is of the essence, the objective is usually to reach the goal in as little time as possible.

A conventional representation for actions and states of planning problems, which is used throughout the AI planning research community, is based on (quantifier-free) predicate calculus. In this representation, an action is schematically defined via a set of atomic formulas that are required to hold, a set of atomic formulas that are asserted as *true*, and a set of atomic formulas that are asserted as *false*. This classical formalism is called *STRIPS*, named after a planning system developed by Fikes and Nilsson [FN71] in 1971. It is still in use today within the planning research community and has been integrated into a common language, called the *Planning Domain Definition Language (PDDL)*, by McDermott and the AIPS-98 Planning Competition Committee [MA98] in 1998. PDDL has since been extended by several other contributors. The most relevant extensions with regard to this thesis are *typing*, which allows to employ a type hierarchy for objects appearing as terms in atomic formulas, and *durative actions* [FL03], which introduce a notion of time and concurrent execution into PDDL. Further extensions that are made use of in this thesis include the support for *numeric formulas* and *quantification*.

Naturally, the application of planning techniques is not restricted to such classical representations. Planning has also been applied to graphical models such as *Petri nets* [Pet62], e.g., for solving assembly problems in manufacturing [Zha89; McC94], and in more recent times, to *graph transformation systems* [Ehr+06], e.g., for solving reconfiguration problems in the context of cyber-physical systems [EW11].

Both Petri nets and graph transformation systems are formal modeling languages with rigorous mathematical definitions and execution semantics. Petri nets are very well suited for modeling the concurrent behavior of distributed systems, and graph transformation systems enable to model the *dynamics of structures* by providing a formal framework for a rule-based modification of graphs or graph-like structures. The latter is particularly convenient for modeling reconfiguration behavior of software architectures.

1.2 Rule-Based Modification of Graphs

In graph transformation systems, the modification of graphs is specified via graph transformation rules. Each graph transformation rule defines a condition that has to

be fulfilled by a graph so that the rule may be applied to this graph. If the condition is fulfilled, the rule gives one or more options how the graph may be transformed into a new graph.

Since graphs provide an intuitive way to describe complex concepts and relations, graph transformations offer a wide range of application areas. Research on graph transformation started in the late 1960s in the fields of pattern recognition and compiler construction. Since then, graph transformations have been applied in software engineering, database design, modeling of concurrent systems, logical programming, model transformation, and many other areas. Their strength lies in their ability to model the dynamics of graphical structures. For this reason, graph transformations have been considered a new paradigm for developing software, especially if this software is of complex structure.

A lot of structural information appears in the fields of software development and visual modeling. In object-oriented design, for example, there is structural information in the relationship between different classes and objects. Computer networks and component-based software systems are also built using a large amount of structural information. All this structural information can be expressed via graphs, and their evolution can be expressed via graph transformations.

Due to its ability to specify how structures evolve over time, graph transformations have been used for the specification of software architecture reconfiguration, e.g., by Wermelinger and Fiadeiro [WF99; WF02], by Taentzer at al. [TGM00], or by Le Métayer [Le 98]. Since graph transformations have a formal foundation, they have also been used for verification. A prominent example is the tool set GROOVE [Ren04], which provides explicit CTL and LTL model checking for graph transformation systems [KR06; Ren08]. There are also approaches to the more specific case of verifying software architecture reconfiguration, e.g., a symbolic invariant checking technique by Becker et al. [Bec+06], which allows to prove the absence of forbidden graph patterns, and a compositional verification approach by Eckardt et al. [Eck+13], which includes the verification of timed properties. However, graph transformation systems have only rarely been employed as system models for planning techniques.

1.3 Research Tasks and Contributions

The main purpose of this thesis is to design planning techniques based on graph transformations. The use of graph transformations renders these planning techniques suitable for software architecture reconfiguration and allows for an integration with MDSD approaches. Depending on the application scenarios of interest, e.g., whether or not they involve timing aspects and concurrent behavior, there are different requirements for such graph transformation planning systems.

In general, reconfigurations of a system's software architecture take time. If multiple such temporal reconfigurations are non-conflicting, they can probably be carried out in parallel. Requiring a strictly sequential execution of reconfigurations might even be counterintuitive in certain application domains, e.g., where reconfigurations

concern highly independent components or different agents in multi-agent systems. However, most current planning approaches to architectural reconfiguration do not support time, and consequently only generate non-temporal plans, which do not include concurrent execution.

Leaving aside planning and concerning only specification, there are *timed story patterns* and *timed story diagrams* [HH11] for defining timed architectural reconfigurations. They are an extension of *story patterns* and *story diagrams*, which combine UML activity diagrams with story patterns, and have a formal semantics based on *timed graph transformation systems (TGTS)* [Neu07]. However, they provide only functionality for specifying *when* something is supposed to happen, not for *how long* it is supposed to happen.

To enable an intuitive specification of reconfigurations whose execution requires time, we also need a formalism in which *durations* can be assigned to reconfigurations. In such a formalism, temporal reconfigurations should be allowed to be executed concurrently if no conflict arises in doing so. Multiple concurrent reconfigurations with conflicting behavior, e.g., the deinstantiation and use of a software component at the same time, should be prevented from being executed without the need for a domain modeler to explicitly address their potential conflicts. The formalism should also have a means to conveniently express certain dependencies between reconfigurations, e.g., that certain reconfigurations have to be applied concurrently or only a very small time frame between two reconfigurations is allowed. Finally, the formalism should go well with existing approaches for the verification of software architecture reconfigurations, but also allow for an application of temporal planning techniques.

Since graph transformation systems have been deemed suitable for modeling software architecture reconfiguration, it seems like a natural choice to use graph transformations for the formal syntax and semantics of such a formalism. A possible alternative to graph transformation systems would have been to employ a variant of Petri nets. There are extensions of Petri nets that have been used for modeling systems that rewrite their structure dynamically, e.g., *(high-level) net transformation systems* [PER95], *dynamically modifiable Petri nets* [RR04], and *reconfigurable Petri nets* [LO04]. Petri nets have also been extended to support notions of time, e.g., *time Petri nets* [Mer74; BD91], where transitions are augmented with enabling intervals, *timed Petri nets* [Ram73; VFC95], which associate transitions with firing durations, and *interval timed colored Petri nets* [AO95], which attach time to tokens and require them to reside in transitions' output places for certain holding durations. In the AI planning research community, Petri nets have also been used for analyzing planning domains [Vaq+09].

Nevertheless, the decision has been made in favor of graph transformation systems. This decision was supported by its great suitability for modeling architecture reconfiguration and the benefit of employing timed graph transformation systems. Timed graph transformation systems extend graph transformation systems with clocks and time constraints, in a similar way that *timed automata* [AD94] extend ordinary automata. While the application of rules is instantaneous in timed graph

transformation systems, their time constraints provide a convenient means for developing a formal semantics where the execution of a graph transformation consumes time. Furthermore, there exist verification procedures for timed graph transformation systems: Heinzemann et al. [HE10] provide a reachability analysis, which supports to check whether or not a certain subgraph exists in any state graph of the state space, and Suck et al. [SHS11] developed an approach that translates a timed graph transformation system and a specification given as a first-order TCTL formula into a TCTL model checking problem [ACD93] for timed automata.

The contributions of this thesis cover two different approaches to planning with graph transformations. The first approach is concerned with untimed planning tasks. It works directly on the state space of a graph transformation system by employing a domain-independent heuristic function during the generation of the state space. An important aspect of this approach is that it preserves the expressiveness of graph transformation systems, where it is possible to model infinite systems. Related planning approaches that also employ domain-independent heuristic functions while searching through the state space of a graph transformation system are rather simple: either their heuristic function does not take the applicable graph transformation rules into account, cf. [EJL06; Sni11], or the actual structure of state graphs, cf. [HHV11].

The second approach to planning with graph transformation puts the focus on timing aspects and concurrency. For this reason, the contributions of this thesis include the design of a formalism for the specification of *durative* graph transformations as well as a means to generate *temporal* reconfiguration plans for domain models that have been specified using this formalism. Due to the formal syntax and semantics this formalism provides, it is suitable for safety-critical environments where a precise timed execution is essential.

The formal semantics is based on the timed graph transformation framework mentioned above. Syntactically, a durative graph transformation rule extends an ordinary graph transformation rule by a natural number, called its *duration*, representing its execution time. To yield a semantics, these durative rules are mapped to existing concepts of the timed graph transformation framework. This mapping integrates a *locking mechanism*, which guarantees that the execution of multiple durative rules with conflicting behavior is avoided. As a consequence, there is no need for a modeler to explicitly address potentially dangerous conflicts between concurrent reconfigurations on the level of syntax.

In addition to the durative rules mentioned above, this formalism provides concepts to express certain dependencies regarding their application. These concepts have been designed specifically for the use in concurrent contexts. They allow to express that certain reconfigurations require the concurrent execution of other reconfigurations or that they have to follow other reconfigurations within a given deadline.

In order to generate temporal plans, we perform a translation of system models that have been specified in this new formalism into PDDL planning domains. The result of running an off-the-shelf planning system on problem instances over such

a generated planning domain is a temporal plan that yields the exact points in time when a reconfiguration starts and finishes. This translation reproduces the fundamental features of the formal semantics, thus ensuring that PDDL-based planning systems generate plans that are valid under the semantics of durative graph transformations.

While the formal syntax and semantics of durative graph transformations have been designed such that a translation into PDDL is possible and reasonable, they are conceptually independent of PDDL or any planning techniques. The semantics is solely based on timed graph transformation systems. For this reason, we can also make use of the verification procedures for timed graph transformation systems given in [HE10; SHS11].

1.4 Application Examples

The motivation for exploring planning systems working with graph transformations stems from several application examples. These application examples are fundamentally different in structure, leading to different requirements for the planning system. This is why different approaches to graph transformation planning have been developed, each with different strengths and weaknesses.

In this section, we informally introduce those application examples that were used for validating the planning systems developed as part of this thesis.

Reconfiguration of ECUs Our first application scenario is the reconfiguration of electronic control units (ECU) in automotive systems. The AUTOSAR consortium proposed a component-based software architecture standard [AUT14] for the development of ECU software. Following the AUTOSAR standard, a Runtime Environment (RTE) is generated out of a predefined set of components. This RTE acts as a middleware, which connects software components with Basic Software (BSW) that controls the hardware, see Figure 1.2.

In current development, software components are deployed on ECUs at design-time. We expect that in the future, only interfaces to the hardware have to be deployed at design-time and software components can be deployed at runtime. This allows to react to hardware failures by initiating a self-healing process, which involves software architecture reconfiguration, cf. [Klö+10]. Such a self-healing process computes a reconfiguration plan and executes it subsequently. In this application example, reconfiguration plans include actions to deploy software components on ECUs, create and destroy component instances, and shut down ECUs.

Note that we assume important system tasks to be performed redundantly. Therefore, hardware failures do not necessarily result in the failure of such crucial tasks. As a consequence, the safe operation of the system is not affected immediately and the self-healing process can be executed in soft real-time. As soon as the reconfiguration of the system is completed, the redundancy in the system is fully reestablished.

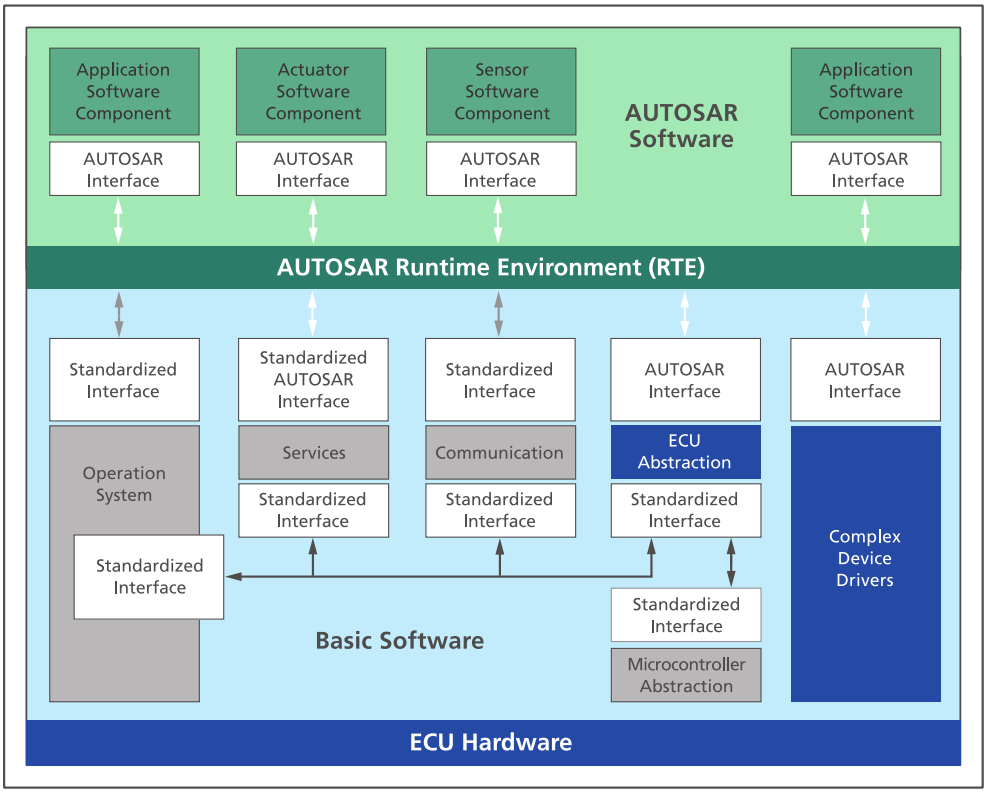


Figure 1.2: Layered architecture of AUTOSAR[1]

RailCab system Our second application scenario is the RailCab system, which is developed at the University of Paderborn. The RailCab system consists of autonomous, driverless shuttles, called RailCabs, that operate on a railway system. Each RailCab has an individual goal, e.g., transporting passengers or goods to a specified target station. An important feature of the RailCab system is the RailCabs' ability to drive in a convoy, i.e., RailCabs can minimize the distance between each other, thus saving energy due to the reduced drag caused by the slipstream effect. To safely operate in a convoy, acceleration and braking has to be coordinated and managed between convoy members. RailCabs also communicate with trackside base stations to inquire properties of upcoming track segments, like their expected coefficients of friction.

The software of the RailCab system is developed in MechatronicUML, where software components interact via communication protocols by means of message passing. To define these communication protocols MechatronicUML provides *real-time coordination patterns (RTCP)*.

Figure 1.3 shows a configuration where such RTCPs are connected to component instances. It consists of three RailCab component instances driving in a convoy. They

[1]Reproduced by kind permission of dSPACE GmbH.

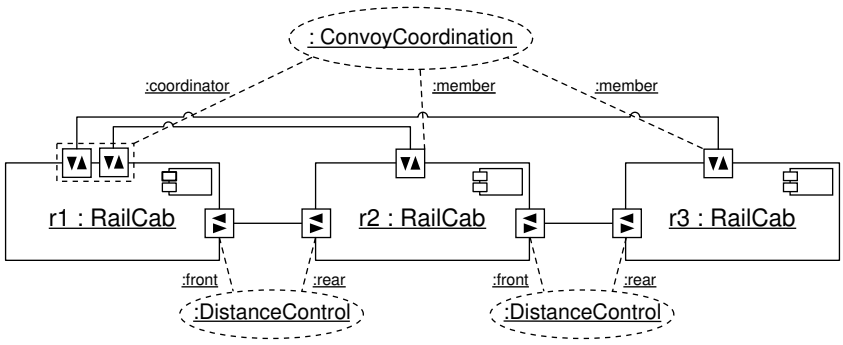


Figure 1.3: Component instance configuration of three RailCabs operating in a convoy

execute the RTCPs ConvoyCoordination and DistanceControl. Each of these RTCPs consists of two interfaces, called *ports*, and a connector, i.e., there is a `coordinator` and `member` port for the RTCP ConvoyCoordination and a `front` and `rear` port for the RTCP DistanceControl.

The behavior of the involved communication partners, called *roles*, is specified via *real-time statecharts (RTSC)*, which combine UML state machines [OMG09] with timed automata. The RTCP ConvoyCoordination defines a *1-to-n* communication. During convoy operation, one RailCab serves as a coordinator, which defines a reference speed and coordinates acceleration and braking by communicating with all other convoy members. Therefore, RailCab `r1` executes an instance of the `coordinator` role – more precisely a *multi-role* instance which consists of multiple *sub-role* instances [Bec+12] – while RailCabs `r2` and `r3` each execute an instance of the `member` role. The two instances of the RTCP DistanceControl are each used by two adjacent RailCabs to keep a constant driving distance.

The RTSCs implementing this role behavior reside in the reflective operator, i.e., the middle layer of the OCM. They can trigger architectural reconfiguration, e.g., the deinstantiation of a subcomponent that directly manages the driving speed and the simultaneous instantiation of another subcomponent that determines the driving speed based on a reference value provided by the convoy coordinator and the distance to the RailCab in front. However, RTSCs do not decide themselves whether driving in a convoy is useful or not. This is done by the cognitive operator, i.e., the top layer of the OCM. This layer generates temporal plans that contain the information when to instantiate which RTCP. Executing such a generated plan results in specific transitions of RTSCs being triggered via asynchronous messages, which is done precisely at those times given in the plan. Note that these plans do not have to be generated by a single, autonomous RailCab, but can be generated by a cross-linked mechatronic system consisting of several RailCabs, cf. [LHL01], and that RailCabs execute multiple planning processes at different levels of abstraction, cf. [RZ13].

1.5 Thesis Outline

The next two chapters introduce the foundations this thesis builds upon. Chapter 2 covers the two main algebraic approaches to graph transformation considered in this thesis. It includes the notions of *parallel and sequential independence*, which are relevant for deciding whether multiple graph transformations are free of any conflicts. Chapter 3 gives a short introduction to PDDL, focusing on those extensions that are used in this thesis, and explains the notion of *required concurrency* in planning domains.

Chapters 4 to 6 present the main contributions of this thesis. Chapter 4 explains the untimed planning approach, which works directly on the state space of a graph transformation system. The new formalism for the specification of durative graph transformations and their dependencies regarding concurrent and urgent execution is presented in Chapter 5, and its translation into PDDL domain models, which enables the generation of temporal plans, is explained in Chapter 6. In each of the three chapters, related work specific to the chapter's contribution is covered directly within the chapter in a separate section.

Finally, Chapter 7 summarizes the results and concludes with a perspective on future work. Discussions on future work can be found both in discussion sections specific to each of the three chapters (with a narrow perspective) and in the conclusion (with a wider perspective).

2 Background on Graph Transformations

In graph grammars and graph transformation systems, the modification of graphs is specified via graph transformation rules[1]. Each rule consists of a pair of graphs, called *left-hand side (LHS)* and *right-hand side (RHS)*, which schematically define how a graph may be transformed into a new graph. Applying a graph transformation rule to a graph can be seen as replacing a subgraph corresponding to the rule's LHS with a copy of its RHS. More precisely, elements that are specified in both LHS and RHS are preserved by the rule application, elements specified only in the LHS are deleted, and elements specified only in the RHS are created. When a graph transformation rule is applied to a graph, this graph is called *host graph* to not confuse it with the LHS and RHS of the rule, which are also graphs.

Naturally, the possibility of applying a graph transformation rule to a host graph underlies the condition that a subgraph corresponding to the rule's LHS can be found. Furthermore, it is also possible that multiple matching subgraphs exist in a host graph. In such a case, multiple rule applications of the same rule can be performed. These rule applications are not necessarily independent. It might be the case that a choice has to be made at which match to transform the host graph, e.g., when different matches overlap and each their respective graph transformation modifies element contained in the other match.

A set of graph transformation rules together with an initial graph spans a transition system. In this transition system, graphs are represented as states and graph transformations as transitions between states. It is important to realize that the nondeterminism indicated by multiple outgoing transitions of a state has two sources: multiple rules may be applicable to a graph and they may potentially be applied at multiple matches.

There exist various approaches to realize graph transformations. They are broadly classified into *connecting* approaches and *gluing* approaches. The main difference of these approaches is how they attach a new replacement subgraph to the remainder

[1] A graph transformation rule is also known as *graph production*, cf. [Cor+97; Ehr+06].

of the host graph. Connecting approaches introduce new edges to connect the new subgraph to the remainder graph. Gluing approaches identify or "glue together" certain elements of the new subgraph with elements of the remainder graph.

In the *node replacement approach* [JR80; ER97], a graph transformation replaces a single node in a graph with a new subgraph. This subgraph is connected with new edges to the remainder graph according to an embedding relation. There are various ways to define such an embedding relation. Consequently, there are several extensions and variations of this approach. All of these variations belong to the connecting approaches.

In the *hyperedge replacement approach* [Fed71; Pav72; DKH97], a hyperedge is replaced by a new hypergraph. This approach does not require an embedding relation. The new hypergraph is glued to the remainder hypergraph by identifying designated attachment nodes with nodes of the remainder hypergraph. This approach belongs to the group of *gluing* approaches.

There are two notable algebraic approaches, the *double pushout (DPO)*, which was invented by Ehrig et al. [EPS73; Cor+97; Ehr+06] and the *single pushout (SPO) approach*, which was invented by Löwe et al. [Löw93; Ehr+97]. Both approaches are based on category theory and the categorical term of a pushout. In DPO, a transformation is formalized via two pushouts in the category of graphs and (total) graph morphisms. One of the pushouts realizes the deletion of elements and the other one realizes their addition. In SPO, only a single pushout is used, which is a pushout in the category of graphs and partial graph morphisms. Both approaches belong to the gluing approaches.

The two algebraic approaches differ in how they handle certain situations. In DPO, the application of a rule is not allowed at a match if it causes one or more dangling edges. The DPO approach also requires that no element in the host graph may have more than one preimage under the match if any of these preimages is to be deleted. Therefore, a transformation deletes exactly as many elements as specified in the rule. The SPO approach has no such restrictions on the applicability of rules. Dangling edges are simply deleted, and situations where an element in the host graph has multiple preimages under the match, one of them specified to be deleted and the other one to be preserved, are also resolved by deleting the element in question.

The algebraic approaches also have alternative set-theoretic presentations, which are commonly seen in tutorial introductions, e.g. [EKL91; BH02], and include explicit constructions of successor graphs. As for practical matters, their workings and outcome are the same. The successor graphs of the set-theoretic and those of the algebraic versions are equivalent up to isomorphism. We adhere to the algebraic versions, which are deemed more suitable for proofs than the explicit constructions, cf. [Cor+97, p. 187].

Other well-known approaches to graph transformation are Courcelle's *monadic second-order logic of graphs* [Cou90; Cou97], which uses logical formulas to specify graph properties and graph transformations, the *theory of 2-structures* by Ehrenfeucht el al. [EHR97], which is a relational framework for the decomposition and transfor-

mation of graphs, as well as approaches to *programmed graph replacement* [Sch97], which employ control programs to steer the application of graph transformation rules.

The formal semantics of timed and durative graph transformation systems, which is provided in Chapter 5, follows the SPO approach. However, there is no conceptual restriction to the SPO approach; the presented concepts work perfectly well in a DPO context. This is why the translation of durative graph transformation system models into PDDL allows to choose whether to comply with the DPO or SPO semantics.

The next sections present the fundamentals of these two approaches. Section 2.1 lays the algebraic foundation for both approaches. It introduces the notions of graphs, graph morphisms, and pushouts. Sections 2.2 and 2.3 explain the workings of graph transformations in the DPO and SPO approach, respectively. Section 2.4 introduces negative application conditions and illustrates the graphical representation used for graph transformation rules in this thesis. The last section, Section 2.5, introduces the notions of parallel and sequential independence of graph transformations. These notions play a vital role in proving properties of the semantics of durative graph transformation systems.

The definitions provided in this chapter are loosely based on the monograph *Fundamentals of Algebraic Graph Transformation* [Ehr+06] and the first volume of the *Handbook of Graph Grammars and Computation by Graph Transformation*, in particular the chapters on the DPO approach [Cor+97] and the SPO approach [Ehr+97]. The DPO approach primarily follows the formalization provided in [Ehr+06]; the SPO approach follows that provided in [Ehr+97]. The notions of parallel and sequential independence follow that of Habel et al. [HHT96]. We also provide less strict variants of parallel and sequential independence that take advantage of the existence of isomorphic matches. Although the idea of graph rewriting modulo isomorphism is not new, cf. [Plu05], we did not find definitions for parallel and sequential independence modulo isomorphism in related work.

2.1 Graphs, Graph Morphisms, and Pushouts

A graph is a structure that represents a set of objects along with relations between them. Here, we consider only directed graphs. Undirected graphs can be simulated by adding both directed edges for each undirected edge.

Definition 2.1.1 (Graph)**.** A *(directed) graph* $G = (V_G, E_G, src_G, tgt_G)$ consists of a set of nodes V_G, a set of edges E_G, and source and target functions $src_G, tgt_G : E_G \to V_G$.

This definition of a graph allows parallel edges, i.e., edges whose pair of source and target node is identical to the pair of source and target node of another edge. Our semantics of durative graph transformations makes use of parallel edges. Each read access to a node or edge is realized as another (possibly parallel) edge. Multiple concurrent read accesses to the same node or edge thus result in multiple parallel edges. Another common definition of graphs defines the set of edges such that

$E \subseteq V \times V$. Such a definition does not allow parallel edges. However, it can be used to simulate graphs that do support parallel edges, cf. [Bon+07].

Relations between graphs can be expressed through graph morphisms. A graph morphism is a mapping of nodes and edges of one graph to nodes and edges of another graph such that the source and target nodes of edges are preserved. Such morphisms are used in graph transformation rules to define which nodes and edges are created, deleted, or preserved when the rule is applied to a graph.

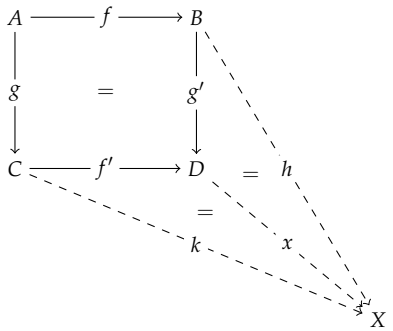
Definition 2.1.2 (Graph morphism, partial graph morphism)**.** A *graph morphism* $f : G \to H$ between two graphs is a pair of mappings $f = (f_E, f_V)$ with $f_E : E_G \to E_H$ and $f_V : V_G \to V_H$ that commutes with the source and target functions, i.e., $f_V \circ src_G = src_H \circ f_E$ and $f_V \circ tgt_G = tgt_H \circ f_E$. A graph morphism $f = (f_E, f_V)$ is called *injective* if f_E and f_V are injective and called *isomorphic* if f_E and f_V are bijective.

A *subgraph* S of G, written $S \subseteq G$ or $S \hookrightarrow G$, is a graph with $V_S \subseteq V_G$ and $E_S \subseteq E_G$ such that $src_S = src_G|_{E_S}$ and $tgt_S = tgt_G|_{E_S}$. A *partial graph morphism* g from G to H is a (total) graph morphism from a subgraph of G to H. This subgraph is called the *restricted domain* of g, written $\mathrm{dom}(g)$. The *range* of a graph morphism $g' : G \to H$, written $\mathrm{ran}(g')$, is a subgraph S' of H where $V_{S'}$ is the image set of g'_V and $E_{S'}$ is the image set of g'_E.

The application of graph transformation rules is based on the concept of "gluing" graphs together. Two different graphs sharing a common subgraph can be glued together by adding the uncommon nodes and edges of both graphs to the common subgraph. This is formalized by the categorical notion of a pushout.

Definition 2.1.3 (Pushout)**.** Let $f : A \to B$ and $g : A \to C$ be two morphisms in a category $\mathbf{C}$. A *pushout* (D, f', g') over f and g is defined by a pushout object D and morphisms $f' : C \to D$ and $g' : B \to D$ such that

- $g' \circ f = f' \circ g$ and (commutativity)
- for all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $h \circ f = k \circ g$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$. (universal property)



Here, A is the common subgraph. The pushout object D is the result of gluing B and C via A, f, and g. The commutativity ensures that all elements of B and C that have a common preimage in A are glued together in D. The universal property ensures that

- elements of B and C that do not have a common preimage in A are not glued together in D and
- D does not contain elements that neither exist in B nor C.

If elements of B and C were glued together in D, then there would exist a graph X for which no morphism $x : D \to X$ satisfies $x \circ g' = h$ and $x \circ f' = k$, because x had to map the glued element simultaneously to different elements in X for $x \circ g' = h$ and $x \circ f' = k$ to hold. If D did contain elements that exist neither in B nor C, there would also exist such a graph, e.g., a subgraph of D that does not contain these elements.

2.2 Double Pushout Approach

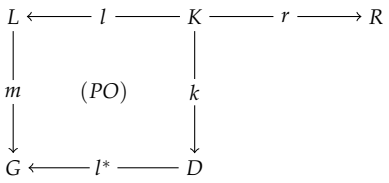
In the double pushout approach, a graph transformation rule connects its LHS and RHS via a so-called *gluing graph*[2], which is a common subgraph of the LHS and RHS. The gluing graph represents those nodes and edges that are preserved during the application of the rule. To identify these nodes and edges in the LHS and RHS, two total graph morphisms are used. Elements of the LHS and RHS that are outside of the range of these morphisms represent those elements that are being deleted and created by the application of the rule, respectively.

Definition 2.2.1 (Graph transformation rule (DPO))**.** A *graph transformation rule* $p = (L, K, R, l, r)$ consists of three graphs L, K, and R, called *left-hand side (LHS)*, *gluing graph*, and *right-hand side (RHS)*, respectively, and two injective graph morphisms $l : K \to L$ and $r : K \to R$.

The semantics of the application of a rule is given by two pushouts in **Graph**, the category of graphs and (total) graph morphisms, cf. [Ehr+06]. The first pushout handles the deletion of nodes and edges, the second pushout their addition. However, whether or not the first pushout can be constructed depends on the host graph and the match of the LHS to the host graph.

Definition 2.2.2 (Applicability of a rule (DPO))**.** A graph transformation rule $p = (L, K, R, l, r)$ is *applicable* at a match $m : L \to G$, if and only if a *context graph* D can be constructed such that there exists a pushout (G, l^*, m) over $l : K \to L$ and $k : K \to D$ in **Graph**.

[2]The gluing graph of a graph transformation rule is also known as *interface*, cf. [Cor+97].



The context graph is unique up to isomorphism if it exists. However, if the deletion of nodes results in the existence of dangling edges, the context graph cannot be constructed. This is because the definition of a graph does not allow any dangling edges. The pushout can also not be constructed if the images of elements in L have been merged by m into the same element in G and at least one of these elements is not going to be preserved. In such a case the universal property of the pushout does not hold.

Unfortunately, this definition makes it rather difficult to see whether a graph transformation rule is applicable at a given match. Fortunately, there exists an equivalent notion of a rule's applicability, called the *gluing condition*, cf. [Ehr+06].

Definition 2.2.3 (Gluing condition (DPO))**.** Let $p = (L, K, R, l, r)$ be a graph transformation rule, G a graph, and $m : L \to G$ a match. Then,

- GP denotes those nodes and edges in L, called *gluing points*, that are not deleted by p, i.e., $GP = l(K)$,

- IP denotes those nodes and edges in L, called *identification points*, whose images under m have been merged into the same element in G, i.e., $IP = \{v \in V_L | \exists w \in V_L, w \neq v : m(v) = m(w)\} \cup \{e \in E_L | \exists f \in E_L, f \neq e : m(e) = m(f)\}$, and

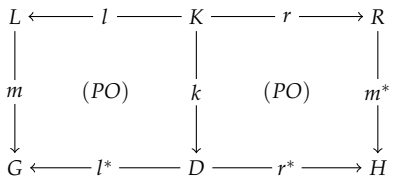
- DP denotes those nodes in L, called *dangling points*, whose images under m are the source or target of an edge in G that is not contained in $m(L)$, i.e., $DP = \{v \in V_L | \exists e \in E_G \setminus m(E_L) : src(e) = m(v) \vee tgt(e) = m(v)\}$.

If all *identification points* and all *dangling points* are also *gluing points*, i.e., $IP \cup DP \subseteq GP$, then p and m satisfy the *gluing condition* (and thus p is applicable at m).

If a DPO graph transformation rule is applicable at a match m, its graph transformation is defined by a double pushout in **Graph**.

Definition 2.2.4 (Graph transformation (DPO))**.** Let $p = (L, K, R, l, r)$ be a graph transformation rule and $m : L \to G$ a match of its LHS L to a graph G such that p is applicable at m. The *(direct) graph transformation*[3] from G to H via p at m, written $G \overset{p,m}{\Longrightarrow} H$, is given by the pushouts (G, l^*, m) and (H, r^*, m^*) in **Graph**.

[3] A (direct) graph transformation is also known as *(direct) derivation*, cf. [Cor+97]



The first pushout results in the construction of a context graph D, which corresponds to a temporary, intermediate graph where all deletion but no creation is performed. Then, the second pushout, which always exists if the first pushout exists, adds new elements of the rule's RHS by gluing them together with the context graph.

2.3 Single Pushout Approach

In the single pushout approach, graph transformation rules are defined by only one morphism. This morphism directly maps from the LHS to the RHS, without the use of a gluing graph. To allow the deletion of elements, this morphism is partial instead of total. Intuitively, elements of the LHS that are outside of the morphism's restricted domain are deleted, and elements of the RHS that are outside of the morphism's range are created.

Definition 2.3.1 (Graph transformation rule (SPO))**.** A *graph transformation rule* $p = (L, R, r)$ consists of two graphs L and R, called *left-hand side (LHS)* and *right-hand side (RHS)*, and an injective partial graph morphism $r : L \to R$, called *rule morphism*.

In an SPO graph transformation rule, the rule morphism specifies both addition and deletion. Therefore, the pushout construction for the SPO approach is more complicated than for the DPO approach. In addition to the concept of gluing, it has to realize deletion.

Deletion is realized in the SPO approach by "equalizing" two partial morphisms that are defined on the same domain of definition but on different restricted domains. This is done by removing all elements from their range that have different preimages under both morphisms. This concept is formalized by the categorical notion of a co-equalizer.

The SPO approach constructs a specific co-equalizer, cf. [Ehr+97]. Its construction assumes that, for each element that is contained in the restricted domains of both morphisms, both morphisms map to the same image. We will see that this is sufficient for the construction of a pushout in $\mathbf{Graph}^{\mathbf{P}}$, the category of graphs and partial graph morphisms, in Definition 2.3.3.

Definition 2.3.2 (Specific co-equalizer in $\mathbf{Graph}^{\mathbf{P}}$)**.** Let $a, b : A \to B$ be two (partial) morphisms such that $\forall x \in \mathrm{dom}(a) \cap \mathrm{dom}(b) : a(x) = b(x)$. The *co-equalizer* of a and b in $\mathbf{Graph}^{\mathbf{P}}$ is the tuple (C, c) where

- $C \subseteq B$ is the largest subgraph of $B \setminus a(\overline{\mathrm{dom}(b)}) \setminus b(\overline{\mathrm{dom}(a)})$ and
- $c : B \to C$, with $\mathrm{dom}(c) = C$, is the identity morphism on C.

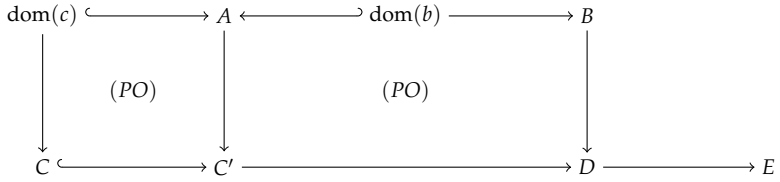
To construct C, the co-equalizer filters out from B all elements for which there is a preimage under one of the morphisms a or b that is not defined under the other morphism. Therefore, only elements remain in C that have either the same preimage(s) under both morphisms or no preimages at all. Dangling edges are deleted because the definition constructs C as the greatest subgraph of $B \setminus a(\overline{\mathrm{dom}(b)}) \setminus b(\overline{\mathrm{dom}(a)})$, which is a graph-like structure containing dangling edges.

The construction of a pushout in $\mathbf{Graph^P}$ is realized via two pushouts in **Graph** and a co-equalizer, cf. [Ehr+97]. The total morphisms for the two pushouts in **Graph** are defined in dependence on the partial morphism of the pushout in $\mathbf{Graph^P}$. The co-equalizer is used to realize deletion in the construction of a pushout in $\mathbf{Graph^P}$.

Definition 2.3.3 (Pushout in $\mathbf{Graph^P}$)**.** Let $b : A \to B$ and $c : A \to C$ be two partial graph morphisms. The *pushout* over b and c in $\mathbf{Graph^P}$ always exists and can be constructed in three steps:

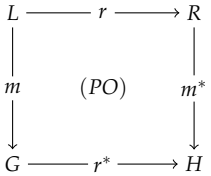
1. Construct the pushout $(C', A \to C', C \to C')$ of the total morphisms $\mathrm{dom}(c) \to C$ and $\mathrm{dom}(c) \to A$ in **Graph**. (gluing 1)
2. Construct the pushout $(D, B \to D, C' \to D)$ of the total morphisms $\mathrm{dom}(b) \to A \to C'$ and $\mathrm{dom}(b) \to B$ in **Graph**. (gluing 2)
3. Construct the co-equalizer $(E, D \to E)$ of the partial morphisms $A \to B \to D$ and $A \to C \to C' \to D$ in $\mathbf{Graph^P}$. (deletion)

The pushout over b and c in $\mathbf{Graph^P}$ is the tuple $(E, C \to C' \to D \to E, B \to D \to E)$.



Since the pushout in $\mathbf{Graph^P}$ always exists, there is no counterpart to the gluing condition in SPO. We can simply define the application of an SPO graph transformation rule as a pushout in $\mathbf{Graph^P}$.

Definition 2.3.4 (Graph transformation (SPO))**.** Let $p = (L, R, r)$ be a graph transformation rule and $m : L \to G$ a match of its LHS L to a graph G. The *graph transformation* from G to H via p at m, written as $G \overset{p,m}{\Longrightarrow} H$, is given by the pushout (H, r^*, m^*) over r and m in $\mathbf{Graph^P}$.



The morphisms r^* and m^* are called the *derivation morphism* and the *co-match* of $G \overset{p,m}{\Longrightarrow} H$, respectively.

The match m and the rule morphism r correspond to $A \to C$ and $A \to B$ of Definition 2.3.3, respectively. Since the match of an LHS to a host graph is always total, the first pushout in **Graph** does not do anything. The second pushout in **Graph** adds elements, similar to the second pushout during the application of a DPO rule. Due to the second pushout, $A \to B \to D$ and $A \to C \to C' \to D$ commute, which allows to construct their co-equalizer. At the end, the co-equalizer deletes all elements for which there is a preimage under m that is not defined under r. Regardless of whether or not such an element has another preimage under m that *is* defined under r, the element is deleted. The existence of another preimage is not relevant, see Definition 2.3.2. To end up in a valid graph, the co-equalizer deletes dangling edges as well.

2.4 Negative Application Conditions, Types, and Visual Representation

The visual representation of graph transformation rules used in this thesis follows the *story pattern* formalism [Det+12]. A story pattern represents a graph transformation rule by integrating the LHS and RHS into one graph and using stereotypes to indicate elements that are only present in the LHS or RHS.

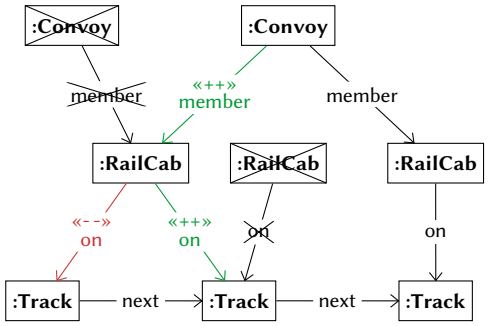


Figure 2.1: An example of a story pattern

Figure 2.1 shows a story pattern from one of the two RailCab domains used in this thesis. The story pattern shows a RailCab joining a convoy of RailCabs. Nodes and edges that are being created by the application of the story pattern, i.e., appear only in the RHS, or deleted, i.e., appear only in the LHS, are labeled with stereotypes «++» and «--» and drawn in green and red, respectively. Elements being created are also referred to as *creation node/edge* and elements being deleted as *deletion node/edge*. This story pattern specifies the creation of a `member` edge representing the RailCab's participation in the convoy operation simultaneously with its movement to the next track segment.

To restrict the applicability of a rule, a *negative application condition (NAC)* can be used. A negative application condition forbids specific graph structures from being present in the host graph. The story pattern formalism also allows to express negative application conditions. In Figure 2.1, the crossed out `Convoy` node and

the `member` edge connecting it to one of the `RailCab` nodes constitute a NAC. The crossed out `RailCab` node and the `on` edge connecting it to a `Track` node constitute another NAC. In principle, each group of adjacent crossed out nodes and edges constitutes a single NAC.

Definition 2.4.1 (Negative application condition)**.** Let $p = (L, R, r)$ be a graph transformation rule, G a graph, and $m : L \to G$ a match. A *negative application condition (NAC)* is a tuple (N, n) where N is a graph and $n : L \to N$ an injective morphism. If there exists no morphism $q : N \to G$ such that $q \circ n = m$, then m *satisfies* (N, n), written $m \models (N, n)$. Given a set of NACs $\mathcal{N}$, if $\forall (N, n) \in \mathcal{N} : m \models (N, n)$, then m satisfies $\mathcal{N}$, written $m \models \mathcal{N}$. For a graph transformation rule $p = (L, R, r)$ with a set of NACs $\mathcal{N}$, we also write $p = (L, R, r, \mathcal{N})$.

The above definition of a NAC follows the SPO approach. A definition for the DPO approach can be made analogously.

We use the terms *forbidden edge* and *forbidden pair* to refer to specific kinds of NACs. A forbidden edge denotes a NAC that consists of a single edge only, i.e., in addition to a subgraph that equals to the LHS of the graph transformation rule. A forbidden pair denotes a NAC that consists of a single node and an edge connecting this node to the LHS. An example for this is the crossed out `Convoy` node and the `member` edge in Figure 2.1. A node within an LHS that is adjacent to a forbidden pair, e.g., the left `RailCab` node in Figure 2.1, is called *connecting node*. In the story pattern formalism, NACs other than forbidden edges and forbidden pairs are usually disallowed, cf. [Det+12].

The formal syntaxes and semantics of timed and durative graph transformation systems build upon *typed graphs*. Note that typed graphs can be simulated via labeled graphs and vice versa, cf. [Ehr+06, pp. 23–24].

Definition 2.4.2 (Typed graph)**.** Let TG be a distinguished graph, called *type graph*. A *typed graph* $G^T = (G, type)$ consists of a graph $G = (V, E, src, tgt)$ and a graph morphism $type : G \to TG$.

To be able to define timed and durative graph transformation rules and represent their matches to state graphs, we also need a concept of morphisms on typed graphs. A typed graph morphism commutes for the source and target function and preserves the types of nodes and edges.

Definition 2.4.3 (Typed graph morphism)**.** A *typed graph morphism* between two typed graphs G_1^T and G_2^T is a partial graph morphism $f : G_1^T \to G_2^T$ such that

- f commutes for the source and target function, i.e., $src_2 \circ f = f \circ src_1$ and $tgt_2 \circ f = f \circ tgt_1$, and
- f preserves types, i.e., $type_2 \circ f = type_1$.

Story patterns can also assign object names to nodes by writing them before the colon stating a node's type. Within the scope of this thesis, object names are used either to

- fix node bindings for matches, in which case they are simply realized as self edges, or

- allow referring to certain nodes of a rule in graphical representations.

Both usages do not require an integration of object names in formal definitions, which is why we refrain from addressing them formally.

The formal semantics of story patterns is based on (typed) graph transformation systems and follows the SPO approach. While the semantics of timed and durative graph transformation systems is also based on the SPO approach, most of the presented concepts also work in the DPO approach. The semantics of story patterns further employs injective matching. Nevertheless, since most of the presented concepts in this thesis also work fine with non-injective matching, we employ non-injective matching unless otherwise noted.

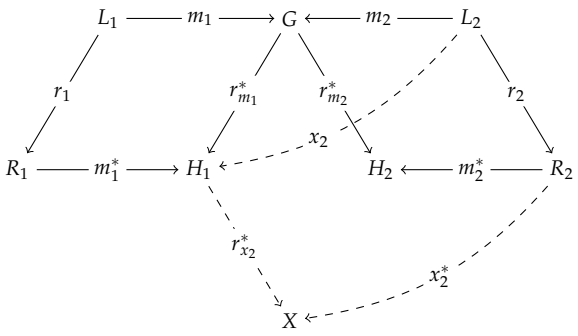
2.5 Parallel and Sequential Independence

When considering two different graph transformations that can be applied in the same configuration, a relevant question might be whether or not both transformations may be executed one after another. If the execution of one of these graph transformation leads to a configuration where the other graph transformation cannot be applied, there seems to be some kind of conflict. If we have two graph transformations that can be executed in sequence, can they be executed in any order and still result in the same graph? If not, then there is obviously some kind of causal dependence between these two graph transformations. Being able to deduce whether or not there is such a causal dependence between two graph transformations is crucial for proving the meaningfulness of the semantics of durative graph transformation systems in Chapter 5.

The above questions result in two different notions of independence: *parallel* and *sequential independence*. Intuitively, parallel independence means that each of the graph transformations can be delayed until after the other transformation has been executed. This is the case, if the range of each graph transformation's match is preserved by the other graph transformation. If the range is not preserved, we call this a *delete-use* (or *use-delete*) *conflict*, cf. [LEO06]. When considering NACs, it is also necessary that each graph transformation's NAC is not invalidated by those elements that the other graph transformation created. If a NAC is invalidated, we call this a *produce-forbid* (or *forbid-produce*) *conflict*.

Definition 2.5.1 (Parallel independence)**.** Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ be two graph transformations, $r^*_{m_1}$ the derivation morphism of $G \overset{p_1,m_1}{\Longrightarrow} H_1$, and $p_2 = (L_2, R_2, r_2, \mathcal{N}_2)$ the graph transformation rule of $G \overset{p_2,m_2}{\Longrightarrow} H_2$.

1. $G \overset{p_2,m_2}{\Longrightarrow} H_2$ is *weakly parallel independent* of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ if there exists a total morphism $x_2 : L_2 \to H_1$ such that $x_2 = r^*_{m_1} \circ m_2$ and $x_2 \models \mathcal{N}_2$.

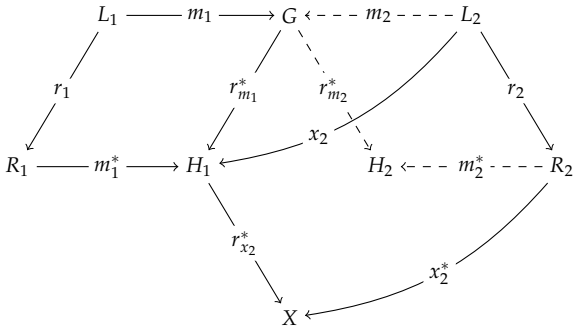


2. $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are *parallel independent* if $G \overset{p_1,m_1}{\Longrightarrow} H_1$ is weakly parallel independent of $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and vice versa.

The concept of sequential independence is analogue to that of parallel independence. The difference is that we assume a consecutive execution of the two graph transformations as a starting point. Sequential independence of two graph transformations means that the graph transformations may be reordered. For this to be possible, the second transformation is not allowed to match elements that have been created by the first transformation. Furthermore, the second transformation is not allowed to delete elements that the first transformation preserves. When considering NACs, the second transformation is also not allowed to depend on a delete operation of the first transformation or create elements that invalidate a NAC of the first transformation.

Definition 2.5.2 (Sequential independence)**.** Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ be two graph transformations, $r^*_{m_1}$ the derivation morphism of $G \overset{p_1,m_1}{\Longrightarrow} H_1$, and $p_2 = (L_2, R_2, r_2, \mathcal{N}_2)$ the graph transformation rule of $H_1 \overset{p_2,x_2}{\Longrightarrow} X$.

1. $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ is *weakly sequentially independent* of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ if there exists a total morphism $m_2 : L_2 \to G$ such that $r^*_{m_1} \circ m_2 = x_2$ and $m_2 \models \mathcal{N}_2$.



2. $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ are *sequentially independent* if $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ is weakly sequentially independent of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_1,m_1}{\Longrightarrow} H_1$ is weakly parallel independent of $G \overset{p_2,m_2}{\Longrightarrow} H_2$.

Since the identities of nodes and edges are rarely relevant for the purposes that graph transformations are employed, isomorphic graphs are usually considered as equal. The semantics we present in Chapter 5 also considers isomorphic graphs are equal. However, the above definitions of parallel and sequential independence do not meet these expectations. The definitions are too strict: they disallow the deletion of elements that are preserved by a second graph transformation, although isomorphic elements (and thus an isomorphic match for the second graph transformation) might exist. Therefore, we replace parallel and sequential independence with parallel and sequential independence *modulo isomorphism*.

The main difference between parallel independence and parallel independence modulo isomorphism is that in the former, we construct x_2 directly in dependence of m_2, whereas in the latter, we construct x_2 in dependence of a match that is isomorphic to m_2.

Definition 2.5.3 (Parallel independence modulo isomorphism)**.** Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ be two graph transformations, $r^*_{m_1}$ the derivation morphism of $G \overset{p_1,m_1}{\Longrightarrow} H_1$, and $p_2 = (L_2, R_2, r_2, \mathcal{N}_2)$ the graph transformation rule of $G \overset{p_2,m_2}{\Longrightarrow} H_2$.

1. $G \overset{p_2,m_2}{\Longrightarrow} H_2$ is *weakly parallel independent modulo isomorphism* of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ if
   - there exists a total morphism $x_2 : L_2 \to H_1$ such that $x_2 = r^*_{m_1} \circ m_2$ and $x_2 \models \mathcal{N}_2$ or
   - there exist total morphisms $\widetilde{m}_2 : L_2 \to G$ and $x_2 : L_2 \to H_1$ such that $\widetilde{m}_2$ is isomorphic to m_2 and $x_2 = r^*_{m_1} \circ \widetilde{m}_2$ and $x_2 \models \mathcal{N}_2$.
2. $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are *parallel independent modulo isomorphism* if $G \overset{p_1,m_1}{\Longrightarrow} H_1$ is weakly parallel independent modulo isomorphism of $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and vice versa.

The difference between sequential independence and sequential independence modulo isomorphism is similar to the difference between parallel independence and parallel independence modulo isomorphism. Instead of constructing m_2 directly in dependence of x_2, we construct m_2 in dependence of a match that is isomorphic to x_2.

Definition 2.5.4 (Sequential independence modulo isomorphism)**.** Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ be two graph transformations, $r^*_{m_1}$ the derivation morphism of $G \overset{p_1,m_1}{\Longrightarrow} H_1$, and $p_2 = (L_2, R_2, r_2, \mathcal{N}_2)$ the graph transformation rule of $H_1 \overset{p_2,x_2}{\Longrightarrow} X$.

1. $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ is *weakly sequentially independent modulo isomorphism* of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ if
   - there exists a total morphism $m_2 : L_2 \to G$ such that $r^*_{m_1} \circ m_2 = x_2$ and $m_2 \models \mathcal{N}_2$ or
   - there exist total morphisms $\widetilde{x}_2 : L_2 \to H_2$ and $m_2 : L_2 \to G$ such that $\widetilde{x}_2$ is isomorphic to x_2 and $r^*_{m_1} \circ m_2 = \widetilde{x}_2$ and $m_2 \models \mathcal{N}_2$.

2. $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ are *sequentially independent modulo isomorphism* if $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ is weakly sequentially independent modulo isomorphism of $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_1,m_1}{\Longrightarrow} H_1$ is weakly parallel independent modulo isomorphism of $G \overset{p_2,m_2}{\Longrightarrow} H_2$.

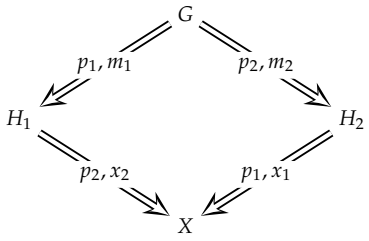
Parallel and sequential independence are symmetric concepts. If we delay one of two parallel independent graph transformations until after the other transformation has been executed, we obtain two sequentially independent graph transformations. We can also consider two sequentially independent graph transformations as two parallel independent graph transformations by applying both to the host graph of the first transformation or reorder them and still end up in the same graph. This is formalized as the Local Church-Rosser Theorem, cf. [HHT96; Ehr+97]. We use this theorem to prove the correct behavior of durative graph transformation rules in Chapter 5.

Theorem 2.5.1 (Local Church-Rosser)**.** *Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ be two parallel independent graph transformations (modulo isomorphism). Then, there exist a graph X and*

- *a graph transformation $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ are sequentially independent (modulo isomorphism) and*
- *a graph transformation $H_2 \overset{p_1,x_1}{\Longrightarrow} X$ such that $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and $H_2 \overset{p_1,x_1}{\Longrightarrow} X$ are sequentially independent (modulo isomorphism).*

Let $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ be two sequentially independent graph transformations (modulo isomorphism). Then, there exist a graph H_2 and

- *a graph transformation $G \overset{p_2,m_2}{\Longrightarrow} H_2$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent (modulo isomorphism) and*
- *graph transformations $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and $H_2 \overset{p_1,x_1}{\Longrightarrow} X$ such that $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,x_1}{\Longrightarrow} X$ are sequentially independent (modulo isomorphism).*



A bijective correspondence between the parallel independent graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and the sequentially independent graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,x_2}{\Longrightarrow} X$ (as well as $G \overset{p_2,m_2}{\Longrightarrow} H_2$ and $H_2 \overset{p_1,x_1}{\Longrightarrow} X$) is given by $x_2 = r^*_{m_1} \circ m_2$ (and $x_1 = r^*_{m_2} \circ m_1$, respectively).

3 Background on AI Planning

Planning is a central part of artificial intelligence research and has been an active research field for several decades. It is a cognitive ability, which is customary performed by people for making decisions in their daily lives. The key point of planning is the analysis of alternatives and reasoning about their consequences. Due to its great relevance for many sciences and businesses, interest in automated and semi-automated planning methods has been increasing lately.

As a result of the many different requirements and application areas for planning tasks, e.g., robotics, manufacturing, logistics, or business process management, there is a vast number of different computer-aided planning techniques available today. In general, planning techniques differ in whether actions are executed deterministically or non-deterministically, the environment is totally or partially observable, there is only a single agent or multiple agents, actions have durations or not, what kind of state representations and state changes are permitted, and what the objective of a planning process is. Classical planning techniques usually consider a single agent with deterministic actions in a totally observable environment without any concept of time or concurrency.

Most languages for describing planning tasks have an action-centric perspective. Depending on the requirements for the planning tasks, there are different planning languages available from academia. Classical planning tasks can be specified in the *Planning Domain Definition Language (PDDL)* [MA98], which is the de facto standard for International Planning Competitions (IPCs) organized by the ICAPS conference series. The same goes for temporal planning problems; they are supported since version 2.1 of PDDL [FL03].

For probabilistic planning, there exists a variant of PDDL called *Probabilistic PDDL (PPDDL)* [YL04]. This variant extends PDDL 2.1 with probabilistic effects and rewards so that planning methods based on *Markov Decision Processes (MDPs)* can be realized. A successor of PPDDL, which also supports *Partially Observable Markov Decision Processes (POMDPs)*, is the *Relational Dynamic influence Diagram Language*

(RDDL) [San10]. It is the planning language currently used in probabilistic tracks of IPCs.

For distributed and multi-agent planning, multiple planning languages have been proposed, some of them extensions to PDDL. The most recent and promising one is *Multi-Agent PDDL (MA-PDDL)* [Kov12]. It supports both planning *for* and planning *by* multiple agents, was used in 2015 during a multi-agent planning competition organized by the ICAPS Workshop on Distributed and Multi-Agent Planning (DMAP 2015), and is expected to become the input language for possible multi-agent planning tracks on future IPCs.

For arbitrary planning problems in the propositional STRIPS formalism, deciding the existence of a plan is PSPACE-complete, cf. [Byl94]. If actions are only allowed to add but not to delete atomic formulas, it is NP-complete. Fortunately, in many transportation domains (where the consumption of fuel is not constrained), a satisfying plan can be found in polynomial time, cf. [Hel14]. However, finding a satisfying plan if fuel is restricted and finding an optimal plan both are NP-complete.

3.1 PDDL Fundamentals

In PDDL, a planning task is separated into a domain and a problem description. A domain description characterizes the mechanics of a domain, i.e., it defines (parameterized) operations, called *action schemata*, as well as object types and predicates, which are used within those action schemata. Here, a predicate means the set-theoretic meaning of a predicate, i.e., a Boolean-valued function. As usual, the term predicate refers to a predicate symbol (or name), and the term literal refers to an atomic formula, which is a predicate symbol together with a list of arguments, or its negation.

Problem instance information, like specific objects available in the planning task's "world", are defined in problem descriptions. Such a problem description also defines an initial state, which is represented as a set of ground atomic formulas, and a goal specification. Multiple problem descriptions can be associated with the same domain description, thus yielding different planning tasks on the same application domain. Note that states follow the *closed-world assumption*: any atomic formula that is not known to be *true* in a state is indeed *false*.

An action schema within a domain description consists of a list of parameters, a precondition, and an effect. In the precondition, a list of literals that are required for applying the action can be specified. Similarly, the effect of an action specifies a list of literals that are obtained when the action is applied. An action schema is instantiated – in the context of PDDL, this is called *grounding* – by substituting the list of parameters with objects defined in the problem description. Since the arguments of all literals are contained in the list of parameters, this transforms the literals into ground literals, which do not contain any free variables.

Examples for a domain and an associated problem description are given in Listings 3.1 and 3.2. According to the domain description, vehicles can move between locations by consuming fuel. To capture this possibility, the domain declares the

Listing 3.1: An example domain desciption in PDDL [FL03]

```
1:  (define (domain vehicle)
2:    (:requirements :strips :typing)
3:    (:types vehicle location fuel-level)
4:    (:predicates
5:      (at ?v - vehicle ?p - location)
6:      (fuel ?v - vehicle ?f - fuel-level)
7:      (accessible ?v - vehicle ?p1 ?p2 - location)
8:      (next ?f1 ?f2 - fuel-level)
9:    )
10:   (:action drive
11:     :parameters (?v - vehicle ?from ?to - location ?fbefore ?fafter -
                ↪    fuel-level)
12:     :precondition (and
13:       (at ?v ?from)
14:       (accessible ?v ?from ?to)
15:       (fuel ?v ?fbefore)
16:       (next ?fbefore ?fafter)
17:     )
18:     :effect (and
19:       (not (at ?v ?from))
20:       (at ?v ?to)
21:       (not (fuel ?v ?fbefore))
22:       (fuel ?v ?fafter)
23:     )
24:   )
25: )
```

types `vehicle`, `location`, and `fuel-level` and four predicates that state at which location each vehicle is (line 5), which fuel level each vehicle has (line 6), which location is accessible from which other location for each vehicle (line 7), and which fuel level follows which fuel level after fuel has been consumed (line 8).

The action schema's precondition ensures that the vehicle is at the starting position assumed by the ground action (line 13), the end position is accessible by the vehicle from the starting position (line 14), the vehicle has the fuel level assumed by the ground action (line 15), and a next lower fuel level exists (line 16). Its effect changes the vehicles position to the end location (lines 19 and 20) and reduces the fuel level of the vehicle (lines 21 and 22). Note that variable names always start with a question mark, and parameters of predicates or actions are denoted by variable names followed by their type.

The given problem description defines two vehicles, three fuel levels, and four locations as available objects (lines 4 to 6). Its initial state defines dynamic state information, like the positions of vehicles and their fuel level (lines 9 to 12), but

Listing 3.2: A problem description to the domain of Listing 3.1 [FL03]

```
1:  (define (problem vehicle-example)
2:    (:domain vehicle)
3:    (:objects
4:      truck car - vehicle
5:      full half empty - fuel-level
6:      Paris Berlin Rome Madrid - location
7:    )
8:    (:init
9:      (at truck Rome)
10:     (at car Paris)
11:     (fuel truck half)
12:     (fuel car full)
13:     (next full half)
14:     (next half empty)
15:     (accessible car Paris Berlin)
16:     (accessible car Berlin Rome)
17:     (accessible car Rome Madrid)
18:     (accessible truck Rome Paris)
19:     (accessible truck Rome Berlin)
20:     (accessible truck Berlin Paris)
21:   )
22:   (:goal (and
23:     (at truck Paris)
24:     (at car Rome)
25:   ))
26: )
```

also static problem instance information, like the connections between different fuel levels and locations (lines 13 to 20).

PDDL provides several extensions to the core functionality explained above, which essentially constitute the STRIPS formalism. The only extension already used in the examples shown above is *typing*, which allows to use a type hierarchy for objects. This extension is supported by every relevant PDDL-based planning system today. Unfortunately, other extensions are not supported universally; their support depends on the employed planning system. For example, the extension *negative preconditions* enables the use of negative literals in an action's precondition, which is a very useful feature in domain modeling. Another extension, which is made use of in Chapter 6, is *equality*. It allows to use the equal sign as a predicate that is interpreted as equality. Disjunctions and quantifiers can be used in preconditions and goals via the extensions *disjunctive preconditions* and *quantified preconditions*, respectively. Universally quantified and conditional effects can both be supported via the extension *conditional effects*.

3.2 Numeric Expressions

To support planning domains involving non-binary resources, version 2.1 of PDDL introduced numeric expressions. Numeric expressions use numeric-valued functions to associate values with objects of the domain. The declaration of these functions works analogously to that of predicates: it requires only a function name and a list of argument types. The support for numeric expressions can be enabled via the extension *fluents*.

The value of a numeric-valued function for a specific list of arguments constitutes a primitive numeric expression. Values are not restricted or distinguished in what they represent; they can represent quantities of resources, counters, indices, or some dimension of utility. Using arithmetic operators, a numeric expression can be constructed from several primitive numeric expressions.

Numeric expressions are only allowed to appear as part of *numeric facts* or *numeric assignments*. A numeric fact can be used to compare the values of two numeric expressions in an action's condition or the condition of a conditional effect. A numeric assignment can be used in the effect of an action to assign a new value to a primitive numeric expression. Numeric expressions are not allowed in action parameters or as arguments to literals or other numeric expressions.

Listing 3.3: A domain description with numeric expressions [FL03]

```
1:  (define (domain jug-pouring)
2:    (:requirements :typing :fluents)
3:    (:types jug)
4:    (:functions
5:      (amount ?j - jug)
6:      (capacity ?j - jug)
7:    )
8:    (:action pour
9:      :parameters (?jug1 ?jug2 - jug)
10:     :precondition (>= (- (capacity ?jug2) (amount ?jug2)) (amount ?jug1))
11:     :effect (and
12:       (assign (amount ?jug1) 0)
13:       (increase (amount ?jug2) (amount ?jug1))
14:     )
15:   )
16: )
```

An example using numeric expressions is given in Listing 3.3. The domain models an action of the jugs-and-water problem, where jugs of different sizes are available, and the goal is to achieve a certain filling level for each jug. There are two numeric-valued function in this domain: a function that yields the filling level of each jug (line 5) as well as a function that yields their holding capacity (line 6). The modeled action allows to pour the water contained in one jug into a second jug under the condition that the second jug has enough empty space left to hold the

additional water. The precondition specifies this condition by use of a numeric fact calculating the empty space of the second jug and comparing it with the amount of water in the first (line 10). The effect specifies two numeric assignments: the first is an absolute assignment that empties the first jug (line 12); the second is a relative assignment increasing the amount of water in the second jug by that of the first jug (line 13).

Along with numeric expressions came *plan metrics*, which evaluate the quality of plans based on numeric expressions. A plan metric can be provided in the problem description to define an objective for the planning process different from minimizing the number of used actions (in sequential planning) or the timespan of the entire plan (in temporal planning). An example of a plan metric for the vehicles domain is to minimize the amount of fuel used by each vehicle. Obviously, the domain has to specify a suitable function for representing this quantity and update its values each time fuel is consumed. Note that this thesis does not make use of plan metrics other than the built-in metric `total-time`, which refers to the plan's timespan.

3.3 Durative Actions

Like numeric expressions, durative actions have been introduced in version 2.1 of PDDL. There are two kinds of durative actions: *discretized* and *continuous* durative actions. Here, we consider discretized durative actions only.

Durative actions split the literals, numeric facts, and numeric assignments used in each their precondition and effect into different sets according to their time of evaluation. They can be required `at_start`, `over_all`, and `at_end` when used in the precondition and be effective `at_start` and `at_end` when used in the effect. While `at_start` and `at_end` refer to the beginning and ending of an action, `over_all` refers to the (open) interval during the action's execution. As a result, a durative action behaves like two untimed but temporally linked actions with an invariant condition that must be met by all states occurring during their application interval.

Without a notion of time, plans were simply interpreted as sequences of states. With durative actions, the application intervals of actions can overlap, which leads to the question under what constraints they are allowed to do so. To answer this question, we first take a look at the notion of states in this temporal context. States are stretched over intervals, which are separated by points in time on a global clock. State change occurs only at those points in time, and all state change at a given time point occurs instantaneously. The time points where state changes occur are given by the beginnings and endings of durative actions. Multiple beginnings or endings of different actions may fall at the same point in time if their conditions and effects do not interfere. In the semantics of durative actions, such a set of action beginnings and endings is called a *happening* and treated like an ordinary untimed action.

Within a happening, it is not allowed to assert a literal and its negation at the same time. It is also forbidden to assert a literal at the same time it is required by another action's beginning or ending in the same happening. No condition may rely on an effect in the same happening. Even if the condition is true before *and*

after executing a concurrent effect, it may not rely on the value of a literal if the effect updates the value. Fox and Long [FL03] called this the *no moving targets* rule. Note that the beginning or ending of a single action is allowed to access a value in its condition and update it in its effect at the same time. This is only forbidden if performed by different actions, essentially like mutual exclusion for shared variables.

A similar restriction holds for numeric facts and assignments in happenings. The only difference is that multiple simultaneous updates are allowed if they commute, i.e., each of them is a relative assignment.

A consequence of the no moving targets rule is a non-zero separation between each pair of happenings: unlike timed automata and related constructs, where the passing of time is not required between two successive changes of the logical state, two happenings have to occur at least a minimum time of $\varepsilon > 0$ apart from each other. Temporal planning systems often use a value of 0.001 as minimum unit of time.

Durative actions complicate the picture of a planning task's state space in that they involve a commitment. A durative action that has been started in a state, has to be finished at a later point in time. All states up to this point in time have to fulfill the `over_all` conditions of the durative action, and the state at this point in time has to fulfill its `at_end` conditions. However, since the `at_end` condition does not have to be fulfilled when the action starts, it can be achieved by concurrent actions. For this reason, the decision whether a durative action is applicable cannot be made alone by looking at all actions that have been applied already.

Instead of defining the semantics of durative rules in terms of state space construction, Fox and Long [FL03] defined it in terms of executability of happening sequences. Each happening has to fulfill the no moving targets rule, their accumulated action beginnings and endings have to be executable in the order given by the happening sequence, and the state resulting by executing the complete happening sequence has to satisfy the goal specification of the planning task. The happening sequence also contains artificial monitoring actions responsible for checking invariant conditions. These monitoring actions do not contain any effects. They are placed after a durative action with invariant conditions has been started and after each other happening occurring during its application interval. Note that the search through the state space might also consider happening sequences that are not executable, because additional durative actions enabling their executability have not yet been scheduled.

3.4 Required Concurrency

The introduction of durative actions into planning domains added a scheduling problem to the planning tasks. A widely used approach to solve these planning tasks is to separate logical from temporal reasoning and solve the planning and scheduling problems separately. By treating durative actions as single instantaneous actions – this is called *action compression* [LF03] – and thus neglecting any opportunities for the concurrent execution of actions, a plan is computed by employing a classical

sequential planner. Afterwards, actions are scheduled in a post-process to achieve a better plan length. As one might expect, such a pragmatic approach is very fast.

Unfortunately, approaches that separate planning and scheduling are not complete. There are planning problems for which no sequential solution exists. A planning problems where at least one action has to be applied concurrently to another action in order to reach the goal is said to have *required concurrency*. Such a planning problem cannot be solved by a sequential planning system. If a planning problem with required concurrency can be formulated on a planning domain, this domain is said to *support* required concurrency. Note that a planning problem on a domain supporting required concurrency does not automatically have required concurrency itself; it is easily possible to model a planning problem without required concurrency on any domain.

While the term required concurrency was coined by Cushing et al. [Cus+07] in 2007, it has been known before that planning systems solving temporal planning tasks via action compression and sequential planning, like SGPlan [CWH06] or MIPS [Ede03], are fast but not complete. Temporal planning systems that did not perform action compression, like LPGP [LF03] or VHPOP [YS03], were not competitive. For this reason, Halsey et al. [HLF04] developed a planning system that integrates scheduling phases into the planning phase. Its idea is to integrate scheduling phases only where necessary, but postpone scheduling where possible, without sacrificing completeness. Their planner CRIKEY and its successors CRIKEY$_{SHE}$ [Col+09b] and CRIKEY3 [Col+08] detect situations where this integration is necessary by identifying specific patterns in conditions and effects of actions. If such a pattern is found in an action, its ending is considered a choice point for state space exploration; otherwise it is simply applied directly or as soon as it is needed [Col+09a]. CRIKEY3 provides the code base for several state-of-the-art planning systems developed by the Planning Group at King's College London[1], among them the temporal planning system POPF [Col+10], which is used in Chapter 6 for evaluating different planning domains with required concurrency.

[1]The Planning Group of Maria Fox and Derek Long moved from the University of Strathclyde to King's College London in 2011.

4 Planning with Graph Transformations

This chapter presents an approach that allows technical systems to autonomously decide how to reconfigure their software architecture by performing planning tasks on graph transformation systems. Focusing solely on structural aspects, this approach excludes any timing and concurrency issues. This allows to use ordinary (typed) graph transformation rules to model possible reconfigurations of the system.

Classical planning approaches for fully observable and deterministic environments work with notions of state and action: the execution of an action results in a state change. In the case of graph transformation planning, states are represented as graphs, and those actions available in a planning domain result from a set of graph transformation rules. More precisely, a graph transformation rule is a *parameterized* action and graph transformations are *grounded* actions in which elements of the LHS have been substituted with elements from the host graph.

The transition system of a graph transformation system can be constructed by successively applying graph transformations to the initial graph and its successor graphs. The planning task is to find a path in this transition system ending in a graph satisfying a goal specification. The transitions on this path constitute the plan. Since the transition system suffers from a state explosion problem, i.e., a combinatorial blowup of the state space, constructing the complete transition system is not an appropriate option to find a plan. To find a plan efficiently, we need a suitable planning technique.

Planning with graph transformations has been covered before, e.g., in [EW11] for coordinating behavior in cyber-physical systems and in [TK11] for planning a self-healing process in automotive systems. The planning problems are usually solved by one of the following two approaches: either a translation into a dedicated planning language, like PDDL, is performed or a planning system is developed that works directly on a graph transformation system. Unfortunately, both approaches have their drawbacks.

Employing a translation-based approach is tempting because it exploits decades

of research in AI planning by applying state-of-the-art planning systems. However, translation-based approaches suffer from a different expressiveness of GTSs and PDDL: while the creation and deletion of nodes is a fundamental feature of graph transformation systems, there is no such thing in PDDL. By not supporting the instantiation and deinstantiation of objects, PDDL maintains a finite state space. In order to handle the object instantiation and deinstantiation in PDDL nevertheless, a modeling workaround can be used that declares all uninstantiated objects in the initial state, but uses a predicate to state their actual existence. However, the workaround is based on the assumption that a maximal number of objects is known beforehand or can be deduced from the graph transformation system.

Unfortunately, planning systems working directly on the transition system of a graph transformation system are not highly evolved. Up until today, there are only few systems that use domain-independent heuristics to guide their search through the state space of a graph transformation system. Their domain-independent heuristics are rather simple: they compute values for the structural similarity of the current configuration and the goal specification. Multiple such *similarity-based* heuristics are presented in [EJL06]. Several of them are different variants of counting those nodes and edges that have to be created or deleted to reach a target configuration, e.g., they differ in whether or not it is allowed to rely on the identity of nodes and edges, and then using this number as a distance measure. A slightly different variant of such a heuristic has also been presented in [Sni11]. Another idea for a similarity-based heuristics given in [EJL06] is to transform the goal specification into a formula and evaluate how many predicates of this formula are false in a given state. Other graph transformation planning systems not employing domain-independent heuristics, e.g., semi-automatically deducing domain-specific heuristics based on expert knowledge [EW11] or performing entirely different kinds of analyses to guide the search in the state space [HHV11], are explained in more detail in Section 4.5, the related work section of this chapter.

A problem of the aforementioned similarity-based heuristics is that they do not take any graph transformation rules into account. A high similarity between the current state and the goal specification is irrelevant if there are no rules available that can efficiently transform the current state into a state satisfying the goal specification. Therefore, we believe that it is mandatory for an efficient planning system working directly with graph transformations to look into search techniques of state-of-the-art AI planning systems. Adapting already established techniques from modern PDDL-based planners to graph transformation systems might be straightforward in some cases or impossible due to the different expressiveness of graph transformation systems and PDDL in other cases, e.g., due to the possibility of instantiating nodes in graph transformation systems. Furthermore, the process of adapting such techniques might lead to ideas that were impossible or very unintuitive in PDDL-based representations, e.g., merging of nodes. This renders the adaptation of known planning approaches to graph transformation systems an interesting research perspective.

We developed a new planning system working with graph transformations, cf. [Zie14]. It employs a domain-independent heuristic function, which can be used

in combination with different search algorithms. The heuristic function is mainly inspired by the planning system Fast-Forward (FF) [HN01], which is a forward-chaining planner with a heuristic function that uses the solution length of a relaxed problem as heuristic estimate. It won the 2nd International Planning Competition (IPC-2000), which led to a shift of planning research towards heuristic-guided approaches. Variants of its techniques are used in many of today's state-of-the-art planners, e.g., LAMA [RW10a]. In our approach, the relaxation is performed by reinterpreting certain parts of the rules' application conditions. Thanks to this reinterpretation, the relaxed problem is easier to solve than the original problem. As part of this contribution, we compare the performance of our heuristic against the performance of a similarity-based heuristic.

The next section introduces the notion of planning problems on graph transformations systems. The reconfiguration of ECUs serves as a running example for this chapter. Its graph transformation system is presented in Section 4.2 and used in Section 4.3 to explain our heuristic approach. An evaluation comparing the performance of our heuristic against the performance of a similarity-based heuristic is given in Section 4.4. We discuss related work in the narrow area of graph transformation planning in Section 4.5. Then, we conclude this chapter with a discussion on the differences of our heuristic function to that employed by FF and an outlook on further possibilities for graph transformation planners in Section 4.6.

4.1 Problem Statement

To define the graph transformation planning problem, we first need a means to specify a goal. Such a goal specification can, for example, be a graph, which has to be found by the planning system by applying graph transformations to the initial graph. In general, we do not search for the exact graph but for a larger graph that contains the graph we are looking for as a subgraph. We also do not require the same identity of nodes and edges, i.e., we search for a subgraph isomorphism.

Goal specifications also supports NACs. Therefore, a goal specification is sort of like a graph transformation rule without an RHS. We call this a *graph pattern*.

Definition 4.1.1 (Graph pattern)**.** A *graph pattern* $P = (L, \mathcal{N})$ consists of a graph L and a set of NACs $\mathcal{N}$ where each $NAC \in \mathcal{N}$ is a tuple $NAC = (N, n)$ with $n : L \to N$ and n being injective. NAC satisfaction is defined as in Definition 2.4.1.

Having a means of specifying target configurations of a planning problem, we can now define the planning problem itself.

Definition 4.1.2 (Graph transformation planning problem)**.** A *graph transformation planning problem* $\mathcal{P} = (G_0, \mathcal{R}, P_{tgt})$ consists of

- an initial graph G_0,
- a set of graph transformation rules $\mathcal{R}$, and
- a target graph pattern $P_{tgt} = (L_{tgt}, \mathcal{N}_{tgt})$.

The initial graph and the set of graph transformation rules define a graph grammar, i.e., the structure on which a graph transformation planner has to perform its search. The target graph pattern defines the goal for this search. Each path through the state space that ends in a target configuration is a solution to the planning problem, i.e., a *graph transformation plan*.

Definition 4.1.3 (Graph transformation plan)**.** Given a graph transformation planning problem $\mathcal{P} = (G_0, \mathcal{R}, P_{tgt})$ with a target graph pattern $P_{tgt} = (L_{tgt}, \mathcal{N}_{tgt})$, a *graph transformation plan* for $\mathcal{P}$ is a sequence of (direct) graph transformations $G_0 \Rightarrow \ldots \Rightarrow G_k$ such that L_{tgt} has an injective match m in G_k and m satisfies $\mathcal{N}_{tgt}$.

Note that it is possible that multiple plans for a given planning problem exist. One obvious reason for this is that there might be multiple configurations satisfying the target graph pattern. Another possibility is that there are multiple paths to the same target configuration.

Depending on the area of application, the objective might be to find a plan as fast as possible or to also regard a notion of plan quality, e.g., to prefer a short plan rather than a long one.

4.2 Application Example: Reconfiguration of ECUs

This approach to planning with graph transformations is specifically targeted at systems where it is not possible to limit the maximal number of instantiable objects in general. Therefore, we use the reconfiguration of ECUs, which heavily relies on the instantiation and deinstantiation of objects, see Section 1.4, as an application example for this chapter. Remember that this application example addresses the deployment of software components on ECUs and their instantiation at runtime. In this application example, the number of instances is potentially infinite. Furthermore, it involves only few and very compact graph transformation rules, making it suitable for illustrating the workings of the relaxed planning heuristic.

When a self-healing process is triggered by a hardware failure, the current configuration of the system can be used as initial configuration for the planning problem. Consider the graph in Figure 4.1 as the current configuration. There are two ECUs, `n1` and `n2`, and two software components, `c1` and `c2`. For each component there is a component instance running on one of the ECUs. Object names before colons are realized as self edges and used to identify objects unambiguously in goal specifications.

Figure 4.2 shows the graph transformation rules of this application example. In Figure 4.2(a) component data is deployed on an ECU so that the component can be instantiated. Figure 4.2(b) specifies the creation of a component instance. For the rule to be applicable, no other instance of the same component is allowed to run on the same or a different ECU. In general, multiple instances of the same component are allowed and even desirable due to safety reasons, as argued in Section 1.4. Nevertheless, we disallow redundant instances for this rule because the NAC added to disallow redundant instances enables a more thorough explanation

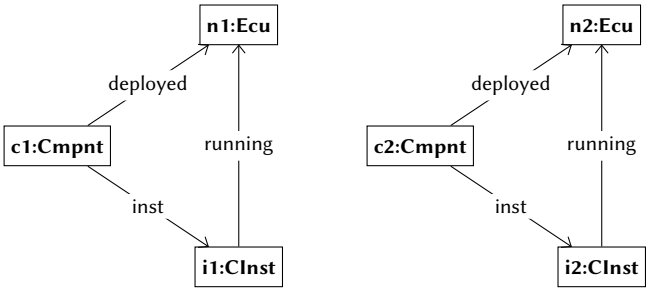


Figure 4.1: An initial configuration for the ECUs domain

of the heuristic function in Section 4.3.2. In Figure 4.2(c) an instance that is running on an ECU is destroyed. At last, Figure 4.2(d) specifies shutting down an ECU if no instances are running on it.

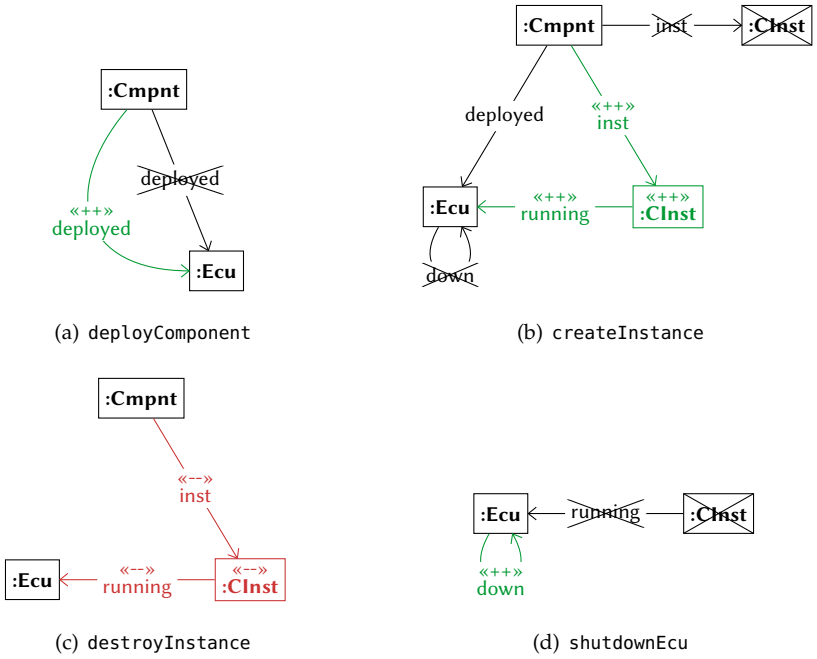


Figure 4.2: Graph transformation rules of the ECUs domain

The goal of the planning problem is specified as a target graph pattern in Figure 4.3. It states that ECU `n1` should be shut down and components `c1` and `c2` should both be instantiated. Since a component instance of `c1` is running on `n1` in the initial state and `n1` is going to be shut down, we added a NAC disallowing component instances of `c1` to run on `n1` in goal states.

An example plan arriving in a target configuration deploys component `c1` at ECU `n2` via the first transformation, then destroys the component instance `i1`, then

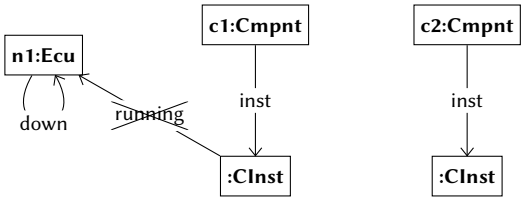


Figure 4.3: A target graph pattern for the ECUs domain

shuts down `n1`, and at last creates a new instance of `c1` that runs on `n2`. This example plan is given in Figure 4.4.

4.3 Relaxed Planning Heuristic

Our approach to solve the graph transformation planning problem is an informed search in the state space of the graph transformation system. Since this approach uses the same graph transformation system for specification and planning, it preserves the expressiveness of graph transformation systems and allows for an integration of the planning process into the state space generation. As a consequence, it supports graph transformations systems that specify infinite systems, which is a significant advantage over translation-based approaches.

Informed searches are usually driven by some kind of evaluation function, which determines which state to expand next. An important part of such an evaluation function is a heuristic function. A heuristic function estimates the costs of the shortest or cheapest path of a given state to a goal state. Heuristic functions can be employed by *global searches*, which systematically explore different paths and keep track of which states have been explored so far, as well as *local searches*, which consider only neighbored states of a current state when deciding which state to expand next.

The main feature of our system is a heuristic function that itself solves a planning problem. This planning problem is a relaxation of the original problem considered from a state given by the search algorithm calling the heuristic function. Information gained during solving the relaxed problem is used to guide the search of the original problem.

The choice of search algorithm employing the heuristic function is not restricted by this approach. We employed both a global and a local search algorithm, i.e., Greedy Best-First (GBF) and Enforced Hill-Climbing (EHC). Both are well-known and thus not considered a part of this contribution. They are briefly explained in Section 4.4, the evaluation section of this chapter.

4.3.1 Abstract State Sequences

In order to compute a plan like the one given in Figure 4.4, the planning system employs a heuristic function that solves a relaxation of the original problem. Such a relaxed problem is solved for each state in the state space that is encountered by

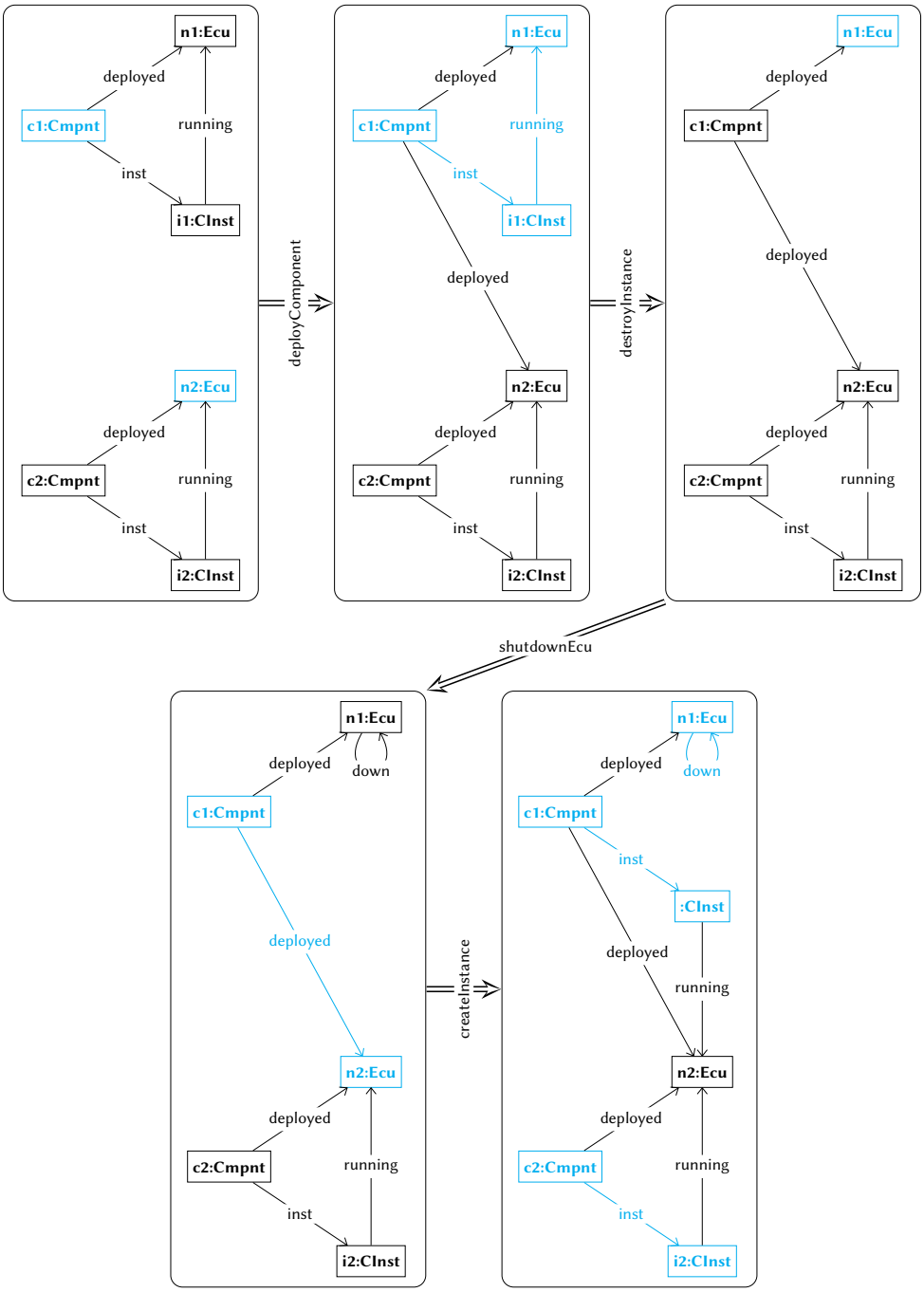


Figure 4.4: An example plan in the ECUs domain. Elements colored in blue indicate those elements contained in the match of the succeeding graph transformation, or in case of the last state, the goal match.

the employed search algorithm. Since, in general, this are a lot of relaxed problems, it is crucial that they can be solved efficiently. In order to do this, the heuristic function combines two ideas. The first idea is to *relax* the applicability of a rule, i.e., to apply only changes that *enable* succeeding rule applications, but discard changes that *disable* succeeding rule applications. In essence, this is realized by not deleting elements when rules are applied, which relaxes LHS matching because succeeding matches are more likely, and the use of markings, which allows to reinterpret and thus relax NAC matching. The second idea is to apply all applicable transformations *in parallel* to construct the next (abstract) state, instead of choosing one state to expand next. This results in a linear (abstract) state space and thus allows to solve the relaxed problem efficiently. Two applicable transformations cannot be in conflict with each other due to the applied relaxation.

Figure 4.5 shows the abstract state sequence from the initial state of the problem to the first state that satisfies the target graph pattern. Each transition corresponds to one parallel and relaxed execution of all applicable transformations. In the first transition, all graph transformations deploying components are executed in parallel as well as all graph transformations destroying component instances. In the second transition, new instances are created and both ECUs are shutdown. Note that object names of newly created instances do not actually exist as self edges in states. The object names of component instances `i3` to `i6` exist solely to allow for an unambiguous reference in tables and writing.

The idea of applying the relaxation is to discard changes that disable succeeding rule applications. Because LHS matching profits from the creation of elements whereas NAC matching profits from their deletion, the tricky part is how to relax LHS matching and NAC matching at the same time. Roughly speaking, this is done by not carrying out the deletion of elements and reinterpreting certain parts of the rules' application conditions. Each element that is supposed to be deleted according to the rule morphism of an applicable transformation is maintained in the abstract successor graph, but marked as *deleted* instead. Each element that is supposed to be created is added to the abstract successor graph as usual but also marked as *created*. These markings give the option to disregard their elements when considering the applicability of transformations in later iterations. This has happened in the second transition of Figure 4.5 with the component instances `i1` and `i2`: since they have been marked as *deleted* by the first transition, new instances can be created during the second transition by applying the `createInstance` rule. In general terms, in order for a transformation to be considered applicable (despite a matching NAC), there has to be at least one element that is marked as *deleted* or *created* in the part of the host graph that is matched by the NAC. If there are multiple NACs or a NAC has multiple matches, then each NAC match has to contain at least one element that is marked as *deleted* or *created* for the transformation to be applicable.

The generation of the abstract state sequence stops as soon as it reached a state that satisfies the target graph pattern. However, it is also possible that there is no path from the initial abstract state to a state that satisfies the target graph pattern. Therefore, we also abort the generation of the abstract state sequence after we

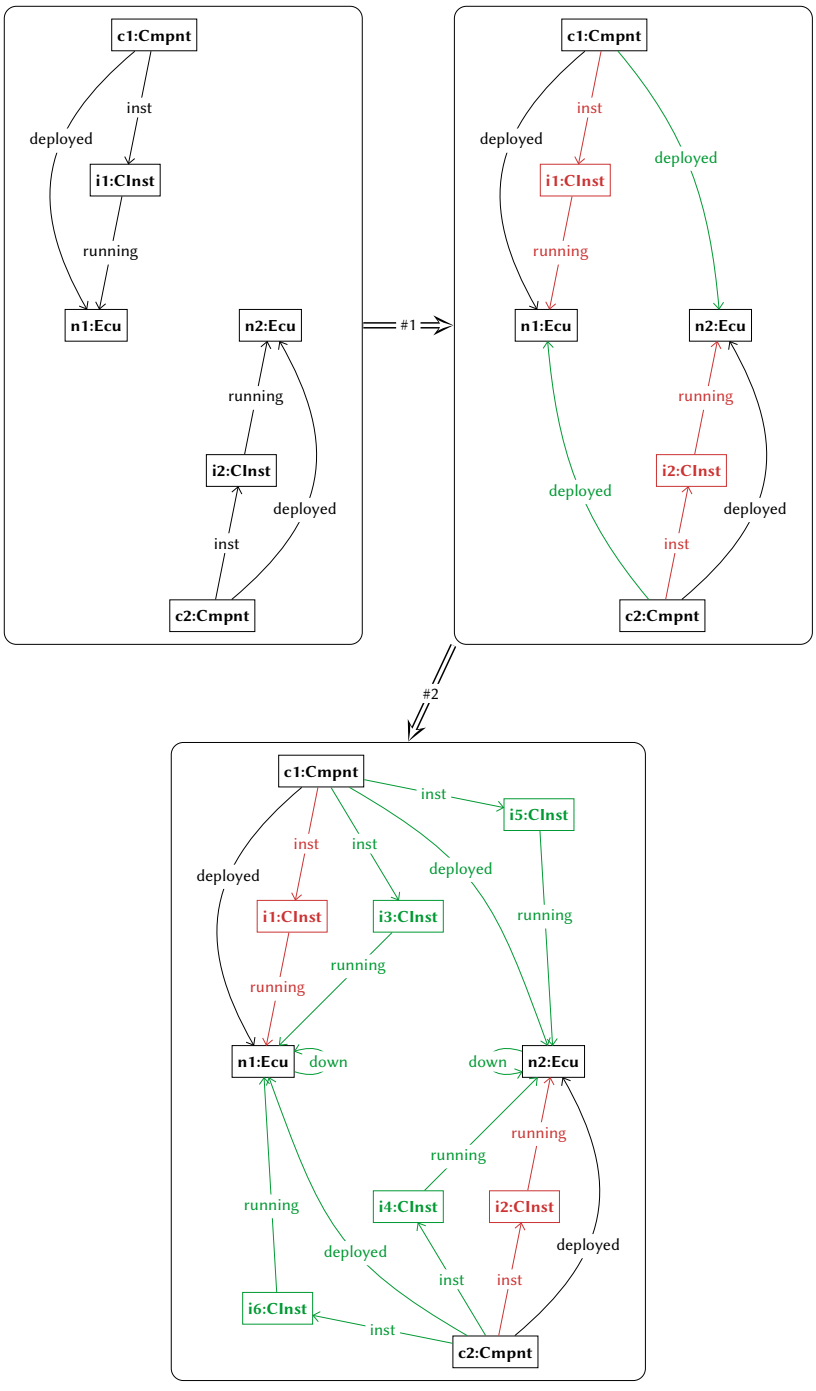


Figure 4.5: An abstract state sequence ending in a state satisfying the target graph pattern of Figure 4.3. Elements colored in red and green indicate those elements marked as *deleted* and *created,* respectively.

Table 4.1: Rule application labels of the first abstract successor state

Element	*Attached labels*
i1	‹#1, 'destroyInst.', #1›
i2	‹#1, 'destroyInst.', #2›
inst(c1,i1)	‹#1, 'destroyInst.', #1›
inst(c2,i2)	‹#1, 'destroyInst.', #2›
running(i1,n1)	‹#1, 'destroyInst.', #1›
running(i2,n2)	‹#1, 'destroyInst.', #2›
deployed(c1,n2)	‹#1, 'deployComp.', #1›
deployed(c2,n1)	‹#1, 'deployComp.', #2›

generated x successor states, where x is two times the heuristic value of the initial state of the concrete planning problem. This ensures that the computation of a heuristic value terminates and that the search algorithm can continue with other states if the relaxed problem is not solvable from a certain state.

4.3.2 Rule Application Labels

Our planning system performs a state space exploration by successively choosing a state and expanding it, i.e., applying each rule at each possible match to generate its successor states. To decide which state to expand next, the system calculates a heuristic value for each unexpanded state. Calculating a heuristic value for a state involves generating the abstract state sequence starting in this state until we reach a state that satisfies the target graph pattern.

Having constructed the abstract state sequence, a naive idea would be to use its length as heuristic estimate. Although the abstract state sequence is expected to be shorter for states which are near to a goal state and longer for states which are further away from a goal state, this value is still rather imprecise. A better idea is to give the approximate number of *individual* graph transformations needed for reaching the (abstract) goal state. However, we cannot simply count all applied transformations per transition to calculate this number, because this would include a lot of transformations that were *not* needed to reach the (abstract) goal state. The transformations that *were* needed to reach the (abstract) goal state are called a *relaxed plan* and their number is called the *length* of the relaxed plan.

Our approach to calculate this number incorporates rule application information into the newly created elements of each successor graph. Each created element is labeled with information about the transformation that caused its creation. This label consists of the iteration number of the successor graph creation loop, the name of the applied rule, and a distinct identifier for the match of the rule to the host graph. As an example, the `deployed` edge from component `c1` to ECU `n2` is labeled with *‹iteration #1, 'deployComponent', match #1›*, see Table 4.1 and the second state in Figure 4.5.

When the goal match is found, we can count the number of distinct rule appli-

Table 4.2: Rule application labels of the second abstract successor state

Element	*Directly attached labels*	*Propagated labels*
i1	‹#1, 'destroyInst.', #1›	
i2	‹#1, 'destroyInst.', #2›	
inst(c1,i1)	‹#1, 'destroyInst.', #1›	
inst(c2,i2)	‹#1, 'destroyInst.', #2›	
running(i1,n1)	‹#1, 'destroyInst.', #1›	
running(i2,n2)	‹#1, 'destroyInst.', #2›	
deployed(c1,n2)	‹#1, 'deployComp.', #1›	
deployed(c2,n1)	‹#1, 'deployComp.', #2›	
i3	‹#2, 'createInst.', #1›	‹#1, 'destroyInst.', #1›
i4	‹#2, 'createInst.', #2›	‹#1, 'destroyInst.', #2›
i5	‹#2, 'createInst.', #3›	‹#1, 'deployComp.', #1›
i6	‹#2, 'createInst.', #4›	‹#1, 'deployComp.', #2›
inst(c1,i3)	‹#2, 'createInst.', #1›	‹#1, 'destroyInst.', #1›
inst(c2,i4)	‹#2, 'createInst.', #2›	‹#1, 'destroyInst.', #2›
inst(c1,i5)	‹#2, 'createInst.', #3›	‹#1, 'deployComp.', #1›
inst(c2,i6)	‹#2, 'createInst.', #4›	‹#1, 'deployComp.', #2›
running(i3,n1)	‹#2, 'createInst.', #1›	‹#1, 'destroyInst.', #1›
running(i4,n2)	‹#2, 'createInst.', #2›	‹#1, 'destroyInst.', #2›
running(i5,n2)	‹#2, 'createInst.', #3›	‹#1, 'deployComp.', #1›
running(i6,n1)	‹#2, 'createInst.', #4›	‹#1, 'deployComp.', #2›
down(n1,n1)	‹#2, 'shutdownEcu', #1›	‹#1, 'destroyInst.', #1›
down(n2,n2)	‹#2, 'shutdownEcu', #2›	‹#1, 'destroyInst.', #2›

cation labels that are contained in the elements of the goal match. This number is the number of transformations needed to create the elements in the goal match. However, these labels contains only labels of elements that appear *directly* in the goal match. It does not yet contain labels of elements that were needed to *arrive* at the goal match. An example for this is the label of the aforementioned `deployed` edge from component `c1` to ECU `n2`. While the `deployed` edge is not contained in the goal match, its creation during the first transition was necessary for the application of *another* transformation during the second transition to create an element in the goal match. In this example, the application of `createInstance` that creates component instance `i5` during the second transition required the `deployed` edge from `c1` to `n2`. Our approach includes labels of such elements when counting the rule application labels in the goal match: each element created by a transformation – in addition to its own label – inherits the labels of all elements in the LHS match of the rule application. In the example above, the rule application label of the `deployed` edge created during the first transition is propagated to the newly created instance `i5` during the second transition, see Table 4.2 and the third state in Figure 4.5.

Elements that have been marked as *deleted* are handled similarly. For example, the component instance `i1` receives the rule application label *‹iteration #1,*

'destroyInstance', match #1› when it is marked as *deleted* by the application of the `destroyInstance` rule, see Table 4.1. Labels of elements being marked as *deleted* are propagated to newly created (or deleted) elements if the labeled element is contained in a NAC match, e.g., the label of `i1` is propagated to the `down` edge at ECU `n1` when `shutdownNode` is applied during the second transition, see Table 4.2. Note that elements being marked as *deleted* inherit labels in the same manner as elements marked as *created*: they inherit labels of created elements if contained in the LHS match and labels of deleted elements if contained in the NAC match. By inheriting the labels of other elements, the elements in the goal match do not only contain labels of transformations that directly created them, but also about all prior transformations that made their creation possible – whether by means of element creation or deletion.

The heuristic value is now simply defined as the number of rule application labels that are attached to all elements in the goal match. This is reasonable because rule application labels have been propagated to elements in the goal match if the referenced rule application assisted in establishing the goal match. In doing so, counting rule application labels follows set semantics, i.e., if elements that were created from different transformations share the same inherited labels, these labels are counted only once. In general, the rule application label set contains at least one label for each iteration of the successor graph creation loop.

Applied to the example of Figure 4.5, the goal match containing `i5` and `i2` results in a heuristic value of 4. This value comes from two rule application labels of `i5` and the two of `n1`'s `down` edge. The rule application label of `i2` is not counted, because `i2` is marked as *deleted*. It would have been counted if `i2` was contained in a NAC. When there exist multiple goal matches present in an abstract state, we use the smaller value. This prevents from basing the heuristic value on a goal match containing `i4` instead of `i2`. If we had not used the NAC for the *running* edge between the component instance of `c1` and `n1` in the target graph pattern, a goal match containing `i1` and thus a heuristic value of 2 would also have been possible. However, since we knew from the initial configuration that `i1` is running on an ECU that is supposed to be shut down, specifying this NAC was reasonable to prevent such an unfavorable goal match from being possible.

4.3.3 Program Code

The heuristic function incorporates two important functionalities. The first functionality is the use of markings during the creation of abstract successor graphs. These markings allow to reinterpret NAC matching such that an element contained in the match of a NAC can be disregarded if it is marked as *created* or *deleted*. As a consequence, the use of these markings relaxes NAC matching and enables – together with the relaxed LHS matching, which results from not deleting elements when transformations are applied – the parallel execution of all applicable transformations. The second functionality is the use of rule application labels and their propagation to newly created or deleted elements. These labels allow to count those

rule applications that assisted in reaching the (abstract) goal state, which constitutes the heuristic value.

Next, we provide program code for this heuristic function. It is divided into three procedures. The first procedure implements the relaxed NAC matching functionality. The second procedure is a simple helper function that collects rule application labels from elements in the LHS match. The third procedure realizes the heuristic function and calls the other two procedures to do so.

Algorithm 4.1: Relaxed NAC matching

Input: Match $m : L \to G$, NACs $\mathcal{N}$, Set *labels*
Output: Boolean *allNacsOk*, Set *labels*

```
1:  procedure checkAllNacMatches(m, 𝒩, labels)
2:      for all q : N → G with (N, n) ∈ 𝒩 and q ∘ n = m do
3:          thisNacOk ← false
4:          for all e ∈ ran(q) with e ∉ ran(m) do
5:              if e is marked as created then
6:                  thisNacOk ← true
7:                  break                          ▷ no need to check other elements
8:              end if
9:              if e is marked as deleted then
10:                 thisNacOk ← true
11:                 insert labels attached to e into labels
12:                 break                          ▷ no need to check other elements
13:             end if
14:         end for
15:         if ¬thisNacOk then
16:             return false                       ▷ no need to check other NAC matches
17:         end if
18:     end for
19:     return true
20: end procedure
```

The first procedure, checkAllNacMatches, is given in Algorithm 4.1. Given an LHS match and the NACs of a graph transformation rule, it checks for each match of a NAC (line 2) whether it contains an element (line 4) that may be disregarded because it has been marked as *created* (line 5) or *deleted* (line 6). If such an element has been found, the current NAC match can be neglected under relaxed NAC matching. This also means that it is not necessary to check any remaining elements in this NAC match (lines 7 and 12). Note that elements that are already contained in the LHS are not considered by this check (line 4), because it is not reasonable to regard them as existing for LHS matching but as not present for NAC matching.

As soon as a single NAC match is found that contains no element marked as either *created* or *deleted*, we know that this NAC is not satisfied, despite the applied relaxation. In such a case, there is no need to check any remaining NACs and

the procedure returns *false* (line 16). If each NAC match can be neglected due to marked elements, the procedure returns *true* (line 19).

As a side effect, this procedure collects rule application labels from those elements that allowed neglecting a NAC match (line 11). This is only done for elements marked as *deleted* but not for elements marked as *created*, because you need to apply a transformation for deleting elements but not for *not* creating them. Note that the set of rule application labels is given as a reference, i.e., there is no need to return this set.

Algorithm 4.2: Collecting rule application labels from an LHS match

Input: Match $m : L \to G$, Set *labels*
Output: Set *labels*

```
1: procedure collectRuleApplicationLabels(m, labels)
2:     for all e ∈ ran(m) do
3:         if e is marked as created then
4:             insert labels attached to e into labels
5:         end if
6:     end for
7: end procedure
```

The second procedure, collectRuleApplicationLabels, is given in Algorithm 4.2. All it does is collect rule application of those elements that are marked as *created*. It is called from the third procedure, either with an LHS match or with a goal match as parameter.

The procedure realizing the heuristic function, computeHeuristicValue, is given in Algorithm 4.3. Given the set of graph transformation rules, an initial graph for the relaxed problem, the target graph pattern, and an upper bound for the length of the abstract state sequence, it constructs successor states until either the most recently created successor state satisfies the target graph pattern or the upper bound is reached (line 4).

The successor graph creation loop is roughly dividable into two parts. The first part (lines 5 to 20) constructs the next abstract successor state. The second part (lines 22 to 29) checks whether it satisfies the target graph pattern.

To construct an abstract successor state, the procedure first searches all LHS matches of all graph transformation rules and checks whether all their NAC matches are satisfied under relaxed NAC matching by calling the procedure checkAllNacMatches (line 9). For all LHS matches that satisfy their NAC matches, new elements are created according to the rule morphism, but none are deleted (line 11). Then, these new elements are marked as *created* (line 12), and elements supposed to be deleted according to the rule morphism are marked as ‹(line 13). Note that each of these elements is only marked if it has not been marked before.

After the marking of elements is completed, the procedure attaches rule application labels to newly marked elements. Labels of elements that enabled the rule application by being marked as *deleted*, i.e., they allowed to neglect one of the

Algorithm 4.3: Heuristic function yielding the length of a relaxed plan

Input: Rules R, Graph G_0, GraphPattern G_{tgt}, Integer $maxLength$
Output: Integer $heuristicValue$

```
1:  procedure computeHeuristicValue(R, G_0, G_tgt, maxLength)
2:      G ← G_0
3:      length ← 0
4:      while length ≤ maxLength do
5:          G_succ ← G
6:          for all p = (L, R, r, N) ∈ R do
7:              for all m : L → G do
8:                  labels ← ∅
9:                  allNacsOk ← checkAllNacMatches(m, N, labels)
10:                 if allNacsOk then
11:                     add created elements of G ⟹^{p,m} H to G_succ
12:                     mark created elements of G ⟹^{p,m} H in G_succ as created
13:                     mark deleted elements of G ⟹^{p,m} H in G_succ as deleted
14:                     collectRuleApplicationLabels(m, labels)
15:                     insert ‹length, p.name, m.id› into labels
16:                     attach labels to newly marked elements in G_succ
17:                 end if
18:             end for
19:         end for
20:         G ← G_succ
21:         length ← length + 1
22:         for all g : L_tgt → G with G_tgt = (L_tgt, N_tgt) do
23:             labels ← ∅
24:             allNacsOk ← checkAllNacMatches(g, N_tgt, labels)
25:             if allNacsOk then
26:                 collectRuleApplicationLabels(g, labels)
27:                 return cardinality of labels
28:             end if
29:         end for
30:     end while
31:     return highest possible value of Integer
32: end procedure
```

NAC matches, are already contained in the set *labels* due to a side effect of the procedure checkAllNacMatches (line 9). Now, the procedure also collects labels from elements that made the LHS match possible, i.e., elements that are marked as *created* and contained in the LHS match, by calling the procedure collectRuleApplicationLabels (line 14). Then, it also puts a label for the rule application that was just executed into the set *labels* (line 15) and attaches this set to all elements that have been marked as *created* or *deleted* by this rule application (line 16). As

a consequence, each element is labeled with both a label for the rule application that created the element and/or its marking as well as labels that made this rule application possible.

After the next abstract successor state has been created, the procedure checks whether this state satisfies the target graph pattern. Like the application of graph transformation rules, this check is performed under relaxed NAC matching (line 24). In case it does satisfy the target graph patten, the procedure collects labels from elements that are marked as *created* and contained in the goal match before returning the size of this set as heuristic value. Labels from elements marked as *deleted* have already been collected when checking relaxed NAC matching.

4.4 Evaluation

We compared the performance of our relaxed planning heuristic (h_{rp}) against that of a similarity-based heuristic (h_{sim}), which resembles those heuristic functions employed by Edelkamp et al. [EJL06] and Snippe [Sni11].

The similarity-based heuristic counts the number of nodes and edges that exist in both the current configuration and the target configuration. It relies on the types of nodes and edges to judge whether a node or edge is counted as existing. More precisely, it puts the type of each node and edge of a configuration into a multiset and takes the cardinality of the intersection of the current configuration's multiset and the target configuration's multiset as a similarity measure. The heuristic value is then defined as the additive inverse of this measure.

Both heuristics have been implemented in GROOVE [Ren04]. To eliminate any potential side effects with a particular search algorithm, each of the heuristics was employed multiple times in combination with a different search algorithm. This evaluation was performed on two different problem domains.

Search algorithms Both heuristic functions were evaluated in combination with greedy best-first and a variant of enforced hill-climbing.

Greedy best-first (GBF) [RN03] is a well-known search algorithm for informed search. It uses a closed list and an open list of states. After expanding a state, i.e., all successor states have been generated, this state is placed in the closed list. For each new successor state found, its heuristic value is computed and then the state is placed in the open list. The decision which state to expand next is solely based on the heuristic values of the states in the open list. The costs to reach the current state are not considered. As a result, the algorithm *greedily chooses* among all known states that state with the smallest expected distance to the goal state.

We also tested a variant of GBF that differs from this approach in that it also *greedily expands* the next state, cf. [CS07, Sect. 3.2]. If a successor state with a better heuristic than the current state is found, this variant immediately chooses this state to expand, without checking any remaining sibling states. When this happens, the current state is not placed in the closed list; it remains in the open list, directly behind the new state. By doing so, the heuristic values of its remaining successor

states can be computed later if the new state turns out to lead to worse successor states. Since there was no significant difference in performance between those two variants of GBF, this section includes only results of the traditional variant.

Enforced hill-climbing (EHC) [HN01] is a local search algorithm. In each iteration it performs a breadth-first search from the current state until it finds a state with a better heuristic value. When such a state is found, it updates the current state and continues with the next iteration. We use a modified EHC that applies best-first search instead of breadth-first search in each iteration. This results in different behavior if EHC encounters plateaus, i.e., regions in the state space where the heuristic values of all successor states are not lower than the current best heuristic value. Using a best-first search rather than a breadth-first search is expected to result in shorter planning times or yield shorter plans on some domains, cf. [CS07, Sect. 6.3].

Problem domains The two problem domains used for our experiments are Blocks World and ECUs.

Blocks World is a classical problem domain in the area of AI planning. It consists of a table with a set of cubes that can be stacked upon each other. A cube can only be moved if there are no other cubes on top of it, and there is only one arm that can hold a cube, i.e., two cubes cannot be moved simultaneously. Finding an optimal solution in this domain has been shown to be NP-hard [GN92].

The ECUs domain works as explained in Section 4.2. In contrast to the Blocks World domain, which does not involve the object instantiation, the ECUs domain contains rules creating new nodes.

Experiment setup For the Blocks World domain, we used 8 different problem sizes (4, 6, 8, 10, 12, 14, 16, and 18 blocks) and 4 different problem instances (2 random initial and 2 random target configurations) per problem size.

For the ECUs domain, we used 4 different problem sizes (2, 3, 4, and 5 ECUs), each with 4 different problem instances. Two of these problem instances had the same number of component instances running in the initial configuration as ECUs were available. The other two problem instances had an additional component instance running. Each target configuration specified every second ECU (rounding down at odd numbers of ECUs) to be shut down.

The experiments were conducted on a Dual Intel Xeon E5520 compute server with 16 (virtual) cores running at 2.27GHz. Each experiment was given 4 cores and 4GB of RAM. If no plan could be computed within 20 minutes, the job was terminated.

Results First, we give an overview of the number of explored states for each combination of heuristic function and search algorithm. The number of explored states counts only those states that have been chosen for expansion and is generally less than the number of all generated states. Therefore, it is a suitable measure for

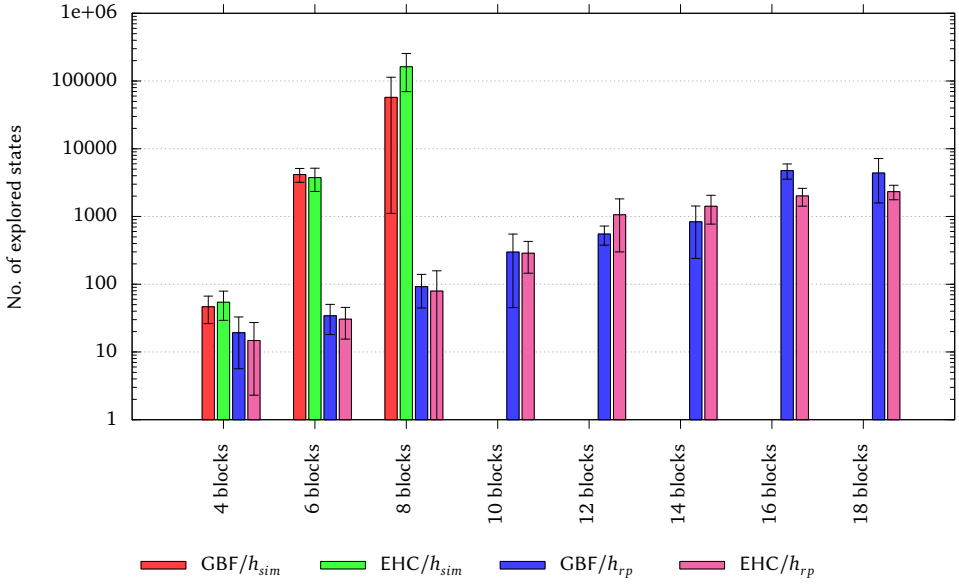


Figure 4.6: Histogram of the number of explored states in Blocks World domains

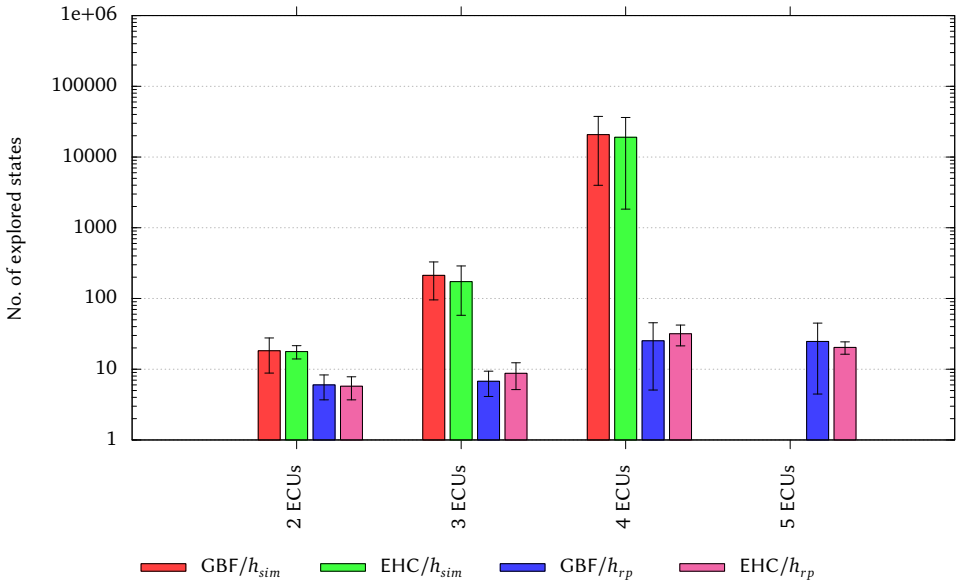


Figure 4.7: Histogram of the number of explored states in ECUs domains

how well the employed heuristic prunes the state space. Note that this number also does not include abstract states computed by h_{rp}.

Figure 4.6 shows a histogram of the average number of states for the BlocksWorld domain, Figure 4.7 for the ECUs domain. Note the logarithmic scale in both histograms. With increasing problem size h_{rp} makes its superiority clear. Combinations with h_{sim} failed to provide a solution within 20 minutes for the problems of size 10 blocks and above (on the BlocksWorld domain) and 5 ECUs (on the ECUs domain). There is no significant difference in performance between GBF and EHC.

Considering the average planning times in Figures 4.8 and 4.9, we can observe that h_{sim} performs better than h_{rp} on small domains. The performance of h_{rp} on small domains is worse than that of h_{sim} because the computation costs of finding a relaxed plan is in general much higher than the computation costs of counting the number of nodes and edges in a state. However, the performance changes to the favor of h_{rp} as the problem size increases: the planning time of h_{rp} scales better than the planning time of h_{sim}. This is expected because the number of generated states also scales better.

Note the small discrepancy between the number of explored states and the total planning time in the case of 5 ECUs. The number of explored states did not increase when switching from instances with 4 ECUs to instances with 5 ECUs, whereas the planning time did increase. This can be explained via the number of generated states. The number of generated states increased when switching from instances with 4 ECUs to instances with 5 ECUs. This led to more heuristic values being calculated, which in turn led to more *and better* candidates being available for further exploration. Better candidates led to a smaller number of states chosen for exploration due to a smaller average plan length.

Table 4.3: Percentage of time spent calculating heuristic values in BlocksWorld domains

#blocks	4	6	8	10	12	14	16	18
GBF/h_{sim}	4,58	3,44	4,97	—	—	—	—	—
EHC/h_{sim}	4,83	3,56	4,18	—	—	—	—	—
GBF/h_{rp}	89,80	93,65	94,31	95,16	97,28	97,81	99,40	99,02
EHC/h_{rp}	88,10	92,67	94,62	95,24	97,32	97,77	99,14	99,37

Table 4.4: Percentage of time spent calculating heuristic values in ECUs domains

#ECUs	2	3	4	5
GBF/h_{sim}	3,70	5,02	3,95	—
EHC/h_{sim}	3,99	6,74	9,87	—
GBF/h_{rp}	87,32	94,15	98,38	99,58
EHC/h_{rp}	81,60	91,88	98,46	99,74

Next, we take a detailed look at the time spent for calculating heuristic values. Tables 4.3 and 4.4 show these times in percentage of total planning time. While h_{sim} consumes only approx. 4% of the total planning time, h_{rp} consumes over 81%,

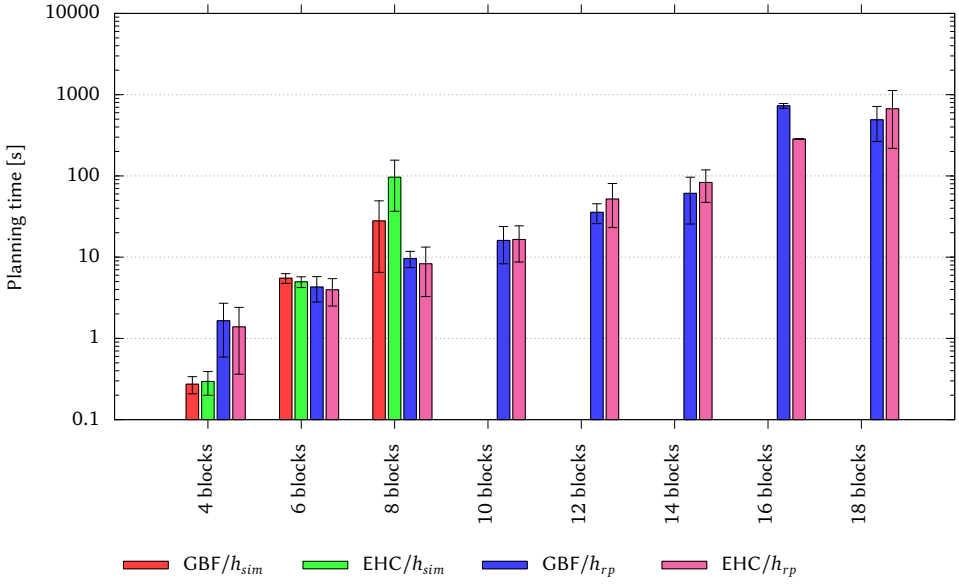


Figure 4.8: Histogram of planning times in Blocks World domains

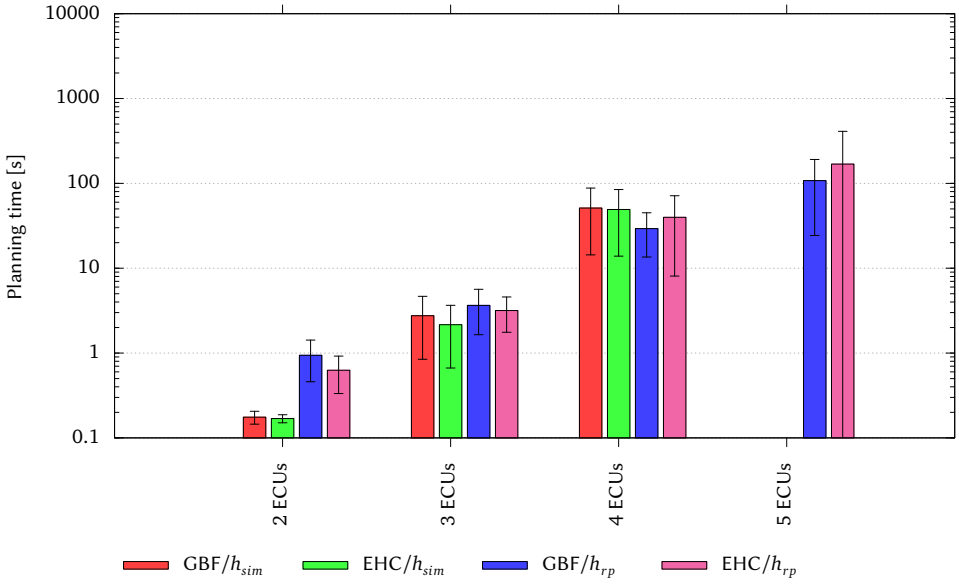


Figure 4.9: Histogram of planning times in ECUs domains

independently of whether used by GBF or EHC. Furthermore, the proportional share of h_{rp} increases with growing problem size. At the same time, the percentage of time needed for matching and transformation increases, which are the main contributors to the time spent for calculating heuristic values in the case of h_{rp}.

Note that our implementation is not optimized for efficiency. Therefore, it is more appropriate to consider the scaling behavior of both heuristic functions than their *absolute* planning times.

Table 4.5: Scaling factor of planning time in BlocksWorld domains. For each problem size, the entry is the quotient of the average planning time from all instances of this problem size and that of all instances from the next smaller problem size.

#blocks	6	8	10	12	14	16	18
GBF/h_{sim}	20,2	5,1	—	—	—	—	—
EHC/h_{sim}	16,8	19,4	—	—	—	—	—
GBF/h_{rp}	2,6	2,2	1,7	2,2	1,7	12,0	0,7
EHC/h_{rp}	2,9	2,1	2,0	3,1	1,6	3,4	2,4

Table 4.6: Scaling factor of planning time in ECUs domains. For each problem size, the entry is the quotient of the average planning time from all instances of this problem size and that of all instances from the next smaller problem size.

#ECUs	3	4	5
GBF/h_{sim}	15,7	18,6	—
EHC/h_{sim}	12,8	22,8	—
GBF/h_{rp}	3,9	8,0	3,7
EHC/h_{rp}	5,1	12,6	4,2

Tables 4.5 and 4.6 show the scaling behavior of both heuristics on both domains. Each entry shows a factor x, which is the quotient between the average planning time t_2 of all instances from one problem size and the average planning time t_1 of all instances from the next smaller problem size, i.e., $t_1 \cdot x = t_2$. Comparing the scaling factors of Tables 4.5 and 4.6 with one another, we can observe that h_{sim} performed equally bad on both domains, whereas h_{rp} performed slightly worse on the ECUs domain.

We suspect the inferior scaling behavior of h_{rp} on the ECUs domain to be caused by the amount of nodes being created during each abstract planning phase. This can possibly be improved by reducing this amount. The idea is to merge all nodes of the same type that are created during the same transition of an abstract state sequence into one node. When we do this, the labeling of new nodes also has to be modified: the question is which rule application label to attach to a new node if there were multiple transformations creating this node. Here, the idea is to choose that transformation whose applicability is easier to fulfill, i.e., that has the least number of rule application labels in its match. As a consequence of such a node merging approach, the amount of nodes being created during each abstract planning

phase is decreased, which should result in lower time requirements for applying rules to abstract states while preserving the quality of the heuristic.

4.5 Related Work

In the introduction of this chapter, we mentioned approaches that translate the graph transformation planning problem into PDDL as an alternative to planning directly on graph transformation systems. Unfortunately, such a translation imposes some restrictions due to the different expressive power of PDDL and graph transformation systems:

1. Today's proposed translation schemes [TK11; Mei12] support only a restricted set of negative application conditions. While it is technically possible to translate arbitrarily large negative application conditions into PDDL, such a translation requires the extension *disjunctive preconditions*, which is rarely supported by planners, or compiling the disjunctions away, i.e., by flattening those actions containing them, which results in a blowup of the number of actions.

2. Since PDDL does not support object instantiation, a translation-based approach cannot be used for planning problems where an unlimited number of nodes can be created. For such planning problems, a planning model designer has to specify the maximal number of objects before translating the model.

By planning directly within the transition system defined by the graph transformation system, we avoid these problems.

Röhs and Wehrheim [RW10b; Röh09] developed an approach to graph transformation planning that diverts a model checker from its intended use of searching for a counterexample of a given property. It plans by reformulating the planning problem into a model checking problem and then asking a model checker to verify the property that *no* plan exists. If the property is false, i.e., a plan exists, the model checker delivers a plan as counterexample of the property. While this approach is very generic and fully automatic, it is not competitive in terms of speed and quality compared to other planning techniques because the state space search of a model checker is generally not optimized for planning, neither for finding a plan quickly nor for finding a short plan.

Estler and Wehrheim [EW11; Est10] developed another approach to graph transformation planning that is, like our approach, based on heuristic search. In contrast to our approach, it employs a *domain-specific* heuristic to search through the state space. In general, a disadvantage of domain-specific heuristics is that they have to be developed specifically for each application domain. A heuristic that is working fine on one domain might not be suitable for another domain. In this approach, the solution to this problem is to learn heuristic functions automatically. A learning algorithm derives a *regression function* that predicts the costs of solving the problem from a given state. To derive the regression function, the learning algorithm needs a predefined declaration of state features and a training set with problem instances.

While this solution is an improvement over developing heuristic functions manually, it still requires the developer to declare a set of state features that is suitable for the given application domain. Such a thing is not necessary in our approach, because we employ a domain-independent heuristic.

Varró-Gyapay and Varró [VV06] also presented an approach to graph transformation planning, although not referred to as planning but *optimization*. Their approach, which is a search for an optimal plan, employs a Petri net abstraction technique. This Petri net abstraction technique derives a so-called *cardinality Petri net*, which counts the number of elements of certain types via tokens. For each type, there is a distinct place in this Petri net, and transitions simulate the effect of graph transformation rules by adding and removing tokens from appropriate places. The goal of the planning problem is then translated into a marking of the cardinality Petri net. The *coverability problem* for this goal marking, i.e., the question whether a marking exists that has at least the same amount of tokens on every place as the goal marking, is encoded into an *integer (linear) programming problem (IP)* [Sch98]. The solution to the IP is a vector, called *occurrence vector*, that states how many times each graph transformation rule has to be applied to reach the goal. This occurrence vector is used to cut off branches of the state space where the number of performed rule applications exceeds the number given in the vector for a certain rule.

An obvious downside of this approach is that a path corresponding to the occurrence vector might not exist in the state space of the graph transformation system. The reason for this is the applied abstraction: the cardinality Petri net abstracts away all structural information of graph transformations rules and configurations. If a path corresponding to the occurrence vector does not exist, the next best solution of the IP is derived. This continues iteratively until an occurrence vector has been derived for which a path exists.

Hegedüs et al. [HHV11] extended this approach by a second cut-off strategy and a choice of two heuristics. Although related to AI planning techniques, this extended approach was also not referred to as planning but *guided trajectory exploration* of graph transformation systems. In addition to employing the aforementioned Petri net abstraction technique, their approach computes a graph expressing dependency relations between graph transformation rules. The information encoded into this graph is used to define the new cut-off strategy and both heuristics. The new cut-off strategy cuts off a path if there is a disabled rule that still has to be applied according to the occurrence vector but cannot be enabled anymore, because each other rule enabling it would exceed the number of applications given in the occurrence vector for this rule. The first heuristic given is a *least commitment strategy*: it applies those rules first that enable the most other rules according to the dependency relations calculated before. The second heuristic applies those rules first that enable the application of other rules whose applicability is considered most important. The applicability of a rule is important if there is only a small number of rules enabling it and those that do only have a small number of applications left.

This extension provides two relevant improvements over the original work. First, by employing a heuristic function, the approach performs an informed search on

those parts of the state space that are not cut off by one of the cut-off strategies. Second, the new cut-off strategy and both heuristics are based on dependency relations between rules, and these dependency relations are derived from the structure of rules. Unfortunately, all cut-off strategies and heuristics are calculated entirely on the level of rules, i.e., independent of the graphs they are applied on. Therefore, none of them considers the actual structure of any host graph, e.g., the initial or current configuration.

The Petri net abstraction technique has also been extended to support a notion of time, essentially by modifying the encoding as IP, cf. [Var12], and the cardinality Petri net has also been used to prove the termination of graph transformation systems, cf. [Var+06]. Unfortunately, all these variants of the approach do not support arbitrary kinds of NACs. They do support a restricted kind of NACs, i.e., NACs that prevent rules from being applied twice at the same match by mimicking the RHS of a rule, cf. [Var+06].

Approaches to planning with graph transformations have also been employed in safety-critical environments, cf. [Gau+14, Sect. 3.2.9]. Here, unsafe configurations, e.g., an unsafe distance between two RailCabs or a RailCab not being registered at a base station while driving, have to be prevented from occurring in a plan. To do so, these approaches take safety requirements into account. These safety requirements restrict the set of valid configurations, i.e., they specify which configurations are not allowed to occur in a plan.

The objective of these approaches has similarities to that of verification approaches for graph transformations systems, cf. [GRS14, Sect. 5.2]. However, in contrast to verification approaches, where the absence of unsafe states is categorically guaranteed at design time, these techniques allow unsafe states to exist in the reachability graph in general, but plan reconfigurations in such a way that no unsafe state is reached.

Depending on the application domain, it can be very complicated to guarantee the absence of all forbidden patterns by means of design-time verification. The exclusion of a forbidden pattern via design-time verification imposes restrictions on the state space of the system, as it is not allowed to contain states matching the forbidden pattern. This, in turn, transfers these restrictions to the design of the application domain's reconfigurations. If these restrictions prove too cumbersome, we can instead allow the forbidden patterns to appear in the state space in principle, but plan such that they do not appear on the path to the target configuration. Of course, when doing this, we do not need to take forbidden patterns into account whose absence has already been verified by a design-time verification.

In safe planning environments, the planning task, as defined in Definition 4.1.2, is extended such that it includes the requirement that no potentially unsafe configuration is reached. In essence, this is done by adding a safety specification that defines whether a configuration meets the safety requirements and is thus allowed in a plan. The safety specification is given as a set of graph patterns, called *forbidden patterns*. If any one of the forbidden patterns matches the host graph, the host

graph is a forbidden configuration, i.e., a configuration that does not meet the safety requirements.

All approaches to graph transformation planning mentioned above either support or can easily be extended to support forbidden patterns and thus are capable of solving the safe planning problem. In the approach based on model checking, the problem is solved by including the safety requirements into the property to be verified, cf. [RW10b]. In doing so, the property states that no *safe* plan exists, i.e., there is no path to a goal state free of intermediate states containing forbidden patterns. Therefore, the model checker must also consider whether any state on the path to the goal state contains a forbidden pattern. In the approach employing a (learned) domain-specific heuristic, checking for forbidden patterns is simply integrated into the search algorithm, cf. [EW11]. This integration is independent of the learning algorithm and can be done in any forward search, i.e., it can be done analogously in our approach or the approach employing the Petri net abstraction technique.

Translations into PDDL (or any other dedicated planning language) do not support forbidden patterns at the moment. Although PDDL supports constraints over intermediate states of a plan since version 3.0 [GL05], translation schemes do not yet support the translation of forbidden patterns into such constraints. Earlier proposed translation schemes only support the translation of graph transformation rules into action schemes of PDDL, cf. [TK11; Mei12]. As remarked above, their capability to support negative application conditions is also limited.

Whether learning a domain-specific regression function or employing a domain-independent heuristic function is preferable, depends on the application domain. If the application domain allows for a straightforward design of a suitable heuristic using human intuition or provides a meaningful set of state features to derive a heuristic function using machine learning techniques, then the regression function might be preferable. If, however, the domain does not provide a meaningful set of state features, i.e., heuristic knowledge is not easy to obtain, then our domain-independent heuristic is the more reasonable choice.

4.6 Discussion

Our approach to graph transformation planning is an adaptation of FF's heuristic function to graph transformation systems. In contrast to our system, which uses label propagation to find the relaxed plan, FF computes the relaxed plan by a backward search on a structure called the *planning graph* [BF97]. Such a planning graph roughly resembles our list of successor graphs in the abstraction.

A planning graph is a directed graph with two kinds of nodes: nodes representing literals and nodes representing (ground) actions. These nodes are arranged in multiple layers. The planning graph starts with a layer of literals, i.e., with those literals available in the initial state. The following layer of actions corresponds to those actions that are applicable in the initial state. Such an action layer also contains a *no-op* for each literal, i.e., an action that copies the literal into the next layer. A

literal layer and an action layer together form a so-called *step* of the planning graph. Each later step also contains two such layers. The i-th literal layer contains those literals that can be asserted within i steps. The i-th action layer contains those actions that are applicable given the i-th literal layer. Note that the first two layers form step 0.

A planning graph has edges between nodes of different layers that mirror the conditions and effects of actions. If a literal in step i is contained in the precondition of an action in step i, there is an edge from the literal to the action. If an action in step i asserts a literal in step $i+1$, there is an edge from the action to the literal. These edges enable to consider the relation between literals and actions when searching through the planning graph for a plan.

In general, planning graphs also have mutual exclusion edges between actions that interfere with one another and between literals that cannot be achieved at the same step simultaneously. However, in the case of a relaxed planning task, there are no mutual exclusion edges because no literal is ever deleted, i.e., the relaxed planning graph is a bipartite graph.

If a layer is reached that contains all goal literals, a backward search for a plan is performed. This is done by selecting an *achiever*, i.e., a (ground) action asserting the literal, for each literal in the goal set. Then, this selection is recursively applied for all literals in the preconditions of the selected actions. When this search reaches the first step, no actions need to be selected anymore and the set of selected actions constitutes the plan. Note that in general, the backward search for a plan can require backtracking when no action can be selected that is not exclusive to actions selected earlier. Since there are no mutual exclusion edges in a relaxed planning graph, no backtracking occurs here. This is what makes the search for a relaxed plan in a planning graph much more efficient than the search for a non-relaxed plan.

By applying the idea of relaxed planning to graph transformation systems instead of PDDL's propositional state representations, we face multiple differences. These differences stem from the fact that graph transformation systems, unlike PDDL, support object instantiation and from the integration of NACs into the abstract planning algorithm.

Achievers and admissibility In FF's relaxed planning graph, a literal can have multiple achievers. A relaxed plan is found by choosing an achiever for each literal in the goal match and for each unfulfilled literal in the precondition of other achievers. Finding the optimal relaxed plan, which results in an admissible heuristic, is NP-hard, cf. [Byl94]. Therefore, FF uses a heuristic for selecting achievers, which prefers those actions whose preconditions are easier to fulfill. While this does not result in an admissible heuristic anymore, it works well in practice and allows to find a relaxed plan in polynomial time.

In our case, there are no multiple achievers for created elements. Each created element has a rule application label identifying the graph transformation that created it. Therefore, no search for an optimal set of achievers is necessary for created elements. Elements marked as *deleted* also have only one rule

application label, i.e., the label from the first rule application that intended to delete the element. When finding an element marked as *deleted* in a NAC match while collecting all rule application labels, this amounts to choosing the first transformation that intended to delete this element and thus resulted in this element being marked. This is similar to FF's approach of preferring those actions whose preconditions are easier to fulfill.

Like the heuristic of FF, our heuristic is not admissible. We can easily create an example domain where our heuristic function counts three graph transformations, e.g., graph transformations that have been applied in parallel to reach the (abstract) goal state in one iteration, although a plan of length two exists, e.g., a plan that requires its two graph transformations to be applied in sequence. In such an example, the overestimation of the costs of reaching a target configuration is essentially caused by the early termination of the successor graph creation loop.

Object instantiation The creation of nodes can lead to an explosion of the graph size during the creation of successor graphs in the abstraction. During each iteration of the successor graph creation loop, the size of the next abstract state grows. This is because for each applicable transformation creating one or more nodes, all those nodes are created by the parallel execution of these transformations. Furthermore, each creation of an element results in a new element even if such an element does already exist, possibly even created by the same rule in an earlier iteration. This increases the graph size of each abstract successor state and thus the matching costs. This is also the reason that target graph patterns are likely to have multiple matches in an abstract state, each with a different heuristic value. Such an explosion of the size of a state is not an issue in PDDL-based planners because they do not support object instantiation.

An idea to reduce the number of new elements per iteration is to merge all new nodes of the same type into a single node. This, however, affects the propagation of rule application labels. It is not possible anymore to distinctly identify the rule application that was responsible for creating a new node if there was more than one applicable transformation creating a node of the new node's type. In such a case, we can again prefer those graph transformations whose preconditions are easier to fulfill, i.e., whose LHS matches needed less newly created elements and less graph transformations to create them, similar to FF's heuristic for selecting achievers.

Negative application conditions The equivalent to NACs in PDDL are negative existential quantifications over conjunctive facts. They are usually solved by compiling them away, i.e., translating them into DNF, which results in a blowup of the propositional domain representation, cf. [HN01]. In our approach, we avoid such a blowup by building the support for NACs directly into the abstract planning algorithm. As soon as a graph element marked as

created or *deleted* is found within a NAC match, there is no need to check any remaining elements in the same NAC match.

We motivated the development of our approach to graph transformation planning by arguing about the need to retain the expressiveness of graph transformation systems when solving graph transformation planning problems and the chances of adapting already known techniques from PDDL-based planning systems. Adapting the idea of FF's heuristic function, i.e., using the solution length of a relaxed problem as heuristic estimate, is only one of the possibilities. Another possibility that we deem promising is the adaptation of *landmark recognition* techniques.

A *landmark* is a literal or a set of literals that occurs in every valid plan. Porteous, Sebastia, and Hoffmann [PSH14] introduced the notion of landmarks in 2001. They identify landmark candidates via a backward search through a relaxed planning graph and verify that they are indeed landmarks by checking whether a relaxed planning graph without those actions achieving a landmark candidate reaches a state satisfying the goal. This approach, which can be performed in polynomial time, is sound but not complete. Since checking whether or not a literal is a landmark is PSPACE-complete, cf. [HPS04], a complete approach is not considered worth the effort. By now, there are several techniques on finding landmarks. The work of Marzal et al. [MSO11] presents a great overview of the most relevant techniques and combines these techniques into one to increase the percentage of landmarks found.

When employing a heuristic based on landmarks, it is also important to find useful orderings among landmarks. Landmarks can then be considered as subgoals that have to be reached in sequence to reach the goal. They can either be used to decompose the planning problem into several subproblems, cf. [PSH14], or to derive a heuristic that states how many landmarks still have to be achieved in the correct order. The LAMA planner [RW10a], which won the sequential satisficing track of the 6th International Planning Competition (IPC-2008), uses such a heuristic in a *multi-heuristic search*, i.e., it alternates between different heuristics, the landmark heuristic and the heuristic of FF, to benefit from both approaches orthogonally.

The success of the LAMA planner gave motivation to adapt landmarks-based techniques to graph transformation planning. A first step into this direction was made by Ahmadian [Ahm12], who also developed a predecessor version of our graph transformation planning approach, which uses the length of the *parallel* relaxed plan as heuristic estimate. He adapted a technique of Zhu and Givan [ZG03], which propagates landmark information through a planning graph via labels.

An important aspect of this adaptation concerns the representation of landmarks itself. In propositional state representations, a landmark is a literal. Such a literal can include information about related objects, e.g., a literal with two parameters can express which software component is deployed on which ECU. This is not as easy in graph transformation systems, because it is not possible to rely on the identity of nodes that do not yet exist. Therefore, Ahmadian defined landmarks on the type level. Unfortunately, this makes them imprecise because they do not provide any structural information. A node landmark only states that a node of a certain type has to exist, and an edge landmark only states that an edge of a certain edge type has to

exist between two nodes of certain types. An idea to integrate structural information into such landmarks is to consider the combination of multiple edge landmarks involving the same nodes as a landmark on its own. Such a "higher-order" landmark conforms to what is known in related work as a *conjunctive landmark*, cf. [KRH10].

5 Durative Graph Transformation Systems

This chapter presents a formalism for graph transformations with time in concurrent contexts. This formalism, called *durative graph transformation systems (DGTS)*, provides concepts to specify structural reconfigurations whose execution consumes time as well as dependencies between such reconfigurations. These concepts enable an intuitive specification of temporal reconfigurations on a high level of abstraction. Their formal semantics have been designed such that planning and verification techniques can be applied reasonably.

Durative graph transformation systems provide three kinds of rules: *durative graph transformation rules*, *concurrency rules*, and *urgency rules*. From these three kinds of rules, durative graph transformation rules are the most intelligible concept. Syntactically, they are a straightforward extension of ordinary graph transformation rules, i.e., each graph transformation rule is annotated with a natural number representing its execution time. The formal semantics employs a *locking mechanism*. The idea of this locking mechanism is similar to concurrency control methods implemented by database management systems. Basically, it restricts read or write access to nodes and edges while they are involved in a durative graph transformation. This guarantees that multiple durative graph transformations can not be executed concurrently if they have conflicting needs, cf. [ZH13b]

Concurrency rules and urgency rules formalize temporal dependencies between different durative graph transformation rules. The idea for concurrency rules is inspired by the notion of *envelope actions* [HLF03] in PDDL planning domains. An envelope action is an action whose execution acts as a time window for another action, i.e., the other action requires the envelope action to be applied concurrently. In durative graph transformation systems, we employ concurrency rules to specify such dependencies. In doing so, we allow the envelope to be a disjunction of multiple transformations: there may be multiple durative graph transformations t_2 acting as a time window for a durative graph transformation t_1 and executing any one of them is sufficient to allow the execution of t_1.

The idea of urgency rules is inspired by that of urgent locations, cf. [BDL04], and urgent transitions, cf. [BST99; BT04], both concepts of timed automata. An urgency rule specifies that a certain durative graph transformation t_2 has to follow another durative graph transformations t_1 urgently, i.e., within a given time frame since the execution of t_1 finished.

Note that the temporal dependencies specified via concurrency and urgency rules appear on the level of transformations, not the level of rules. Consider an example of two robotic arms: one rule implements a robotic arm to continuously rotate an object, another rule specifies a robotic arm to apply adhesive to an object that is continuously being rotated by another robotic arm. In this example, the second rule depends on a concurrent application of the first rule. However, if there are multiple robotic arms and objects, it matters which robotic arm rotates which object. Therefore, a concurrency rule that specifies such a dependency has to include information that defines how the matches of different rules have to relate to each other. The same holds for urgency rules.

The formal semantics of all these rules are based on timed graph transformation systems, i.e., any given durative graph transformation system can be translated into a timed graph transformation system. Obviously, instead of specifying a system model as a durative graph transformation systems, a modeler could decide to specify the system model directly as a timed graph transformation systems. However, this would be much less convenient. In the TGTS formalism, the application of rules is timed, but instantaneous, i.e., timed graph transformations do not consume time. Instead, time passes in between two consecutive graph transformations. A durative graph transformation rule could be simulated via two timed graph transformation rules, but this would mean solving a problem manually and repeatedly hat has already been solved by the semantics of durative graph transformation systems. Furthermore, it would require the use of other constructs of timed graph transformation system, i.e., clock instance rules, which enable the measurement of time, and invariant rules, which specify timed conditions. However, the manual handling of clock instances is a tedious duty and can be an error-prone endeavor. Durative graph transformation systems have the advantage that such clock instances are abstracted away and concurrent and urgent behavior are made explicit.

Due to being based on timed graph transformation systems, we can make use of the verification procedures for timed graph transformation systems provided by Heinzemann et al. [HE10] and Suck et al. [SHS11]. The approach by Heinzemann et al. [HE10] enables to check whether or not a forbidden graph exists in any state of a timed graph transformation system's state space, i.e., it is possible to verify CTL formulas of the form $EF\phi$ and $AG\neg\phi$. In this approach, the absence of a forbidden graph is verified by a backward rule application from the forbidden graph to the start graph. This has previously been done (for untimed graph transformation systems) by Becker et al. [Bec+06] both via an explicit search and the use of symbolic encodings. The approach by Suck et al. [SHS11] introduces a first-order variant of TCTL [ACD93] and enables a verification of first-order TCTL formulas by translating them into TCTL model checking problems for timed automata.

After introducing the running example for this chapter in the next section, the syntax and semantics of durative graph transformation rules is presented in Section 5.2. Being based on timed graph transformation systems, this section also explains the concepts available in the TGTS formalism. For reasons of clarity, the support for negative application conditions is left out for now. Then, Section 5.3 covers how a durative graph transformation correlates with an untimed graph transformation, the termination of durative rules, and possible interleavings among multiple durative rules. The DGTS formalism is extended successively in Section 5.4 to support forbidden edges and forbidden pairs, in Section 5.5 to support concurrency rules, and in Section 5.6 to support urgency rules. Related work in the area of graph transformations with time is covered in Section 5.7. Eventually, Section 5.8 concludes this chapter with a discussion on design decisions regarding the syntax and semantics or concurrency and urgency rules.

5.1 Application Example: RailCab System

Each of the three kinds of rules in the DGTS formalism can be motivated with the help of the RailCab system, see Section 1.4, which is why we use a domain of the RailCab system as a running example in this chapter. Instead of providing all rules for the RailCab system at once, we show them as needed, i.e., in introductory paragraphs and syntax sections within the remainder this chapter. Here, we give a general overview on how the RailCab system is modeled.

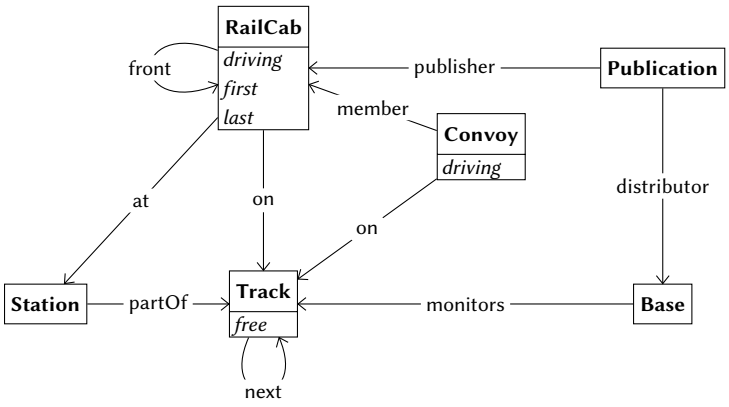


Figure 5.1: Type graph of the RailCab domain

Figure 5.1 shows a type graph for the rules in the durative graph transformation system that models the RailCab domain. RailCabs operate on a railway system whose physical structure is specified as part of the system configuration. The railway system consists of track segments that are connected to each other via `next` edges. A RailCab can occupy one such track segment at a time, which is represented by an `on` edge to the track segment. Furthermore, RailCabs can coordinate with other RailCabs to form a convoy. Such an active convoy operation is represented in a configuration

by a node of the `Convoy` type. A `Convoy` node has a `member` edge to each participating RailCab and represents an active instance of the RTCP ConvoyCoordination as well as one instance of the RTCP DistanceControl for each pair of neighboring RailCabs in the convoy. The position of each RailCab in the convoy is given by the `first` edge, `last` edge, and `front` edges in a configuration. Within a convoy, there is a `front` edge between each pair of neighboring RailCabs. The `first` and `last` edge are self edges that represent the head and tail of the convoy.

The application scenario mainly consists of reconfigurations to move RailCabs or convoys of RailCabs as well as reconfigurations related to convoy instantiation, deinstantiation, and membership change. Each of these reconfigurations will be specified as a durative graph transformation rule. RailCabs drive slower if they are on their own because driving alone is less energy efficient than driving in a convoy. Therefore, these rules will have different durations.

Remember that RailCabs also have to communicate with base stations of the RailCab system. Each RailCab has to be registered at a base station that monitors the track segment that the RailCab occupies. This is represented by an instance of the RTCP Publication. When a RailCab moves from a track segment monitored by one base station to a track segment monitored by another base station, it has to deinstantiate this RTCP and instantiate a new one with the new base station. This change of the RailCab's registration is a reconfiguration that is required to be executed concurrently to the movement of the RailCab. Therefore, this requirement will be modeled as a concurrency rule.

In this application scenario, a RailCab is not allowed to stop abruptly if it is in driving motion. To be allowed to stop, a RailCab first has to brake while still moving to one track segment ahead. Being not allowed to stop abruptly means that there may be no pause between multiple consecutive transformations moving a RailCab; they have to be applied continuously without intermission. Technically, as soon as a reconfiguration that moves a RailCab to another track segment finishes (and the RailCab did not brake during this reconfiguration), another reconfiguration moving this RailCab has to start. This requirement will be specified by means of urgency rules.

5.2 Durative Graph Transformation Rules

Adding a notion of durations to graph transformation rules is trivial if the execution of these rules is assumed to be strictly sequential. However, defining a timed semantics for graph transformation rules allowing a concurrent execution is difficult due to the many ways multiple transformations can interact with each other. While unproblematic in some cases, the execution of multiple graph transformation rules simultaneously, i.e., applying them to the same configuration in parallel, can lead to conflicts in other cases.

Technically, durative graph transformations can be realized by translating them into two discrete graph transformations that are temporally linked to each other. One graph transformation represents the start of the durative transformation; a

second one represents its end. In doing so, durative graph transformations have application intervals, and as a result, it is possible to apply multiple durative graph transformations concurrently.

The question is when to actually perform the reconfiguration that is specified by the durative graph transformation rule. Performing it as part of the discrete graph transformation that represents the start of the durative graph transformation would not be a reasonable solution. The state of the system would be changed long before the durative transformation finished its execution. Therefore, we execute the reconfiguration as part of the second discrete graph transformation.

Unfortunately, there might be conflicts between two durative graph transformations if their matches are allowed to overlap arbitrarily. Such a conflict can cause discrete graph transformations representing the end of a durative graph transformation not to be applicable when they are due. Consider a naive approach, which simply uses the application conditions of those discrete graph transformations that represent the start of a durative transformation to decide whether or not multiple durative graph transformations may be applied concurrently. In this case, a discrete graph transformation representing the end of a durative graph transformation might not be applicable when it is due, because other graph transformations that have been applied concurrently may have invalidated its application condition.

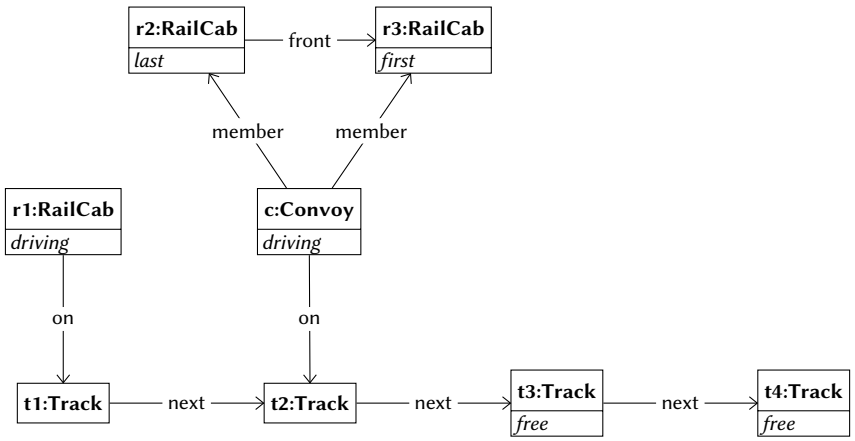


Figure 5.2: A configuration in the RailCab domain

As an example, consider the configuration given in Figure 5.2 and the two graph transformation rules `joinConvoy` and `dissolveConvoy` given in Figures 5.3 and 5.4. Each of these rules has only one match in the configuration. The match of `joinConvoy` maps to the `Convoy` node `c`, `RailCab` nodes `r1` and `r2`, and `Track` nodes `t1` to `t3`, that of `dissolveConvoy` to `Convoy` node `c`, `RailCab` nodes `r2` and `r3`, and `Track` nodes `t2` to `t4`. Let us assume that one of the two rules, `dissolveConvoy`, is currently being applied. This means that its condition has already been checked but its actual reconfiguration not yet been executed. Let us further assume that an execution of `joinConvoy` is scheduled to start while `dissolveConvoy` is being executed and to end after the execution of `dissolveConvoy` finished. Since `dissolveConvoy` ends before

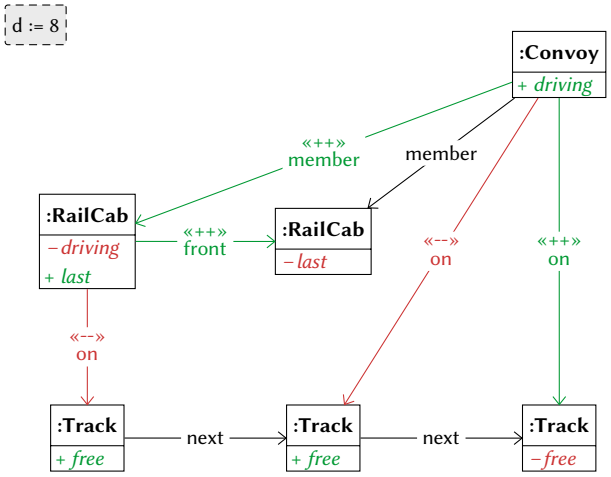


Figure 5.3: Durative rule `joinConvoy`

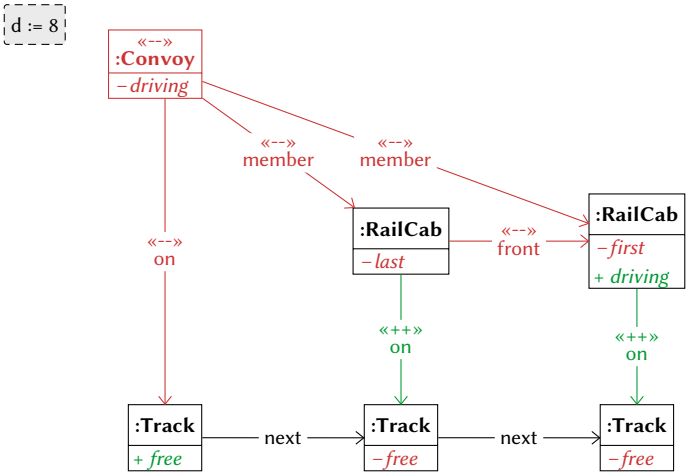


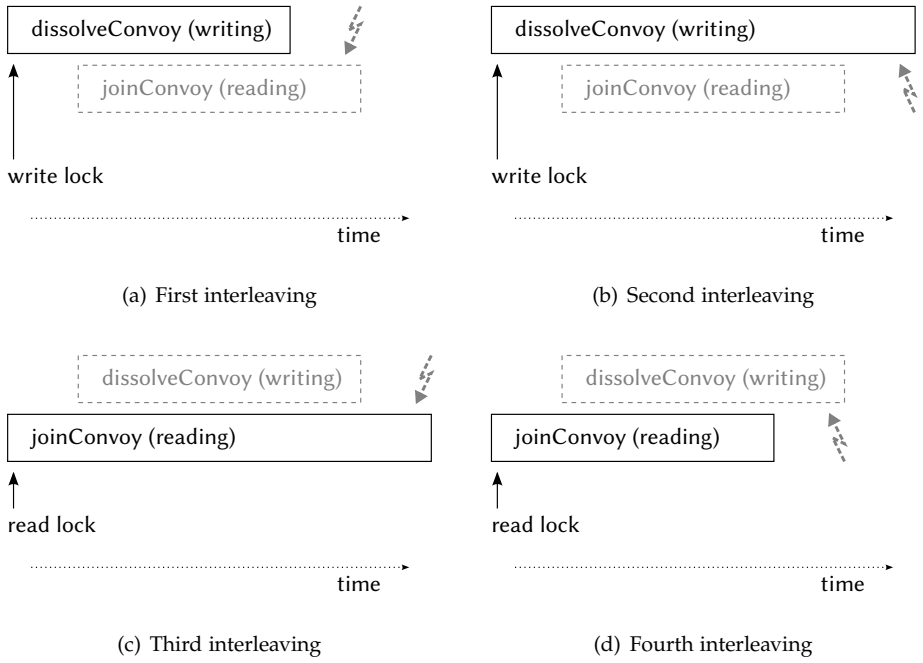
Figure 5.4: Durative rule `dissolveConvoy`

`joinConvoy` ends, there will be no `Convoy` node anymore that the joining `RailCab` node can be connected to. If such a sequence of reconfigurations was to be executed, a rear-end collision might occur. Since the `Convoy` node is going to be deinstantiated by `dissolveConvoy`, the execution of a graph transformation that makes use of the `Convoy` node makes no sense and should not be allowed. The problem is that the system is in the process of being reconfigured but this is not reflected in the intermediate configuration. Checking the applicability at the beginning of a durative graph transformation and performing the actual graph transformation discretely at its end is ineligible as a general solution.

To solve this problem, we add information about the execution of durative graph transformations into the configuration. This can be seen as locking access to the

elements of the configuration. Whether a second durative graph transformation is allowed to be applied concurrently, can thus be checked by testing for the locks.

Given two durative graph transformations, there are four different interleavings that might result in conflicts. In all of them, if the interleaving results in a conflict, then at least one element is concurrently being read (required or forbidden) by one durative graph transformation and being written (deleted or created) by another. The four interleavings differ only in their beginning and ending times.



(a) First interleaving (b) Second interleaving

(c) Third interleaving (d) Fourth interleaving

Figure 5.5: Prevention of conflicting interleavings

A schematic overview of these four conflicting interleavings for the durative graph transformation rules `joinConvoy` and `dissolveConvoy` is shown in Figure 5.5. The aforementioned example, in which the execution of `dissolveConvoy` began before `joinConvoy` started its execution, corresponds to the first two interleavings. In both interleavings, `dissolveConvoy` removes the edge between `RailCab` node `r2` and `Convoy` node `c` although it is required by `joinConvoy`. In the first interleaving, the execution of `dissolveConvoy` ends first, which causes the convoy to be deinstantiated and the ongoing transformation of `joinConvoy` not to work anymore, because the `Convoy` node has vanished. In the second interleaving, the execution of `joinConvoy` ends first, which results in a new configuration where the execution of `dissolveConvoy` can still perform its reconfiguration. However, such a concurrent execution of `joinConvoy` and `dissolveConvoy` is not intuitive: although `r1` joined a convoy and was not involved in the durative graph transformation of `dissolveConvoy`, there is no convoy that `r1` can be member of after the execution of both transformations has been finished. From the perspective of `dissolveConvoy`,

the application of `joinConvoy` did not take the pending changes of `dissolveConvoy`'s execution into account. Our solution to this problem incorporates information about the deletion of the `Convoy` node into the configuration by acquiring a write lock on the `Convoy` node when the execution of `dissolveConvoy` starts and releasing the lock when it ends.

Although the third and fourth interleaving differ from the first two in their ordering of the transformations' starting points, they are essentially caused by the same reason: either the changes resulting from `dissolveConvoy`'s execution cause the convoy to be deinstantiated while `r1` joins the convoy (third interleaving) or the application of `joinConvoy` does not take the pending changes of `dissolveConvoy`'s execution into account (fourth interleaving). These situations, however, require a different solution due to their different ordering of starting points. Since the durative graph transformation of `joinConvoy` starts first, the write lock on the `Convoy` node has not yet been acquired by that of `dissolveConvoy`. Therefore, the transformation of `joinConvoy` itself incorporates into the configuration that it requires the `Convoy` node by acquiring a read lock of the `Convoy` node. In doing so, no concurrent graph transformation of `dissolveConvoy` can deinstantiate the convoy.

In short, our solution approach uses write locks on parts of the system's configuration to avert conflicting interleavings similar to those illustrated in Figures 5.5(a) and 5.5(b) and read locks to avert conflicting interleavings similar to those illustrated in Figures 5.5(c) and 5.5(d). This is essentially an implementation of two-phase locking (2PL) [BHG87] on timed graph transformation systems. Since the locking and unlocking of all nodes and edges happen atomically at the beginning and end of a durative graph transformation, no deadlocks can occur. As a result of this, durative graph transformations that are not in conflict with each other allow to interleave their end points in any order that is compatible with their specified durations.

The next section explains the syntax of durative graph transformation systems. Section 5.2.2 introduces *timed graphs* and *clock instances*, which both are needed for the notion of a configuration in timed graph transformation systems. Then, Section 5.2.3 gives an informal overview of the semantic concepts of durative graph transformation systems, i.e., it explains how the locking mechanism is technically realized and how the execution of durative graph transformations is tracked. In Sections 5.2.4 and 5.2.5, we map durative graph transformation rules to rules of the TGTS formalism. These rules are then given an operational semantics in the form of a transition system in Section 5.2.6. The formalization of timed graph transformation systems provided in the following sections loosely follows that given by Suck et al. [SHS11] and Eckardt et al. [Eck+13].

5.2.1 Syntax

Our notion of durative graph transformation rules is inspired by the fact that reconfigurations in software systems require time. On the syntactic level, a durative rule is merely an ordinary graph transformation rule with an annotated name and a value for its duration. The idea is that the execution of a durative graph

transformation cannot be aborted once it has been started. A modeler who uses durative rules for specification does not need to deal with the error-prone definition of multiple discrete rules and their correct timed behavior, potentially leading to conflicts caused by a concurrent execution. This is completely resolved by the semantics.

Definition 5.2.1 (Durative graph transformation rule)**.** A *durative graph transformation rule* $\mathcal{D} = (L, R, r, name, d)$ consists of

- two typed graphs, a left-hand side L and a right-hand side R,
- a partial graph morphism $r : L \to R$,
- a distinct name *name*, and
- a duration $d \in \mathbb{N}^{>0}$.

To specify a complete durative graph transformation system, a modeler has to specify a type graph and an initial graph in addition to the durative rules. The semantics of such a durative graph transformation system is given by a timed graph transformation system whose different kinds of rules are all induced by durative rules.

Definition 5.2.2 (Durative graph transformation system)**.** A *durative graph transformation system* $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR})$ consists of

- a type graph $\mathcal{TG}$,
- an initial typed graph G_0^T, which is typed over $\mathcal{TG}$, and
- a set of durative graph transformation rules $\mathcal{DR}$, each of whose LHS and RHS are typed over $\mathcal{TG}$.

5.2.2 Timed Graphs and Clock Instances

The semantics of durative graph transformations is defined by a mapping to rules of the TGTS formalism. We refer to this mapping as an *inducement* of rules. A durative rule induces a pair of *timed graph transformation rules*, called *start rule* and *end rule*. Intuitively, the application of the start and end rule indicate the interval of the durative rule's execution. The durative rule further induces a *clock instance rule*, which triggers the measuring of time, and an *invariant rule*, which enforces the application of the end rule after d time units have passed since the application of the start rule. A schematic overview of this approach is given in Figure 5.6.

All these rules work on *timed graphs*, which employ *clock instances* to support a concept of time. The purpose of clock instances is to measure the passing of time and to facilitate making timed statements, e.g., formulating a condition over the allowed values of a clock instance. Such conditions have to be formulated along with the graph transformation rules at design time.

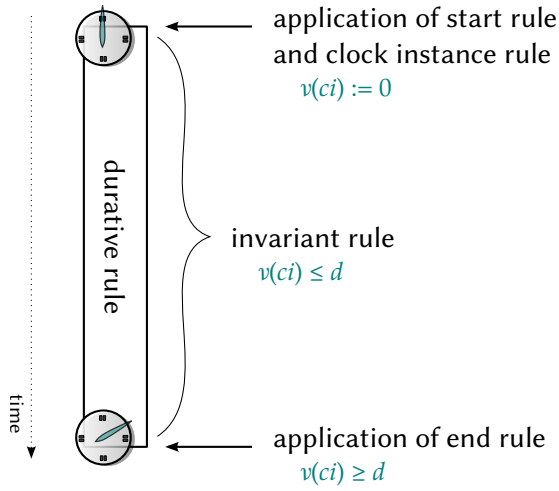


Figure 5.6: Schematic overview of a durative rule's execution

Similar to clocks in timed automata, cf. [AD94; BY04], the values of all clock instances increase continuously and synchronously with the same rate. However, clocks in timed automata pertain to a complete timed automaton or a complete timed system. This is not reasonable in a timed graph transformation system due to the dynamic nature of graph transformation systems. In timed graph transformation systems, clock instances pertain to parts of a configuration, i.e., specific structures of nodes and edges. These parts can be dynamically created and deleted; they might even exist more than once in a configuration. As a consequence, it is impossible to decide at design time how many clock instances a timed graph transformation system needs. For this reason, timed graph transformation systems allow to instantiate and deinstantiate clock instances at runtime.

We define timed graphs as an extension of typed graphs. A timed graph contains two kinds of nodes: ordinary graph nodes and clock instances. Since a clock instance pertains to a subgraph of a timed graph, it has an edge to each node of the subgraph but no other edges.

Definition 5.2.3 (Timed graph)**.** Let TG be a distinguished graph, called *type graph*. A *timed graph* $TiG = (G, type)$ consists of a graph G and a graph morphism $type : G \to TG$ where $G = (V_G, V_{CI}, E_G, E_{CI}, src, tgt)$ and

- V_G and V_{CI} denote the set of graph nodes and clock instances, respectively,
- E_G and E_{CI} denote the set of graph edges and clock instance edges, respectively,
- $src : E_G \cup E_{CI} \to V_G \cup V_{CI}$ denotes the source function for graph edges and clock instance edges and is defined such that $src|_{E_G} : E_G \to V_G$ and $src|_{E_{CI}} : E_{CI} \to V_{CI}$, and

- $tgt : E_G \cup E_{CI} \rightarrow V_G$ denotes the target function for graph edges and clock instance edges.

To be able to define timed graph transformation rules and represent their matches to a configuration, we also need a concept of morphisms on timed graphs. Such timed graph morphisms work exactly like a typed graph morphisms, i.e., they commute for the source and target function and preserve the types of nodes and edges.

A timed graph itself does not have any values assigned to clock instances. However, to be able to represent the snapshot of a system, we also need an assignment of values to clock instances.

Definition 5.2.4 (Clock instance value assignment)**.** Let V_{CI} be a set of clock instances. A *clock instance value assignment* is a function $\nu : V_{CI} \rightarrow \mathbb{R}^+$ that assigns a non-negative real value to each clock instance. A *clock instance reset* changes the values of a subset V_{res} of all clock instances in a clock instance value assignment to zero, i.e., for $V_{res} \subseteq V_{CI}$, $\nu' = \nu[V_{res} \mapsto 0]$ is defined as $\nu'(ci) = 0$ for all $ci \in V_{res}$ and $\nu'(ci) = \nu(ci)$ for all $ci \in V_{CI} \setminus V_{res}$. A *time delay* is an addition of a non-negative real value δ to the value of each clock instance in a clock instance value assignment, i.e., for $\delta \in \mathbb{R}^+$, $\nu'' = \nu + \delta$ is defined as $\nu''(ci) = \nu(ci) + \delta$ for all $ci \in V_{CI}$.

A timed graph and a clock instance value assignment together form a configuration. The notion of a configuration serves as a basis for the operational semantics we define later in Section 5.2.6.

Definition 5.2.5 (Configuration)**.** A *configuration* is a tuple $\langle TiG, \nu \rangle$ where TiG is a timed graph and ν a clock instance value assignment.

The TGTS formalism includes rules enabling a modeler to formulate conditions over the allowed values of a clock instance. To enable the specification of these conditions, we need a definition of clock instance constraints. Such clock instance constraints can be used to restrict the allowed values of a clock instance to a specific interval. They can be employed either as an additional application condition of a rule or as an invariant.

Definition 5.2.6 (Clock instance constraint)**.** Let V_{CI} be a set of clock instances. A *clock instance constraint* is a conjunctive formula of atomic constraints of the form $c_i \sim n$ or $c_i - c_j \sim n$ where $c_i, c_j \in V_{CI}$, $\sim \in \{<, \leq, =, \geq, >\}$, and $n \in \mathbb{N}$. $\mathcal{Z}(V_{CI})$ denotes the set of clock instance constraints over V_{CI}.

Of course, we also need to be able to express whether a clock instance constraint is satisfied. Clock instance constraint satisfaction is a property of clock instance value assignments.

Definition 5.2.7 (Clock instance constraint satisfaction)**.** Let V_{CI} be a set of clock instances, $z \in \mathcal{Z}(V_{CI})$ a clock instance constraint over V_{CI}, and ν a clock instance value assignment over V_{CI}. Then, ν *satisfies* z, written $\nu \models z$, if and only if $z[\nu(ci)/ci] \equiv true$.

5.2.3 Locking Edges and Application Indicators

In timed graphs, locking of nodes and edges is done via the creation and deletion of additional edges, called *locking edges*. Elements of a configuration are locked by applying start rules and unlocked by applying end rules to prevent a concurrent access. There are separate locking edges for reading nodes, reading edges, writing nodes, and writing edges. All these locking edges also have respective edge types in a type graph.

To guarantee that the match of an end rule conforms with an earlier match of a start rule, the match of each start rule has to be remembered in a configuration. This is done by adding designated nodes, called *application indicators*, when applying a start rule. Such an application indicator has an edge to each node in the match of the start rule. These nodes indicate the application scope of the durative rule that induced the start rule. When applying the end rule, we can ensure conformity with the start rule's match by requiring the same application scope. To properly indicate to which durative rule an application indicator belongs, its type is distinct for each durative rule.

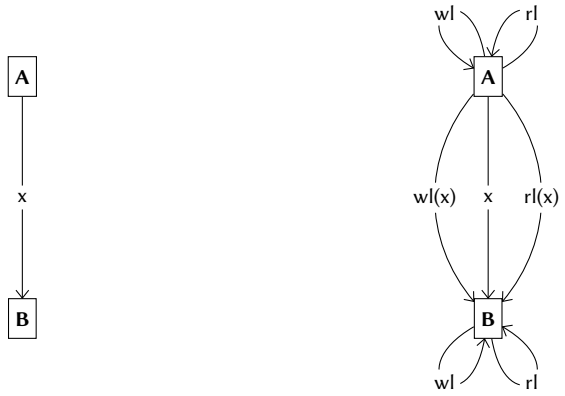
All induced rules mentioned earlier are typed via a type graph of the TGTS formalism. This type graph is induced by the type graph of the durative graph transformation system under consideration and contains equivalent types and edge types. In addition to these types and edge types, it needs types and edge types to allow for the creation and deletion of locking edges and application indicators. These types and edge types provide the basis for realizing the locking mechanism and indicating the ongoing application of a durative rule in a configuration.

The next paragraphs explain the construction of an induced TGTS type graph via examples, with special attention paid to locking edges and application indicators. Its formal definition is given afterwards.

To conveniently refer to locking edge types, we use the functions $rlnode : V_{TG} \to E_{TG}$, $wlnode : V_{TG} \to E_{TG}$, $rledge : E_{TG} \to E_{TG}$, and $wledge : E_{TG} \to E_{TG}$. Each node type nt has two locking edge types, $rlnode(nt)$ and $wlnode(nt)$, as self edges in the TGTS type graph. For every edge type et that is no locking edge type itself, there are locking edges types $rledge(et)$ and $wledge(et)$ adjacent to the same source and target node types. An example for the inducement of locking edge types is shown in Figure 5.7.

In a configuration, an edge of type $rlnode(nt)$ depicts an obtained read lock for a node that has the type nt, and an edge of type $wlnode(nt)$ depicts an obtained write lock. Similarly, an edge of type $rledge(et)$ depicts an obtained read lock for an edge that has the type et, and an edge of type $wledge(et)$ depicts an obtained write lock.

Application indicators are used to indicate the ongoing execution of a durative graph transformation in a configuration. Their outgoing edges, called *application indicator edges*, mark the match of the durative graph transformation rule that has been used, i.e., the subgraph of the configuration that is changed by the transformation. For a durative graph transformation rule with the name $name$, the node type of the application indicators it instantiates is given by $aiType(name)$; the edge type of an edge connecting the application indicator with a node v is given by $aiEdgeType(v)$.

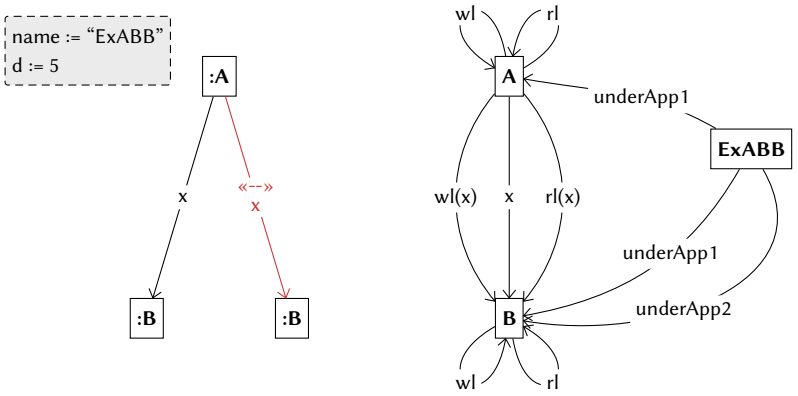


(a) A DGTS type graph

(b) Its induced TGTS type graph (incomplete, shows only locking edge types)

Figure 5.7: Inducement of locking edge types

The induced TGTS type graph shown in Figure 5.7(b) is not yet complete. In addition to locking edges, it contains node types for application indicators: there is a node type for each durative rule and edge types from this node type to every other node type used within the rule, i.e., the TGTS type graph depends on the set of durative rules contained in the durative graph transformation system. Figure 5.8 shows the inducement of a TGTS type graph for a durative graph transformation system containing only a single rule.



(a) A durative rule with the name `ExABB` and a duration of 5

(b) An induced TGTS type graph for a DGTS containing only the rule `ExABB`

Figure 5.8: Inducement of a TGTS type graph

Application indicator edges to different nodes of the same type have to be distinguishable to guarantee that the match of the start rule can be inferred from them. This is important because the end rule of a durative rule is supposed to match the same nodes and edges as its start rule did. If there was no possibility to distinguish application indicator edges, the end rule would still match the correct set of nodes but not necessarily with a correct structure, i.e., it might not necessarily match the correct set of edges. Therefore, application indicator edges to nodes of the same type are individualized with numbers that are distinct within the rule. Consequently, the TGTS type graph contains multiple application indicator edge types from a single application indicator type to a single node type nt if the application indicator's rule specifies multiple nodes of type nt, see Figure 5.8(b).

Now, we give a formal definition for the induced TGTS type graph. Such a type graph is needed by the various kinds of rules of the TGTS formalism. In addition to the types and edge types defined by the DGTS type graph, the induced TGTS type graph provides locking edge types and types for application indicators and its edges.

Definition 5.2.8 (Induced TGTS type graph)**.** Let $\mathcal{TG}$ be a type graph of a durative graph transformation system. The *induced TGTS type graph* of $\mathcal{TG}$ is a type graph TG where

- $V_{TG} = V_{\mathcal{TG}} \cup V_{AI}$,
- $E_{TG} = E_{\mathcal{TG}} \cup E_{AI} \cup E_{RL.node} \cup E_{WL.node} \cup E_{RL.edge} \cup E_{WL.edge}$,
- $|V_{AI}| = |\mathcal{DR}| \wedge$
  $\forall \mathcal{D} \in \mathcal{DR} : aiType(name) \in V_{AI}$,
- $|E_{AI}| = \sum_{\mathcal{D} \in \mathcal{DR}} |V_{G,L}| \wedge$
  $\forall \mathcal{D} \in \mathcal{DR} : \forall vt \in V_{\mathcal{TG}} : \exists E_{AI'} \subseteq E_{AI} :$
  $|E_{AI'}| = |\{v \in V_{G,L} | type(v) = vt\}| \wedge$
  $src(E_{AI'}) = aiType(name) \wedge tgt(E_{AI'}) = vt$,
- $|E_{RL.node}| = |V_{\mathcal{TG}}| \wedge$
  $\forall vt \in V_{\mathcal{TG}} : rlnode(vt) \in E_{RL.node} \wedge$
  $src \circ rlnode(vt) = tgt \circ rlnode(vt) = vt$,
- $|E_{WL.node}| = |V_{\mathcal{TG}}| \wedge$
  $\forall vt \in V_{\mathcal{TG}} : wlnode(vt) \in E_{WL.node} \wedge$
  $src \circ wlnode(vt) = tgt \circ wlnode(vt) = vt$,
- $|E_{RL.edge}| = |E_{\mathcal{TG}}| \wedge$
  $\forall et \in E_{\mathcal{TG}} : rledge(et) \in E_{RL.edge} \wedge$
  $src \circ rledge(et) = src(et) \wedge tgt \circ rledge(et) = tgt(et)$, and
- $|E_{WL.edge}| = |E_{\mathcal{TG}}| \wedge$
  $\forall et \in E_{\mathcal{TG}} : wledge(et) \in E_{WL.edge} \wedge$
  $src \circ wledge(et) = src(et) \wedge tgt \circ wledge(et) = tgt(et)$.

There is exactly one application indicator for each durative rule, and each node in its LHS has an own application indicator edge type, which connects its type with the application indicator. The latter is so that application indicator edges to different nodes of the same type are distinguishable, which ensures a correct match of the end rule.

For each node type, the induced TGTS type graph has two locking edge types as self edges, one for reading and one for writing. There are also two locking edge types for each edge type that is no locking edge type itself. These locking edge types have the same source and target node as the edge type.

5.2.4 Timed Graph Transformation Rules

The induced start rule and end rule of a durative graph transformation rule are both defined on top of a timed graph transformation rule. A timed graph transformation rule is similar to an ordinary graph transformation rule, except that it operates on timed graphs instead of ordinary graphs. Finding a match for a timed graph transformation rule works exactly in the same way as finding a match for an ordinary graph transformation rule. In addition to the LHS, RHS, and rule morphism, a timed graph transformation rule specifies a timed guard as well as a set of clock instances to be reset. The time guard is a clock instance constraint that is expressed via clock instances contained in the rule's LHS. For the rule to be applicable, the time guard has to be evaluated to true. When the rule is applied, those clock instances specified in the set are reset to zero.

Definition 5.2.9 (Timed graph transformation rule)**.** A *timed graph transformation rule* $tr = (L, R, r, \mathcal{N}, z, V_{res})$ consists of

- two timed graphs L and R,
- an injective rule morphism $r : L \to R$ with $r(V_{CI,L}) = V_{CI,R}$ and $|V_{CI,L}| = |V_{CI,R}|$,
- a set of NACs $\mathcal{N}$ where each NAC is a tuple $(N, n) \in \mathcal{N}$ with $n : L \to N$,
- a clock instance constraint $z \in \mathcal{Z}(V_{CI,L})$, called *time guard*, and
- a set of clock instances $V_{res} \subseteq V_{CI,R}$.

This timed graph transformation rule is further specialized by the induced start and end rule. Intuitively, the induced start rule serves two purposes. First, it adds information about the execution of the durative rule into the host graph. This is needed for the annotation of time and for the end rule to find a match that corresponds to the match of the start rule. Finding a corresponding match is important because together both rules are supposed to represent the application interval of the durative graph transformation. With a wrong match there would be no meaningful interpretation for the application of a durative rule. Second, it adds locking edges into the host graph such that subsequent rules do not match if they access the same elements in a conflicting manner.

Definition 5.2.10 (Induced start rule)**.** Let $\mathcal{D} = (L_\mathcal{D}, R_\mathcal{D}, r_\mathcal{D}, name, d)$ be a durative rule. The *induced start rule* of $\mathcal{D}$ is a timed rule $sr = (L, R, r, \mathcal{N}, z, V_{res})$ where

- $V_{G,L} = V_{L_\mathcal{D}} \wedge E_{G,L} = E_{L_\mathcal{D}}$,
- $V_{G,R} = V_{L_\mathcal{D}} \cup \{ai\} \wedge type(ai) = aiType(name) \wedge E_{G,R} = E_{L_\mathcal{D}} \cup \{e | src(e) = ai \wedge tgt(e) \in V_{G,R} \setminus \{ai\} \wedge type(e) = aiEdgeType \circ tgt(e)\} \cup E_{RL.node,R} \cup E_{WL.node,R} \cup E_{RL.edge,R} \cup E_{WL.edge,R}$,
- $V_{CI,L} = V_{CI,R} = \emptyset \wedge E_{CI,L} = E_{CI,R} = \emptyset$,
- $i_L : L_\mathcal{D} \to L$ is the identity morphism on $L_\mathcal{D} \wedge r$ is total,
- $\mathcal{N} = \mathcal{N}_{WL.node} \cup \mathcal{N}_{RL.node} \cup \mathcal{N}_{WL.edge} \cup \mathcal{N}_{RL.edge}$,
- $z = \emptyset \wedge V_{res} = \emptyset$,
- $E_{RL.node,R} = \{le | \exists v \in V_{G,L} : src(le) = tgt(le) = r(v) \wedge type(le) = rlnode \circ type(v)\}$,
- $E_{WL.node,R} = \{le | \exists v \in V_{G,L} \setminus i_L \circ \mathrm{dom}(r_\mathcal{D}) : src(le) = tgt(le) = r(v) \wedge type(le) = wlnode \circ type(v)\}$,
- $E_{RL.edge,R} = \{le | \exists e \in E_{G,L} : src(le) = src \circ r(e) \wedge tgt(le) = tgt \circ r(e) \wedge type(le) = rledge \circ type(e)\}$,
- $E_{WL.edge,R} = \{le | \exists e \in E_{G,L} \setminus i_L \circ \mathrm{dom}(r_\mathcal{D}) : src(le) = src \circ r(e) \wedge tgt(le) = tgt \circ r(e) \wedge type(le) = wledge \circ type(e)\}$,
- $\mathcal{N}_{WL.node} = \{(N, n) | \exists v \in V_{G,L} : V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge src(ne) = tgt(ne) = n(v) \wedge type(ne) = wlnode \circ type(v) \wedge n \text{ is injective}\}$,
- $\mathcal{N}_{RL.node} = \{(N, n) | \exists v \in V_{G,L} \setminus i_L \circ \mathrm{dom}(r_\mathcal{D}) : V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge src(ne) = tgt(ne) = n(v) \wedge type(ne) = rlnode \circ type(v) \wedge n \text{ is injective}\}$,
- $\mathcal{N}_{WL.edge} = \{(N, n) | \exists e \in E_{G,L} : V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge src(ne) = src \circ n(e) \wedge tgt(ne) = tgt \circ n(e) \wedge type(ne) = wledge \circ type(e) \wedge n \text{ is injective}\}$, and
- $\mathcal{N}_{RL.edge} = \{(N, n) | \exists e \in E_{G,L} \setminus i_L \circ \mathrm{dom}(r_\mathcal{D}) : V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge src(ne) = src \circ n(e) \wedge tgt(ne) = tgt \circ n(e) \wedge type(ne) = rledge \circ type(e) \wedge n \text{ is injective}\}$.

The LHS of the induced start rule is the same as the LHS of the durative rule. Its RHS is a copy of the LHS with an additional node *ai*, which is its application indicator, additional application indicator edges from *ai* to all other nodes in the RHS, and additional locking edges $E_{RL.node,R}$, $E_{WL.node,R}$, $E_{RL.edge,R}$, and $E_{WL.edge,R}$. Intuitively, the existence of an application indicator in the host graph indicates the

application of its durative rule, its application indicator edges mark the subgraph that is being changed by the rule application, and locking edges in the host graph indicate whether read or write access to specific nodes and edges is locked.

The start rule shall not delete any node or edge. Therefore, the rule morphism r (restricted to graph nodes and edges) is total. According to the definition of a timed graph transformation rule, it is also injective, and as a consequence of its LHS and RHS, unique (up to isomorphism). This allows for a deterministic inducement of start rules.

The sets of clock instances, clock instance edges, time guards, and clock instance resets are empty because a start rule does not add a clock instance measuring the execution time itself. Instead, the addition of a clock instance for the execution of a durative rule is done by a clock instance rule, which is presented in Section 5.2.5.

The remainder conditions implement the locking functionality. The locking edge sets $E_{RL.node,R}$ and $E_{RL.edge,R}$ specify the creation of a read lock for every required node or edge, respectively. The sets $E_{WL.node,R}$ and $E_{WL.edge,R}$ specify the creation of a write lock for every node or edge that is deleted according to the syntax of the durative rule, i.e., that is not contained in $i_L \circ \mathrm{dom}(r_{\mathcal{D}})$. The last four sets $\mathcal{N}_{WL.node}$, $\mathcal{N}_{RL.node}$, $\mathcal{N}_{WL.edge}$, and $\mathcal{N}_{RL.edge}$ define NACs that are used to check for the existence of locking edges. For each read lock, there is a NAC that forbids the existence of a write lock and vice versa.

If the host graph contains parallel edges, the locking mechanism operates more restrictive than necessary. If any one of multiple parallel edges is accessed, this has the effect of locking all those parallel edges. Fortunately, a less restrictive locking of parallel edges can be achieved without changing the semantics: graphs that support parallel edges can simply be simulated by graphs that do not support them, as done in [Bon+07]. Thus, we can preprocess a durative graph transformation system employing parallel edges by mapping it into an equivalent durative graph transformation system without parallel edges.

The purpose of the induced end rule is to actually realize the transformation that is syntactically specified by the durative rule and to remove those locking edges that have been created by the start rule.

Definition 5.2.11 (Induced end rule)**.** Let $\mathcal{D} = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}}, name, d)$ be a durative rule. The *induced end rule* of $\mathcal{D}$ is a timed rule $er = (L, R, r, \mathcal{N}, z, V_{res})$ where

- $V_{G,L} = V_{L_{\mathcal{D}}} \cup \{ai\} \wedge type(ai) = aiType(name) \wedge E_{G,L} = E_{L_{\mathcal{D}}} \cup$
  $\{e | src(e) = ai \wedge tgt(e) \in V_{G,L} \setminus \{ai\} \wedge type(e) = aiEdgeType \circ tgt(e)\} \cup$
  $E_{RL.node,L} \cup E_{WL.node,L} \cup E_{RL.edge,L} \cup E_{WL.edge,L}$,
- $V_{G,R} = V_{R_{\mathcal{D}}} \wedge E_{G,R} = E_{R_{\mathcal{D}}}$,
- $V_{CI,L} = V_{CI,R} = \{ci\} \wedge E_{CI,L} = \{(ci, ai)\} \wedge E_{CI,R} = \emptyset$,
- $i_L : L_{\mathcal{D}} \to L$ is a subgraph isomorphism $\wedge$
  $i_R : R_{\mathcal{D}} \to R$ is the identity morphism on $R_{\mathcal{D}} \wedge$
  $r|_{\{V_{G,L}, E_{G,L}\}} = i_R \circ r_{\mathcal{D}} \circ i_L^{-1} \wedge r|_{\{V_{CI,L}\}}$ is total,

- $\mathcal{N} = \emptyset$,
- $z = \{ci \geq d\} \wedge V_{res} = \emptyset$,
- $E_{RL.node,L} = \{le | \exists v \in V_{G,L} : src(le) = tgt(le) = v \wedge type(le) = rlnode \circ type(v)\}$,
- $E_{WL.node,L} = \{le | \exists v \in V_{G,L} \setminus \mathrm{dom}(r) : src(le) = tgt(le) = v \wedge type(le) = wlnode \circ type(v)\}$,
- $E_{RL.edge,L} = \{le | \exists e \in E_{G,L} : src(le) = src(e) \wedge tgt(le) = tgt(e) \wedge type(le) = rledge \circ type(e)\}$, and
- $E_{WL.edge,L} = \{le | \exists e \in E_{G,L} \setminus \mathrm{dom}(r) : src(le) = src(e) \wedge tgt(le) = tgt(e) \wedge type(le) = wledge \circ type(e)\}$.

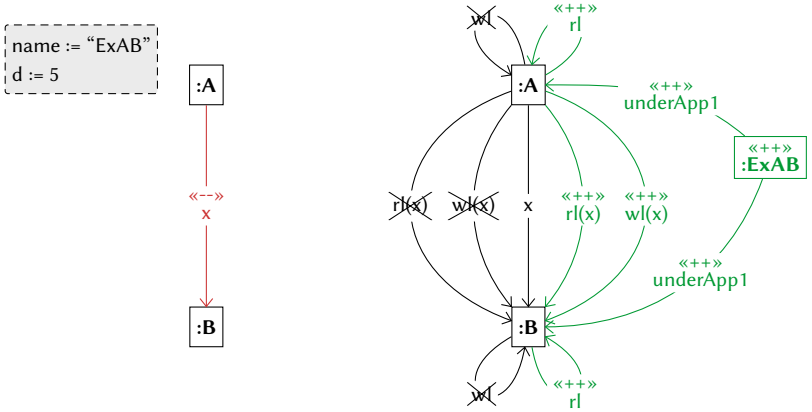
The LHS of the induced end rule is defined analogously to the RHS of the induced start rule, i.e., it corresponds to the LHS of the durative rule plus an application indicator node, application indicator edges, and locking edges. The RHS of the induced end rule is the same as the RHS of the durative rule. Therefore, the application of the end rule removes the application indicator, the application indicator edges, and the locking edges that were created when the start rule was applied. The rule morphism r is defined in conformity with $r_{\mathcal{D}}$, i.e., the end rule realizes the graph transformation syntactically specified by the durative rule.

The end rule also includes a time guard on the value of clock instance ci, which guarantees that the proper amount of time is consumed before the end rule is applied. Note that ci, which is connected via only one edge to ai, is not removed by the end rule. This is because timed graph transformations may neither add nor remove clock instances to or from a timed graph. Adding clock instances is subject to clock instance rules and removing them is subject to a singleton clock instance removal rule. Both are covered in Section 5.2.5.

Figure 5.9 shows an example of a durative graph transformation rule and its induced start and end rule. The durative rule is named `ExAB` and specifies the removal of an edge `x` during an interval of 5 time units, see Figure 5.9(a). Its induced start rule specifies an application indicator node `ExAB` to be created, along with two application indicator edges, one to the node of type `A` and one to the node of type `B`, see Figure 5.9(b). Here, the target nodes of both application indicator edges are of different type. Therefore, both edges are labeled with `underApp1`. If both target nodes were of the same type, one of the application indicator edges would have been labeled with `underApp2` instead.

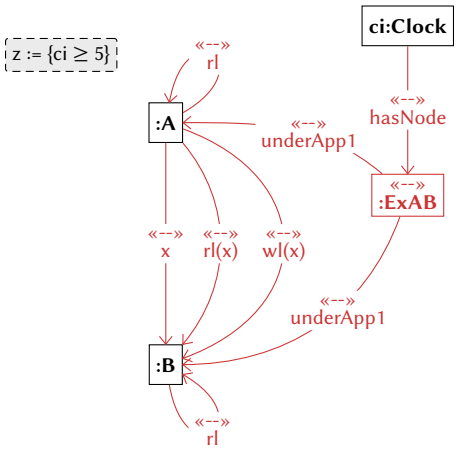
Since both of these nodes are preserved in the durative rule, only their read access is locked by an attached creation edge `rl` in the start rule. For the edge, which is deleted in the durative rule, both its read and write access are locked by attached creation edges `rl(x)` and `wl(x)`. Furthermore, forbidden edges allow the application of the start rule only if write access to the preserved nodes and both write and read access to the deletion edge are not locked.

The end rule deletes the application indicator node, its adjacent edges, and all locking edges that the start rule creates, see Figure 5.9(c). The application indicator



(a) A durative rule with the name ExAB and a duration of 5

(b) Its induced start rule



(c) Its induced end rule

Figure 5.9: Inducement of start and end rule

edges ensure that the match of the end rule corresponds to the match of the start rule when the end rule is applied. Note that if there were multiple nodes of the same type in the durative rule, the application indicator edges being added to these nodes by the induced start rule would be of different edge types to ensure a correct match for the end rule.

The time guard $z = \{ci \geq 5\}$ is a condition for the rule's application. It guarantees that the rule cannot be applied before 5 time units have been passed on the clock instance ci. In the graphical representation, the clock instance the time guard refers to can be identified via its object name. In the formal syntax, we can simply use variable names to make clear to which clock instance a time guard refers

to. Therefore, we did not define such object names in the syntax of timed graphs or timed graph transformation rules.

The time guard only guarantees that the induced end rule is not applied too early. We also need to ensure that it is not applied too late. More precisely, we need to enforce the application of the rule as soon as its time guard is fulfilled. In order to do this, we use invariant rules, which are formally explained in the next section.

5.2.5 Clock Instance and Invariant Rules

In Section 5.2.2, we explained that clock instances pertain to parts of a configuration – as opposed to timed automata, where clocks pertain to the complete automaton. Since it is impossible to decide at design time how many clock instances a timed graph transformation system needs, clock instances have to be instantiable. Their instantiation could simply be supported by allowing timed rules to add clock instances; however, the designers of the timed graph transformation formalism decided to put this functionality into separate rules, called clock instance rules. This decision can easily be explained by looking at the implementation of invariants.

Invariants are realized as invariant rules in the TGTS formalism. Invariant rules state how long a specific structure is allowed to exist. This structure represents a part of a configuration. It does not matter which transformations resulted in this configuration or whether it is the result of a single or multiple transformations. Since an invariant does not care which timed transformation led to a configuration, why should a clock instance care? By putting the creation and deletion of clock instances into separate rules, the existence of a clock instance in a configuration depends entirely on its structure, not on what happened before.

Clock instance rules identify those parts of a configuration that clock instances pertain to. They work similar to timed rules; however, they are specified such that they do not delete anything and create only a single clock instance as well as edges adjacent to this clock instance. To prevent the creation of more than one clock instance to the same part of the configuration, the rule specifies a NAC that is identical to its RHS.

Definition 5.2.12 (Clock instance rule)**.** A *clock instance rule* $cr = (L, R, r, \mathcal{N})$ consists of two timed graphs L and R, a rule morphism $r : L \to R$, and a negative application condition $(N, n) \in \mathcal{N}$ where

- $V_{G,L} = V_{G,R} \wedge E_{G,L} = E_{G,R}$,
- $V_{CI,L} = \emptyset \wedge |V_{CI,R}| = 1 \wedge |E_{CI,R}| \geq 1$,
- r is total, and
- $N = R \wedge n = r \wedge |\mathcal{N}| = 1$.

In early variants of the timed graph transformation formalism [Neu07; Hir08], clock instance rules have been derived from timed rules and invariant rules. Later variants, such as [SHS11; Eck+13], also allow their explicit specification. Both variants

are suitable for a semantics of durative rules. Here, we follow the latter approach, i.e., we explicitly define the induced clock instance rule for a given durative rule.

An induced clock instance rule has only an application indicator node in its LHS. Therefore, it attaches a clock instance only if a start rule that has been induced by the same durative rule has been applied before. Since the application indicator is typed via the name of the durative rule, there is exactly one induced clock instance rule for each durative rule.

Definition 5.2.13 (Induced clock instance rule)**.** Let $\mathcal{D} = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}}, name, d)$ be a durative rule. The *induced clock instance rule* of $\mathcal{D}$ is a rule $cr = (L, R, r, \mathcal{N})$ where

- $V_{G,L} = V_{G,R} = \{ai\} \wedge type(ai) = aiType(name) \wedge E_{G,L} = E_{G,R} = \emptyset$ and
- $V_{CI,L} = \emptyset \wedge E_{CI,L} = \emptyset \wedge V_{CI,R} = \{ci\} \wedge E_{CI,R} = \{(ci, ai)\}$.

The operational semantics in Section 5.2.6 is designed such that all applicable clock instance rules are applied immediately after a timed rule has been applied. Upon application, an induced clock instance rule attaches a clock instance to an application indicator node that is not yet connected to a clock instance. Since start rules create application indicator nodes, a clock instance rule creates a clock instance directly after a start rule has been applied.

If the part of the configuration that the clock instance pertains to is no longer present, the clock instance needs to be removed as well. This is the case when an end rule is applied because each application of an end rule removes an application indicator. Removing the clock instance is subject to a clock instance removal rule. For a given set of clock instance rules, a clock instance removal rule can be derived automatically. It has a single clock instance as its LHS and an empty RHS. In addition, it specifies the RHSs of all clock instance rules as NACs. As a consequence, the clock instance removal rule deletes a clock instance if the part of the configurations that the clock instance pertains to is no longer present. There is only one clock instance removal rule for the complete timed graph transformation system.

Definition 5.2.14 (Clock instance removal rule)**.** Let CR be a set of clock instance rules. A *clock instance removal rule* for CR is a rule $rr_{CR} = (L, R, r, \mathcal{N})$ where

- $V_{G,L} = V_{G,R} = \emptyset \wedge E_{G,L} = E_{G,R} = \emptyset$,
- $V_{CI,L} = \{ci\} \wedge V_{CI,R} = \emptyset \wedge E_{CI,L} = E_{CI,R} = \emptyset$,
- $\mathcal{N} = \{(N, n) | \exists cr = (L_{cr}, R_{cr}, r_{cr}, \mathcal{N}_{cr}) \in CR : N = R_{cr} \wedge n : L \to N$ with $n(ci) \in V_{CI,R_{cr}}\}$.

Invariant rules, which state how long a specific part of a configuration is allowed exist, specify only an LHS. There is no need for an RHS, because invariant rules do not perform transformations. The LHS of an invariant rule contains an arbitrary number of graph node and edges, but exactly one clock instance. They also specify a clock instance constraint.

Definition 5.2.15 (Invariant rule)**.** An invariant rule $ir = (L, z)$ consists of a timed graph L with $|V_{CI,L}| = 1$ and a clock instance constraint $z \in \mathcal{Z}(V_{CI,L})$.

Whenever the LHS matches to the configuration via a match m, the clock instance constraint has to be fulfilled for the image of that clock instance under m. This results in a timed graph transformation rule being applied immediately before a clock instance constraint of an invariant rule turns false. Its application prevents the invariant rule's match from existing and thus the clock instance constraint from being evaluated. If no rule can be applied that destroys an invariant rule's match and time cannot elapse without violating its clock instance constraint, a *time-stopping deadlock* occurs.

Remember that an induced end rule specifies a time guard $z = \{ci \geq d\}$, which is a condition for its application. This time guard guarantees the consumption of at least d time units since the clock instance contained in the match has been created. To correctly represent the end of a durative rule's execution, we have to ensure that the end rule is indeed applied after d time units instead of being postponed arbitrarily. Unfortunately, the time guard only guarantees that it is not applied earlier. To ensure that the end rule is applied after d time units have passed, a durative rule also induces an invariant rule. It specifies an application indicator *ai* and a clock instance *ci* as its only nodes and $z = \{ci \leq d\}$ as the constraint to be fulfilled whenever the LHS is matched.

Definition 5.2.16 (Induced invariant rule)**.** Let $\mathcal{D} = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}}, name, d)$ be a durative rule. The *induced invariant rule* of $\mathcal{D}$ is a rule $ir = (L, z)$ where

- $V_{G,L} = \{ai\} \wedge type(ai) = aiType(name) \wedge E_{G,L} = \emptyset$,
- $V_{CI,L} = \{ci\} \wedge E_{CI,L} = \{(ci, ai)\}$, and
- $z = \{ci \leq d\}$.

Intuitively, each match of an induced invariant rule's LHS indicates that an application of a durative rule is taking place. At every match there is a distinct application indicator that was created by a start rule induced by the same durative rule as the invariant rule. There is also a clock instance measuring the elapsed time since the application of the start rule. Since an invariant rule forbids the existence of an LHS match such that its clock instance constraint is unfulfilled, a timed graph transformation rule has to be applied no later than the instant the constraint turns false. This timed rule has to delete the application indicator *ai* to destroy the LHS match of the invariant rule. Since the types of their application indicator nodes have to match, the timed rule has to be an end rule that is induced by the same durative rule as the invariant rule. Therefore, the invariant rule guarantees that the end rule is indeed applied after d time units.

Figure 5.10 shows the induced clock instance rule and the induced invariant rule for the durative rule of Figure 5.9(a). The structure of the durative rule's LHS is not relevant for the inducement of these two rules. The induced clock instance rule depends only on the name of the durative rule, see Figure 5.10(a), and the induced

(a) Induced clock instance rule of a durative rule with the name ExAB

(b) Induced invariant rule of a durative rule with the name ExAB and a duration of 5

Figure 5.10: Inducement of clock instance and invariant rule

invariant rule depends only on the name and duration of the durative rule, see Figure 5.10(b).

5.2.6 Operational Semantics

The semantics of a timed graph transformation system is based on the definitions of the timed graph transformation rule, the clock instance rule, the clock instance removal rule, and the invariant rule. Before defining the semantics of a timed graph transformation system, we need a definition of a timed graph transformation system itself.

Definition 5.2.17 (Timed graph transformation system)**.** A *timed graph transformation system* $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ consists of

- a type graph TG,
- an initial timed graph TiG_0,
- a set of timed graph transformation rules TR,
- a set of invariant rules IR, and
- a set of clock instance rules CR.

Note that the definition of a timed graph transformation system does not include a clock instance removal rule in the tuple. For a given set of clock instance rules CR, the clock instance removal rule rr_{CR} is deterministically implied, cf. Definition 5.2.14.

As stated in Definition 5.2.2, a durative graph transformation system is an initial typed graph and a set of durative graph transformation rules. Its semantics is given by a timed graph transformation system whose timed graph transformation rules TR, invariant rules IR, and clock instance rules CR are all induced by durative rules.

Definition 5.2.18 (Induced timed graph transformation system)**.** Given a durative graph transformation system $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR})$ with initial typed graph $G_0^T = (G_0, type_0)$ and $G_0 = (V_G, E_G, src, tgt)$, the *induced timed graph transformation system* of $\mathcal{DS}$ is a timed graph transformation system $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ where

- TG is the type graph induced by $\mathcal{TG}$,
- TR are the timed graph transformations induced by $\mathcal{DR}$,
- IR are the invariant rules induced by $\mathcal{DR}$,
- CR are the clock instance rules induced by $\mathcal{DR}$, and
- $TiG_0 = (G'_0, type'_0)$ is an initial timed graph with $G'_0 = (V_G, \emptyset, E_G, \emptyset, src, tgt)$, $type'_0 = i \circ type_0$, and $i : TG \to TG'$ is a subgraph isomorphism.

When modeling with durative rules, all timed graph transformation rules TR, invariant rules IR, and clock instance rules CR are induced by durative graph transformation rules. Usually, there are no rules that are not induced by durative rules. Modeling any rules of the timed graph transformation system formalism directly could break the applicability of an end rule and hence might cause time-stopping deadlocks.

As a basis for our operational semantics we use the notion of a configuration, which we defined in Section 5.2.2. Intuitively, a configuration consists of a timed graph and an assignment of values to the clock instances of the timed graph. Here, we need the notion of an initial configuration.

Although not the case for the initial configuration of an induced timed graph transformation system, an initial configuration might have matching clock instance rules in general. Remember that clock instance rules are used to specify subgraphs of the host graph that clock instances pertain to. For each match of a clock instance rule there should be exactly one clock instance in the host graph. Consequently, an initial timed graph should also have exactly one clock instance for each match of a clock instance rule. The correct placement of clock instances is later maintained by a regular application of all available clock instance rules as well as the clock instance removal rule (as we will see later in Definition 5.2.21). An initial configuration is a configuration that resets all existing clock instances.

Definition 5.2.19 (Initial configuration)**.** Let $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ be a timed graph transformation system. An *initial configuration* of $\mathcal{TS}$ is a configuration $\langle TiG_0, \nu_0 \rangle$ with $TiG_0 = (G_0, type_0)$ and $G_0 = (V_G, V_{CI}, E_G, E_{CI}, src, tgt)$ where

- for all clock instance rules $cr \in CR$ and all matches $m : L_{cr} \to TiG_0$ there exists a co-match $m^* : R_{cr} \to TiG_0$ such that $m^* \circ r_{cr} = m$,
- $V_{CI} = \{ci | \exists cr \in CR : \exists m^* : R_{cr} \to TiG_0 : m^*(V_{CI,R}) = \{ci\}\}$, and
- for all clock instances $ci \in V_{CI} : \nu_0(ci) \mapsto 0$.

For the general case, the above definition guarantees a correct placement of clock instances in the initial timed graph. The first condition ensures that there is a clock instance at each match of a clock instance rule, and the second condition ensures that there are no further clock instances in the initial timed graph. The last condition assigns zero to all clock instances.

In case of an induced timed graph transformation system, the initial timed graph does not contain any application indicators. Since each clock instance rule of an induced timed graph transformation system has an application indicator in its LHS, none of the clock instance rules matches and, consequently, the initial timed graph contains no clock instances. Therefore, ν_0 is an empty function in this case.

For defining the operational semantics of a timed graph transformation system, we also need to be able express that the invariant of a timed graph is satisfied. More precisely, we need to express whether the clock instance value assignment of a configuration fulfills the conjunctive invariant defined by the clock instance constraints of all invariant rules.

Definition 5.2.20 (Conjunctive invariant of a timed graph)**.** The *conjunctive invariant* $I(TiG)$ of a timed graph TiG is defined as

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

where $I_{ir}(TiG)$ is defined for a given timed graph TiG and a given invariant rule $ir = (L, z) \in IR$ with matches $m_i : L \to TiG$ for $i = 1, \ldots, k$ as

$$I_{ir}(TiG) = \bigwedge_{i=1,\ldots,k} z[\forall ci \in V_{CI,L} : m_i(ci)/ci].$$

Now, we have everything we need to define the operational semantics of a timed graph transformation system. The operational semantics defines two kinds of transitions: delay transitions and action transitions. Its definition is similar to the operational semantics defined for UPPAAL timed automata, cf. [BY04].

Definition 5.2.21 (Operational semantics of a timed graph transformation system)**.** Let $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ be a timed graph transformation system. The *operational semantics* of $\mathcal{TS}$ is defined by a transition system whose states are configurations $\langle TiG, \nu \rangle$. Execution starts in the initial configuration $\langle TiG_0, \nu_0 \rangle$ and transitions are defined by the following rules:

1. $\langle TiG, \nu \rangle \stackrel{\delta}{\Rightarrow} \langle TiG, \nu + \delta \rangle$ if $(\nu + \delta) \models I(TiG)$ for $\delta \in \mathbb{R}^+$. (delay trans.)

2. $\langle TiG, \nu \rangle \stackrel{tr,m}{\Longrightarrow} \langle TiG''', \nu' \rangle$ if $\nu \models z[\forall ci \in V_{CI,L} : m(ci)/ci]$ for a timed graph transformation rule $tr = (L, R, r, \mathcal{N}, z, V_{res}) \in TR$ and a match $m : L \to TiG$ where

   - TiG' has been derived by applying tr at m to TiG,
   - TiG'' has been derived by applying rr_{CR} to TiG',
   - TiG''' has been derived by applying all $cr \in CR$ in any order to TiG'', and
   - $\nu' = \nu[V_{res} \mapsto 0]$ with $\nu' \in I(TiG''')$. (action trans.)

Delay transitions do not apply any rules. Instead, they increase the values of all clock instances synchronously. As a condition for the transition, the new clock instance value assignment has to satisfy $I(G)$, which is the conjunction of all

invariant clock instance constraints. While firing a delay transition in a configuration with no clock instance is possible, it has no effect, i.e., it produces a self edge in the state space.

Action transitions are defined by the application of timed graph transformation rules. To create a successor configuration, each application of a timed graph transformation rule is accompanied by the deletion of clock instances through applications of the clock instance removal rule as well as the creation of clock instances through the application of clock instance rules. In the presence of durative rules, a clock instance is created after applying an induced start rule and deleted after applying an induced end rule.

5.3 Properties of Durative Graph Transformation Rules

In this section we explain and prove some properties that support that the DGTS formalism works as a modeler might expect. One of these properties concerns the application of durative rules. It states how the application of a durative graph transformation rule correlates to the transformation of an untimed graph transformation rule and explains the different transitions caused by its application. This property is formalized in Section 5.3.1.

Two other properties concern the termination of durative rules and the possible interleavings of start and end transformations when multiple durative rules are applied concurrently. The first property states that each application of a durative rule terminates properly. The second one ensures that each interleaving of two pairs of start and end transformations results in the same configuration when finished. Both properties are formalized in Section 5.3.2.

5.3.1 Correspondence of a Durative Graph Transformation

On a syntactical level, durative graph transformation rules seem like ordinary untimed graph transformation rules with additional values for their application duration. A modeler thus has an intuitive assumption how the application of such a rule works, especially if the modeler is already acquainted with graph transformation systems.

The semantics, however, is rather complex. A modeler should not need to be familiar with timed graph transformation systems, but be spared from thinking about clock instances and the many induced rules. Therefore, it is important that the system works as a modeler expects.

During the execution of a durative graph transformation, the system is in a configuration that neither represents the configuration before nor the configuration after its execution. Instead, it contains application indicators and locking edges, which have no meaning in the DGTS model. For ease of comprehensibility, it is vital that the complete durative graph transformation results in a configuration that leaves no application indicators or locking edges behind and corresponds to a successor

configuration that would have been created in an untimed GTS formalism – in our case, the SPO approach. Fortunately, the DGTS semantics fulfills this property.

Before formalizing this property in Proposition 5.3.2, we define a lemma stating that after applying the induced start rule of a durative rule and letting its duration pass, its induced end rule can be applied. This lemma is used during the proof of Proposition 5.3.2.

Lemma 5.3.1 (Applicability of induced end rule)**.** *Let $\mathcal{D}$ be a durative rule with $\mathcal{D} = (L_\mathcal{D}, R_\mathcal{D}, r_\mathcal{D}, name, d)$, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, \emptyset, E_G, \emptyset)$, and $\nu = \emptyset$ a clock instance value assignment. The induced start and end rule of $\mathcal{D}$ are denoted by sr and er, respectively.*

If there is a match $m : L_{sr} \to TiG$ and an action transition $\langle TiG, \nu \rangle \overset{sr,m}{\Longrightarrow} \langle TiG', \nu' \rangle$, then there is a unique (up to isomorphism) match $x : L_{er} \to TiG'$ and transitions $\langle TiG', \nu' \rangle \overset{d}{\Rightarrow} \langle TiG', \nu'' \rangle \overset{er,x}{\Longrightarrow} \langle TiH, \nu \rangle$.

Proof. Since $L_{er} = R_{sr}$ holds and $\langle TiG, \nu \rangle \overset{sr,m}{\Longrightarrow} \langle TiG', \nu' \rangle$ does not delete any elements, we can define $x : L_{er} \to TiG$ such that $x \circ i_{L,er} = r^*_m \circ m \circ i_{L,sr}$. Since the application indicator edges created by $\langle TiG, \nu \rangle \overset{sr,m}{\Longrightarrow} \langle TiG', \nu' \rangle$ are distinguishable, x is unique (up to isomorphism).

According to the operational semantics given in Definition 5.2.21, the application of $\langle TiG, \nu \rangle \overset{sr,m_{sr}}{\Longrightarrow} \langle TiG', \nu' \rangle$ creates a clock instance ci and $\nu'(ci) = 0$ holds. The induced invariant rule $ir = (L_{ir}, z_{ir})$ now has a match $g : L_{ir} \to TiG'$ with $g(ci_{ir}) = ci$ to the current host graph TiG' where co_{ir} denotes the clock instance specified in L_{ir}. Since this is the only invariant rule with a match to the host graph, $I(G) = z_{ir}[ci/ci_{ir}] = \{ci \leq d\}$. Since $(\nu' + d) \models I(G)$ holds, the delay transition $\langle TiG', \nu' \rangle \overset{d}{\Rightarrow} \langle TiG', \nu'' \rangle$ is applicable. By applying the transition, we obtain ν'' with $\nu''(ci) = d$ and thus fulfill the time guard $z_{er}[ci/ci_{er}] = \{ci \geq d\}$ of $\langle TiG', \nu'' \rangle \overset{er,x}{\Longrightarrow} \langle TiH, \nu \rangle$.

Now, we can construct TiH according to the operational semantics. The clock instance ci is removed by an application of the clock instance removal rule. Thus, the new clock instance value assignment is $\nu = \emptyset$. □

After applying the induced end rule of $\mathcal{D}$ and obtaining configuration $\langle TiH, \nu \rangle$, the application of $\mathcal{D}$ can be seen as finished. The resulting graph TiH corresponds to a graph that would have been created by the application of $\mathcal{D}$ in an untimed graph transformation framework. More precisely, the execution of the durative graph transformation $\langle TiG, \nu \rangle \overset{\mathcal{D},m}{\Longrightarrow} \langle TiH, \nu \rangle$ results in a graph that is structurally identical to the graph we receive by executing an untimed graph transformation that is defined by an equivalent rule morphism and match. This property has already been shown in [ZH13a]. It assumes that no other durative graph transformation is executed concurrently during the execution of $\langle TiG, \nu \rangle \overset{\mathcal{D},m}{\Longrightarrow} \langle TiH, \nu \rangle$.

Proposition 5.3.2 (Correspondence of a durative transformation)**.** *Let $\mathcal{D}$ be a durative rule with $\mathcal{D} = (L_\mathcal{D}, R_\mathcal{D}, r_\mathcal{D}, name, d)$, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, \emptyset, E_G, \emptyset)$, and $\nu = \emptyset$ a clock instance value assignment. The induced start and end rule of $\mathcal{D}$ are denoted by sr and er, respectively. Further, let the graph transformation*

$p = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}})$ and the graph $G = (V_G, E_G)$ be a projection of $\mathcal{D}$ and TiG to the untimed case.

If and only if there is a match $g : L_{\mathcal{D}} \to G$ and a (direct) graph transformation $G \overset{\mathcal{D},g}{\Longrightarrow} H$, then there are matches $m : L_{sr} \to TiG$ and $x : L_{er} \to TiG'$ and transitions $\langle TiG, \nu \rangle \overset{sr,m}{\Longrightarrow} \langle TiG', \nu' \rangle \overset{d}{\Rightarrow} \langle TiG', \nu'' \rangle \overset{er,x}{\Longrightarrow} \langle TiH, \nu \rangle$ such that

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

Proof. Let $i_G : G \to TiG$ denote the isomorphism between G and $TiG|_{\{V_G, E_G\}}$.

First, we show that, given the match $g : L_{\mathcal{D}} \to G$, we can define the matches $m : L_{sr} \to TiG$ and $x : L_{er} \to TiG'$ such that $H = TiH|_{\{V_H, E_H\}}$. Since $L_{sr} = L_{\mathcal{D}}$ and $TiG|_{\{V_G, E_G\}} = G$ hold, we can define $m = i_G \circ g \circ i_{L,sr}^{-1}$. Then, we can construct TiG' and ν' according to the operational semantics given in Definition 5.2.21. TiG' now contains one clock instance ci and $\nu'(ci) = 0$. According to Lemma 5.3.1, there is a unique (up to isomorphism) match $x : L_{er} \to TiG'$ with $x \circ i_{L,er} = r_m^* \circ m \circ i_{L,sr}$ and transitions $\langle TiG', \nu' \rangle \overset{d}{\Rightarrow} \langle TiG', \nu'' \rangle \overset{er,x}{\Longrightarrow} \langle TiH, \nu \rangle$. Since $m = i_G \circ g \circ i_{L,sr}^{-1}$ and $x \circ i_{L,er} = r_m^* \circ m \circ i_{L,sr}$ hold, we have $x = r_m^* \circ i_G \circ g \circ i_{L,er}^{-1}$. Since r_m^* is total, $H = TiH|_{\{V_H, E_H\}}$ holds.

Second, we show that, given the matches $m : L_{sr} \to TiG$ and $x : L_{er} \to TiG'$, we can define the match $g : L_{\mathcal{D}} \to G$ such that $H = TiH|_{\{V_H, E_H\}}$. Since $L_{\mathcal{D}} \subseteq L_{er}$ holds, we can define $g = i_G^{-1} \circ x \circ i_{L,er}$. Then, we can construct H according to the SPO approach. Since $r_{\mathcal{D}} = i_{R,sr}^{-1} \circ r_{er}|_{\{V_{G,L}, E_{G,L}\}} \circ i_{L,er}$ and $g = i_G^{-1} \circ x \circ i_{L,er}$ hold, we have $H = TiH|_{\{V_H, E_H\}}$. □

Intuitively, the resulting graphs H and $TiH|_{\{V_H, E_H\}}$ are identical due to three facts. First, executing the start transformation leaves the essential parts of the graph unchanged. Second, all locking edges and special nodes, i.e., application indicators and clock instances, that are created by executing the start transformation are deleted again by executing the end transformation. Third, the end transformation realizes a graph transformation that conforms to the untimed graph transformation – to be precise, their RHSs are the same.

5.3.2 Rule Termination and Interleaving Transition Sequences

The application interval of a durative graph transformation is defined by the delay transition that is executed between the application of its induced start and end rule. At an arbitrary point in time during its application interval, an induced start or end rule of another durative rule can be applied. This is intended; otherwise, no concurrent execution would be possible.

The question is whether the end rule of an ongoing durative transformation can still be applied if one or more start or end rules induced by other durative rules have been applied during its application interval. The DGTS semantics is designed such that this works, i.e., durative transformations are guaranteed to finish once they have been started. This is formalized in Theorem 5.3.6.

The concurrent application of two durative graph transformation rules means that their induced start and end rules are applied in an interleaving manner. Multiple such interleavings are possible and each such interleaving results in the same configuration. This is formalized in Theorem 5.3.7.

Both of these properties build upon some lemmas, which are formalized first. Each of these lemmas gives the sequential or parallel independence between two applications of induced rules. We can then apply the Local Church-Rosser Theorem, see Theorem 2.5.1, which states that two sequential or parallel independent (direct) graph transformations can be applied in any order and both orderings result in the same graph. This is useful when proving Theorems 5.3.6 and 5.3.7.

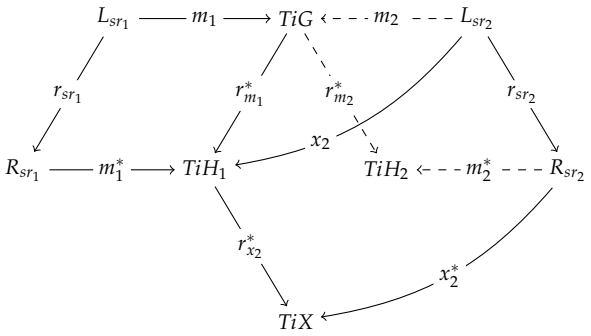
The first lemma considers two induced start rules that are applied in sequence.

Lemma 5.3.3 (Sequential independence between two start transformations)**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two durative rules, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, V_{CI}, E_G, E_{CI})$, and ν a clock instance value assignment such that $\nu \models TiG$. The induced start rules of $\mathcal{D}_1$ and $\mathcal{D}_2$ are denoted by sr_1 and sr_2, respectively. Further, ci_1 denotes the clock instance existing during the application interval of $\mathcal{D}_1$ and d_1 its duration.*

If there are two action transitions $\langle TiG, \nu\rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu'\rangle$ with $\nu'(ci_1) = 0$ and $\langle TiH_1, \nu''\rangle \overset{sr_2,x_2}{\Longrightarrow} \langle TiX, \nu'''\rangle$ with $0 \leq \nu''(ci_1) \leq d_1$, then they are sequentially independent.

Proof. We have to show that (i) $\langle TiH_1, \nu''\rangle \overset{sr_2,x_2}{\Longrightarrow} \langle TiX, \nu'''\rangle$ is weakly sequentially independent of $\langle TiG, \nu\rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu'\rangle$ and (ii) $\langle TiG, \nu\rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu'\rangle$ is weakly parallel independent of $\langle TiG, \nu\rangle \overset{sr_2,m_2}{\Longrightarrow} \langle TiH_2, \hat{\nu}\rangle$.

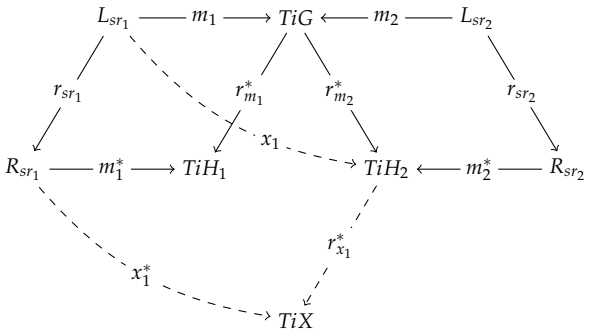
(i)



Since there are no application indicators or locking edges in the LHS of sr_2 (and thus there are none in the range of x_2) and $\langle TiG, \nu\rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu'\rangle$ creates only such elements, $\text{ran}(x_2) \cap TiH_1 \setminus \text{ran}(r^*_{m_1}) = \emptyset$ holds. Thus, we can construct $m_2 : L_{sr_2} \to TiG$ such that $r^*_{m_1} \circ m_2 = x_2$.

NACs Furthermore, m_2 fulfills each NAC in $\mathcal{N}_{sr_2}$ because $\langle TiG, \nu\rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu'\rangle$ does not delete any elements of TiG when deriving TiH_1 and x_2 already fulfills each NAC in $\mathcal{N}_{sr_2}$ by definition.

(ii)



Since $\langle TiG, \nu \rangle \overset{sr_2, m_2}{\Longrightarrow} \langle TiH_2, \hat{\nu} \rangle$ does not delete any elements of TiG when deriving TiH_2, $\text{ran}(m_2) \cap TiG \setminus \text{dom}(r^*_{m_1}) = \emptyset$ holds. Thus, we can construct $x_1 : L_{sr_1} \to TiH_2$ such that $x_1 = r^*_{m_2} \circ m_1$.

NACs We show that x_1 fulfills each NAC in $\mathcal{N}_{sr_1}$ by contradiction. Let us assume that $\langle TiG, \nu \rangle \overset{sr_2, m_2}{\Longrightarrow} \langle TiH_2, \hat{\nu} \rangle$ creates locking edges that conflict with a NAC in $\mathcal{N}_{sr_1}$, i.e., there is a match $q_1 : N_{sr_1} \to TiH_2$ with $q_1 \circ n_{sr_1} = x_1$ and $(N_{sr_1}, n_{sr_1}) \in \mathcal{N}_{sr_1}$ such that $\text{ran}(q_1) \cap TiH_2 \setminus \text{ran}(r^*_{m_2}) \neq \emptyset$.

According to Definition 5.2.10, each NAC that realizes a check for a read lock [write lock] is accompanied by attaching a write lock [read lock] to the same element and vice versa. Thus, $\langle TiG, \nu \rangle \overset{sr_1, m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ creates locking edges that conflict with a NAC of sr_2, i.e., there is a match $q_2 : N_{sr_2} \to TiH_1$ with $q_2 \circ n_{sr_2} = x_2$ and $(N_{sr_2}, n_{sr_2}) \in \mathcal{N}_{sr_2}$ such that $\text{ran}(q_2) \cap TiH_1 \setminus \text{ran}(r^*_{m_1}) \neq \emptyset$. This is a contradiction to the applicability of $\langle TiH_1, \nu'' \rangle \overset{sr_2, x_2}{\Longrightarrow} \langle TiX, \nu''' \rangle$. □

Intuitively, Lemma 5.3.3 holds due to two facts:

1. The latter start transformation only adds elements to the host graph but deletes none, thus cannot conflict with the applicability of the earlier start transformation. In other words, there cannot be a use-delete conflict.

2. Since no NAC of the latter start transformation matches, i.e., the two transformations are free of produce-forbid conflicts, and the locking mechanism is designed symmetrically, they also have to be free of forbid-produce conflicts.

The next lemma considers an end transformation being applied after a start transformation. Instead of "ordinary" sequential independence, it states sequential independence *modulo isomorphism*. Ordinary sequential independence is not sufficient due to the shared read locks. A locking edge created by the start transformation can be deleted by the end transformation if both transformations read the same element. However, in such a case, there exists another locking edge, which is isomorphic to the first one.

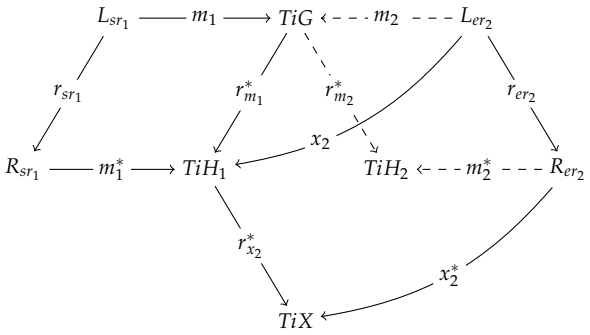
Lemma 5.3.4 (Sequential independence modulo isomorphism between a start and an end transformation)**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two durative rules, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, V_{CI}, E_G, E_{CI})$, and ν a clock instance value assignment such that $\nu \models TiG$ and $\langle TiG, \nu \rangle$ is reachable from the initial configuration. The induced start rule*

of $\mathcal{D}_1$ and end rule of $\mathcal{D}_2$ are denoted by sr_1 and er_2, respectively. Further, ci_1 denotes the clock instance existing during the application interval of $\mathcal{D}_1$ and d_1 its duration. Also, ai_{sr_1} and ai_{er_2} denote the application indicator in the RHS of sr_1 and the LHS of er_2, respectively.

If there are two action transitions $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ with $\nu'(ci_1) = 0$ and $\langle TiH_1, \nu'' \rangle \overset{er_2,x_2}{\Longrightarrow} \langle TiX, \nu''' \rangle$ with $0 \leq \nu''(ci_1) \leq d_1$ and $x_2(ai_{er_2}) \neq m_1^*(ai_{sr_1})$, then they are sequentially independent modulo isomorphism.

Proof. We have to show that (i) $\langle TiH_1, \nu'' \rangle \overset{er_2,x_2}{\Longrightarrow} \langle TiX, \nu''' \rangle$ is weakly sequentially independent modulo isomorphism of $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ and (ii) $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ is weakly parallel independent modulo isomorphism of $\langle TiG, \nu \rangle \overset{er_2,m_2}{\Longrightarrow} \langle TiH_2, \hat{\nu} \rangle$.

(i)



If $\langle TiH_1, \nu'' \rangle \overset{er_2,x_2}{\Longrightarrow} \langle TiX, \nu''' \rangle$ does not delete any locking edges that have been created by $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$, we have $\text{ran}(x_2) \cap TiH_1 \setminus \text{ran}(r^*_{m_1}) = \emptyset$ and can thus construct m_2 such that $r^*_{m_1} \circ m_2 = x_2$. For the case it does, we have to show that m_2 can be constructed such that $r^*_{m_1} \circ m_2 = \widetilde{x}_2$ where $\widetilde{x}_2$ is isomorphic to x_2.

Since x_2 is total, there exists an application indicator $x_2(ai_{er_2})$ in TiG. This application indicator must have been created by the induced start rule sr_2 of $\mathcal{D}_2$. Thus, there has to be an application of sr_2 in the transition sequence from the initial configuration to TiG. Let $y_2 : L_{sr_2} \to TiF$ denote its match and seq the transition sequence from TiF to TiG.

Now we have two cases: either none of the locking edges created by the start rule transformation $\overset{sr_2,y_2}{\Longrightarrow}$ has been deleted by any of the transformations in seq or at least one of the locking edges has been deleted.

a) None of the locking edges has been deleted by any of the transformations in seq. In this case, we can construct m_2 such that $r^*_{m_1} \circ m_2 = \widetilde{x}_2$ where $\widetilde{x}_2$ is isomorphic to x_2 because TiG contains locking edges which are isomorphic to the ones created by $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$.

b) At least one of the locking edges has been deleted by at least one of the transformations in seq. To construct m_2 such that $r^*_{m_1} \circ m_2 = \widetilde{x}_2$ where $\widetilde{x}_2$ is isomorphic to x_2, locking edges have to exist that are isomorphic to the locking edges that have been created by $\overset{sr_2,y_2}{\Longrightarrow}$ but deleted by a

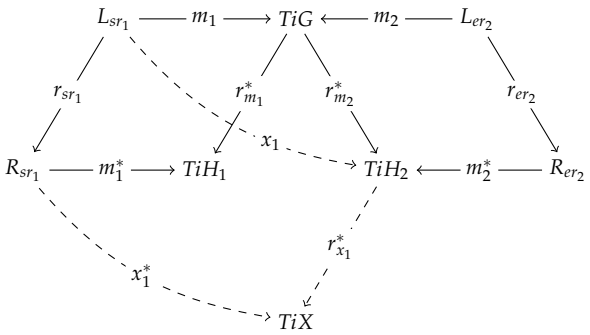
transformation in seq. Let $\bar{er}_i$ denote the rules of transformations deleting the locking edges and $\bar{m}_i : L_{\bar{er}_i} \to TiE_i$ their matches with $i = 1, \ldots, n$ where n is the number of transformations deleting the locking edges.

Each transformation $\xRightarrow{\bar{er}_i, \bar{m}_i}$ must have been the application of an end rule because start rules do not delete anything. Thus, for each $\xRightarrow{\bar{er}_i, \bar{m}_i}$, there must have been a start rule transformation $\xRightarrow{\bar{sr}_i, \bar{y}_i}$ with $\bar{y}_i : L_{\bar{sr}_i} \to TiD_i$ creating the application indicator that $\xRightarrow{\bar{er}_i, \bar{m}_i}$ deletes. According to Definitions 5.2.10 and 5.2.11, each $\xRightarrow{\bar{sr}_i, \bar{y}_i}$ also creates locking edges which are isomorphic to the ones that $\xRightarrow{\bar{er}_i, \bar{m}_i}$ deletes.

Again we have two cases: either they still exist in TiG or they have been deleted. If they still exist in TiG, we can construct m_2 as in (a). If not, they have been deleted by transformations in the transition sequences from TiD_i to TiG. In such a case, we can repeat the argument of (b). This argument loop terminates because the sequence of transitions from the initial configuration to TiG is finite. Thus, locking edges exist in TiG that are isomorphic to the ones created by $\langle TiG, \nu \rangle \xRightarrow{sr_1, m_1} \langle TiH_1, \nu' \rangle$ and m_2 can be constructed such that $r^*_{m_1} \circ m_2 = \widetilde{x}_2$ where $\widetilde{x}_2$ is isomorphic to x_2.

NACs Furthermore, $m_2 \models \mathcal{N}_{er_2}$ holds obviously because er_2 does not contain any NACs.

(ii)



We show that $\langle TiG, \nu \rangle \xRightarrow{sr_1, m_1} \langle TiH_1, \nu' \rangle$ is weakly parallel independent of $\langle TiG, \nu \rangle \xRightarrow{er_2, m_2} \langle TiH_2, \hat{\nu} \rangle$. This implies its weak parallel independence modulo isomorphism.

We show that the match x_1 such that $x_1 = r^*_{m_2} \circ m_1$ can be constructed by contradiction. Let us assume that $\langle TiG, \nu \rangle \xRightarrow{er_2, m_2} \langle TiH_2, \hat{\nu} \rangle$ deletes elements of TiG which are required for x_1 to be total, i.e., $\mathrm{ran}(m_1) \cap TiG \setminus \mathrm{dom}(r^*_{m_2}) \neq \emptyset$.

According to Definition 5.2.11, the deletion of elements is accompanied by releasing a write lock, i.e., deleting a locking edge that constitutes a write operation. Thus, each of the elements in $\mathrm{ran}(m_1) \cap TiG \setminus \mathrm{dom}(r^*_{m_2})$ has a write lock attached.

According to Definition 5.2.10, each element in the range of the LHS's match is accompanied by a NAC that checks for write locks. Since the elements

in $\text{ran}(m_1) \cap TiG \setminus \text{dom}(r^*_{m_2})$ are obviously contained in the range of m_1 and these elements have write locks attached, m_1 does not fulfill each NAC in $\mathcal{N}_{sr_1}$. This is a contradiction to the applicability of $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$.

NACs Furthermore, x_1 fulfills each NAC in $\mathcal{N}_{sr_1}$ because $\langle TiG, \nu \rangle \overset{er_2,m_2}{\Longrightarrow} \langle TiH_2, \hat{\nu} \rangle$ does not create any locking edges when deriving TiH_2, $\mathcal{N}_{sr_1}$ contains only NACs that constitute checks for locking edges, and m_1 already fulfills each NAC in $\mathcal{N}_{sr_1}$ by definition. □

Intuitively, Lemma 5.3.4 holds due to two facts:

1. The end transformation consuming locking edges implies the existence of another start transformation that created such locking edges earlier, thus providing exactly the same (up to isomorphism) locking edges as if none of the two transformations were applied.

2. Elements supposed to be deleted by a future end transformation, i.e., the counterpart of the start transformation, cannot be deleted by any other transformation because the start transformation attached locking edges to them.

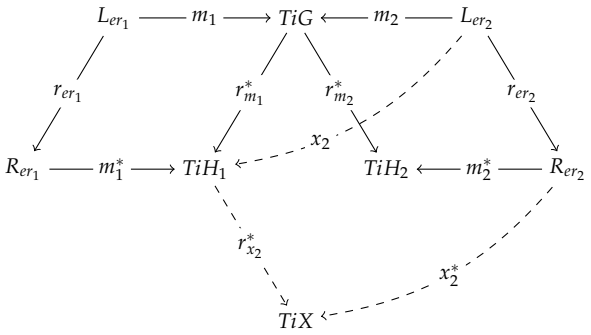
The next lemma considers two end transformations being applicable in the same configuration. For the first two lemmas, we assumed a situation where the transformations are applied in sequence. This ensured their sequential independence. If they were not sequentially independent, the second transformation would not have been applicable at all. When considering two end transformations, this is not necessary. Here, their applicability alone already ensures that they are parallel independent.

Lemma 5.3.5 (Parallel independence modulo isomorphism between two end transformations)**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two durative rules, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, V_{CI}, E_G, E_{CI})$, and ν a clock instance value assignment such that $\nu \models TiG$ and $\langle TiG, \nu \rangle$ is reachable from the initial configuration. The induced end rules of $\mathcal{D}_1$ and $\mathcal{D}_2$ are denoted by er_1 and er_2, respectively. Further, ci_1 and ci_2 denote the clock instance existing during the application interval of $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Also, ai_{er_1} and ai_{er_2} denote the application indicator in the LHS of er_1 and er_2, respectively.*

If there are two action transitions $\langle TiG, \nu \rangle \overset{er_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ with $\nu'(ci_1) = 0$ and $\langle TiG, \nu \rangle \overset{er_2,m_2}{\Longrightarrow} \langle TiH_2, \nu'' \rangle$ with $\nu''(ci_2) = 0$ and $m_2(ai_{er_2}) \neq m_1(ai_{er_1})$, then they are parallel independent modulo isomorphism.

Proof. We have to show that (i) $\langle TiG, \nu \rangle \overset{er_2,m_2}{\Longrightarrow} \langle TiH_2, \nu'' \rangle$ is weakly parallel independent modulo isomorphism of $\langle TiG, \nu \rangle \overset{er_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ and (ii) $\langle TiG, \nu \rangle \overset{er_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ is weakly parallel independent modulo isomorphism of $\langle TiG, \nu \rangle \overset{er_2,m_2}{\Longrightarrow} \langle TiH_2, \nu'' \rangle$.

(i)



If $r^*_{m_1} \circ m_2$ is total, i.e., $\text{ran}(m_2) \cap TiG \setminus \text{dom}(r^*_{m_1}) = \emptyset$, we can simply construct $x_2 : L_{er_2} \to TiH_1$ such that $x_2 = r^*_{m_1} \circ m_2$. Otherwise, we have to show that there exist a total morphism $\widetilde{m}_2 : L_{er_2} \to TiG$ such that $\widetilde{m}_2$ is isomorphic to m_2 and then construct x_2 such that $x_2 = r^*_{m_1} \circ \widetilde{m}_2$ or there is a contradiction.

There are two cases: either $\text{ran}(m_2) \cap TiG \setminus \text{dom}(r^*_{m_1})$ contains only locking elements or it also contains other elements than locks.

a) $\text{ran}(m_2) \cap TiG \setminus \text{dom}(r^*_{m_1})$ contains only locking elements. In this case, there exist a $\widetilde{m}_2 : L_{er_2} \to TiG$ which is isomorphic to m_2. Thus, we can construct $x_2 : L_{er_2} \to TiH_1$ such that $x_2 = r^*_{m_1} \circ \widetilde{m}_2$. The proof that $\widetilde{m}_2$ exists is analogous to the proof of Lemma 5.3.4 (i).

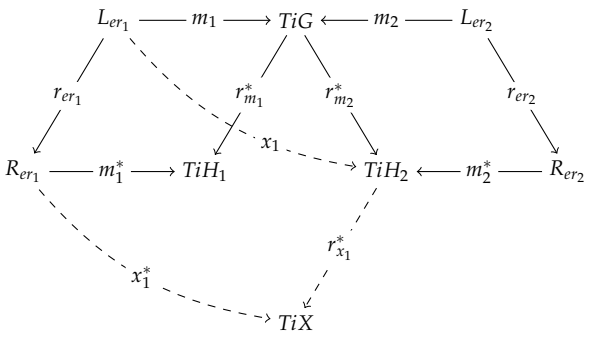
b) $\text{ran}(m_2) \cap TiG \setminus \text{dom}(r^*_{m_1})$ contains elements other than locks. According to Definition 5.2.11, the deletion of elements is accompanied by releasing a write lock, i.e., deleting a locking edge that constitutes a write operation. Thus, each of the elements in $TiG \setminus \text{dom}(r^*_{m_1})$ has a write lock attached.

   These write locks (or isomorphic ones) must have been created by an application of a start rule sr_1 of $\mathcal{D}_1$ that also created the application indicator that $\stackrel{er_1,m_1}{\Longrightarrow}$ deletes. Similarly, each of the elements in $\text{ran}(m_2)$ has a read lock attached, which (modulo isomorphism) must have been created by an application of a start rule sr_2 of $\mathcal{D}_2$. Let $y_1 : L_{sr_1} \to TiF_1$ and $y_2 : L_{sr_2} \to TiF_2$ denote the the match of sr_1 and sr_2, respectively. Since $m_2(ai_{er_2}) \neq m_1(ai_{er_1})$ holds, we have $sr_1 \neq sr_2 \vee y_1 \neq y_2$.

   Now, there are two possible orderings: either $\stackrel{sr_1,y_1}{\Longrightarrow}$ happens before or after $\stackrel{sr_2,y_2}{\Longrightarrow}$ in the transition sequence from the initial configuration to TiG. In case of the former, $\stackrel{sr_1,y_1}{\Longrightarrow}$ creates a write lock that still exists in TiF_2. In case of the latter, $\stackrel{sr_2,y_2}{\Longrightarrow}$ creates a read lock that still exists in TiF_1. The existence of these locking elements (or isomorphic ones) can be shown analogously to the argument loop in the proof of Lemma 5.3.4 (i). According to Definition 5.2.10, each creation of a write lock [read lock] is accompanied by a NAC that realizes a check for a read lock [write lock]. Thus, the write lock [read lock] existing in TiF_2 [TiF_1] conflicts with a NAC of sr_2 [sr_1]. Both constitute a contradiction to the applicability of the second transformation.

NACs Furthermore, $x_2 \models \mathcal{N}_{er_2}$ holds obviously because er_2 does not contain any NACs.

(ii)



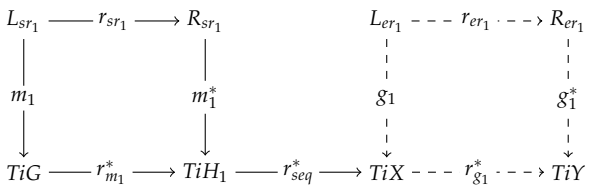
This proof is analogous to the proof of (i). □

Intuitively, the parallel independence of the end transformations results from the independence of their start transformation counterparts. If the start transformations were not independent, they could not have been applied during the transition sequence from the initial configuration to the current configuration.

Now, we formalize the property that ensures that each durative graph transformation terminates properly, i.e., no other transformation can cause the induced end rule of the ongoing durative graph transformation not to be applicable anymore. As a consequence, durative graph transformations can only be executed if they do not interfere with ongoing durative graph transformations.

Theorem 5.3.6 (Termination of a durative rule)**.** *Let $\mathcal{D}_1$ be a durative rule, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, V_{CI}, E_G, E_{CI})$, and ν a clock instance value assignment such that $\nu \models TiG$ and $\langle TiG, \nu \rangle$ is reachable from the initial configuration. The induced start and end rule of $\mathcal{D}_1$ are denoted by sr_1 and er_1, respectively. Further, i_{L,sr_1} and i_{L,er_1} denote the morphisms identifying the elements of $L_{\mathcal{D}_1}$ in L_{sr_1} and L_{er_1}, respectively. Also, ai_{sr_1} and ai_{er_1} denote the application indicator in the RHS of sr_1 and the LHS of er_2, respectively.*

If there exists a transition sequence $\langle TiG, \nu \rangle \overset{sr_1, m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle \overset{seq}{\rightsquigarrow} \langle TiX, \nu'' \rangle$ with $\langle TiA, \hat{\nu} \rangle \overset{er_1, a_1}{\Longrightarrow} \langle TiB, \hat{\nu}' \rangle \notin seq$ for any match $a_1 : L_{er_1} \to TiA$ such that $a_1(ai_{er_1}) = r^*_{pre} \circ m^*_1(ai_{sr_1})$ where r^*_{pre} denotes the derivation morphism of a prefix transition sequence of seq ending in TiA, then there exists a unique (up to isomorphism) match $g_1 : L_{er_1} \to TiX$ such that $g_1 \circ i_{L,er_1} = r^*_{seq} \circ r^*_{m_1} \circ m_1 \circ i_{L,sr_1}$ and an action transition $\langle TiX, \nu'' \rangle \overset{er_1, g_1}{\Longrightarrow} \langle TiY, \nu''' \rangle$.



Proof. We show this property by induction over the number of transitions in *seq*.

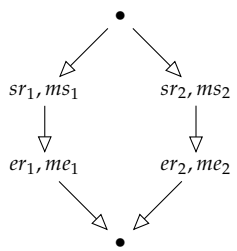
Basis step. The transition sequence *seq* is empty. Thus, $\langle TiH_1, \nu' \rangle = \langle TiX, \nu'' \rangle$ holds. Now, we only have to show that there exists a match $g_1 : L_{er_1} \to TiH_1$ such that $g_1 \circ i_{L,er_1} = r^*_{m_1} \circ m_1 \circ i_{L,sr_1}$. According to Definitions 5.2.10 and 5.2.11, $L_{er_1} = R_{sr_1}$ holds. Thus, we can simply define g_1 such that $g_1 \circ i_{L,er_1} = m^*_1 \circ r_{sr_1} \circ i_{L,sr_1} = r^*_{m_1} \circ m_1 \circ i_{L,sr_1}$.

Induction step. Let $\overset{tr_2,x_2}{\Longrightarrow}$ denote the first transition in *seq*. That way, we have $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle \overset{tr_2,x_2}{\Longrightarrow} \langle TiI, \xi \rangle \overset{seq'}{\rightsquigarrow} \langle TiX, \nu'' \rangle$. According to Lemmas 5.3.3 and 5.3.4 the transformations $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle \overset{tr_2,x_2}{\Longrightarrow} \langle TiI, \xi \rangle$ are sequentially independent (modulo isomorphism). Theorem 2.5.1 states that sequentially independent transformations can be reordered and still result in the same graph. Thus, we get $\langle TiG, \nu \rangle \overset{tr_2,m_2}{\Longrightarrow} \langle TiH_2, \xi' \rangle \overset{sr_1,x_1}{\Longrightarrow} \langle TiI, \xi \rangle \overset{seq'}{\rightsquigarrow} \langle TiX, \nu'' \rangle$. Now, we can apply the induction hypothesis, which results in the existence of $g_1 : L_{er_1} \to TiX$ such that $g_1 \circ i_{L,er_1} = r^*_{seq'} \circ r^*_{x_1} \circ x_1 \circ i_{L,sr_1}$. Using m_1 instead of x_1, we get $g_1 \circ i_{L,er_1} = r^*_{seq'} \circ r^*_{x_2} \circ r^*_{m_1} \circ m_1 \circ i_{L,sr_1}$. Since $r^*_{seq} = r^*_{seq'} \circ r^*_{x_2}$ holds, we get $g_1 \circ i_{L,er_1} = r^*_{seq} \circ r^*_{m_1} \circ m_1 \circ i_{L,sr_1}$. □

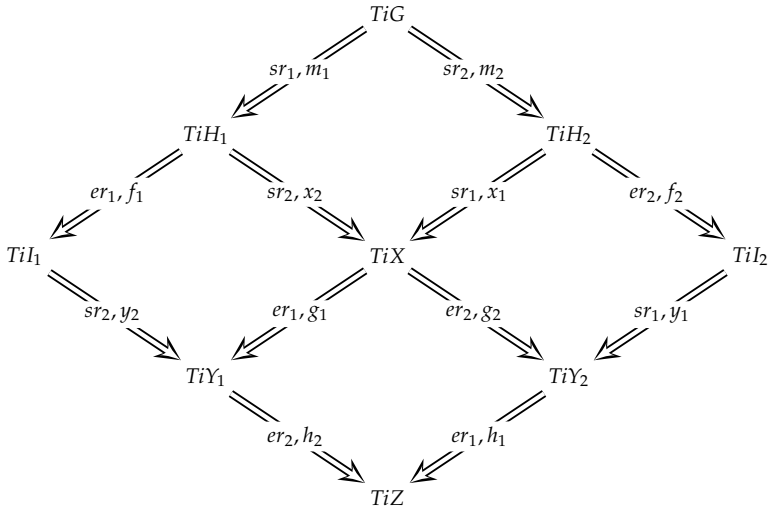
While Theorem 5.3.6 ensures that each durative graph transformation terminates properly, even when other transformations are applied during its application interval, it does not state anything about the configuration that results in such cases. The next property does. It states that each interleaving of two durative graph transformations results in the same configuration when both transformations finished (and no other transformation is involved). From a more abstract perspective, this property characterizes all possible interleavings of two durative graph transformations that are independent of each other.

Theorem 5.3.7 (Existence of interleaving transition sequences)**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two durative rules, $TiG = (G_{TiG}, type_{TiG})$ a timed graph with $G_{TiG} = (V_G, V_{CI}, E_G, E_{CI})$, and ν a clock instance value assignment such that $\nu \models TiG$ and $\langle TiG, \nu \rangle$ is reachable from the initial configuration. The induced start and end rules of $\mathcal{D}_1$ and $\mathcal{D}_2$ are denoted by sr_1, er_1, sr_2, and er_2, respectively.*

If $\langle TiG, \nu \rangle \overset{sr_1,m_1}{\Longrightarrow} \langle TiH_1, \nu' \rangle$ and $\langle TiG, \nu \rangle \overset{sr_2,m_2}{\Longrightarrow} \langle TiH_2, \nu'' \rangle$ are parallel independent, there exist matches ms_1, me_1, ms_2, and me_2 for sr_1, er_1, sr_2, and er_2, respectively, such that each of the transition sequences fulfilling the partial order



exists and results in the same graph TiZ.



Proof. According to the Local Church-Rosser Theorem, both sequentializations of two parallel independent transformations result in the same graph. Thus, we have the transition sequence shown in Figure 5.11(a).

According to Definitions 5.2.10 and 5.2.11, $L_{er_1} = R_{sr_1}$ holds. Thus, the match $g_1 : L_{er_1} \to TiX$ can be defined such that $g_1 \circ i_{L,er_1} = x_1^* \circ r_{sr_1} \circ i_{L,sr_1} = r_{x_1}^* \circ x_1 \circ i_{L,sr_1}$. The match $g_2 : L_{er_2} \to TiX$ can be defined analogously. Now, we have the transition sequence shown in Figure 5.11(b).

According to Lemma 5.3.4, the transformations $\overset{sr_2,x_2}{\Longrightarrow}$ and $\overset{er_1,g_1}{\Longrightarrow}$ are sequentially independent. Since the Local Church-Rosser Theorem also works for sequentially independent transformations, we get the transition sequence shown in Figure 5.11(c).

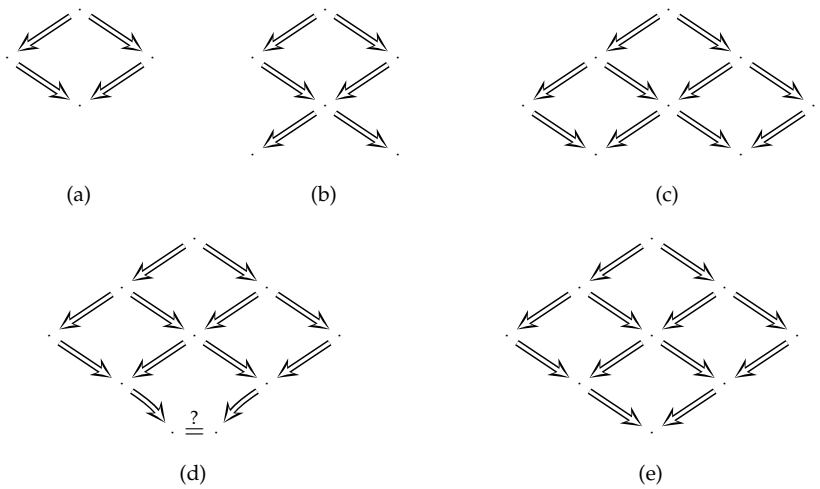


Figure 5.11: Visual aid for the proof of Theorem 5.3.7

Now, we can define $h_1 : L_{er_1} \to TiY_2$ such that $h_1 \circ i_{L,er_1} = y_1^* \circ r_{sr_1} \circ i_{L,sr_1} = r_{y_1}^* \circ y_1 \circ i_{L,sr_1}$. Again, we can define $h_2 : L_{er_2} \to TiY_1$ analogously. Thus, we get the transition sequence shown in Figure 5.11(d).

According to Lemma 5.3.5, the transformations $\overset{er_1,h_1}{\Longrightarrow}$ and $\overset{er_2,h_2}{\Longrightarrow}$ are parallel independent. By applying the Local Church-Rosser Theorem again, we get the transition sequence shown in Figure 5.11(e). □

5.4 Support for Negative Application Conditions

In Section 5.2, durative graph transformation rules have been defined without the support for negative application conditions. Since produce-forbid and forbid-produce conflicts cannot occur when there are no NACs, the locking mechanism was implemented considering only delete-use and use-delete conflicts. Now, we extend the syntax and semantics to support certain kinds of NACs, i.e., forbidden edges and forbidden pairs, on the level of durative rules. Note that on the level of timed graph transformation rules, NACs have already been used.

Definition 5.4.1 (Durative graph transformation rule with NACs)**.** A *durative graph transformation rule with NACs* $\mathcal{D}' = (L, R, r, \mathcal{N}, name, d)$ differs from a durative graph transformations rule $\mathcal{D} = (L, R, r, name, d)$ in that it contains a set of NACs $\mathcal{N}$ where each NAC is a tuple $(N, n) \in \mathcal{N}$ with $n : V_{G,L} \cup E_{G,L} \to N$, n being injective, and either

- $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{fe\}$, (forbidden edge)
- $V_N = V_{G,L} \cup \{fn\} \wedge E_N = E_{G,L} \cup \{fe\} \wedge src(fe) \in \text{ran}(n) \wedge$ $tgt(fe) = fn$, or (forbidden pair with outgoing edge)
- $V_N = V_{G,L} \cup \{fn\} \wedge E_N = E_{G,L} \cup \{fe\} \wedge src(fe) = fn \wedge$ $tgt(fe) \in \text{ran}(n)$. (forbidden pair with incoming edge)

The extension of the syntax is straightforward. To ensure a well-formed specification, NACs are restricted to forbidden edges, forbidden pairs with outgoing edges, and forbidden pairs with incoming edges.

In the RailCab example, the durative rule `accelerateRailCab`, which is given in Figure 5.12, contains a forbidden pair. The rule specifies the movement of a RailCab from one track section to the next. The RailCab's acceleration is captured by the `driving` self edge. In the rule's LHS, there is no such edge, i.e., the RailCab is not yet in driving motion. To ensure that the RailCab is also not involved in a convoy operation (or engaged in establishing a convoy operation) when the rule is being applied, it specifies a forbidden pair with an incoming edge adjacent to the `RailCab` node.

The support for NACs brings about the existence of potential produce-forbid and forbid-produce conflicts. Using locking for the prevention of such conflicts is slightly more complicated than with delete-use and use-delete conflicts. The problem is that elements contained in a forbidden pair are not available in the host graph an

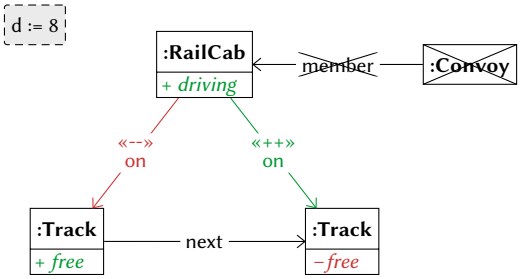


Figure 5.12: Durative rule `accelerateRailCab`

can thus not be locked. For this reason, locks are attached to the forbidden pairs' connecting nodes instead.

The locking functionality required for supporting forbidden edges can be implemented almost analogous to that of required edges. While we can can use the same locking edge types for forbidden edges as for required edges, this is not possible in the case of forbidden pairs. In order to support forbidden pairs, we extend the type graph by two locking edge types for each potential forbidden pair: one of them implements a read lock, the other one a write lock. By being distinctive for each node, edge, and reading direction possible in a forbidden pair (incoming or outgoing edge), each of these locking edge types unambiguously identifies the forbidden pair it relates to.

To refer to these locking edge types, we use the functions $rlpair_{src} : E_{TG} \to E_{TG}$, $rlpair_{tgt} : E_{TG} \to E_{TG}$, $wlpair_{src} : E_{TG} \to E_{TG}$, and $wlpair_{tgt} : E_{TG} \to E_{TG}$. Each source type of an edge type et has two locking edge types, $rlpair_{src}(et)$ and $wlpair_{src}(et)$ in the TGTS type graph. An edge of type $rlpair_{src}(et)$ denotes an obtained read lock for a forbidden pair consisting of an outgoing edge of type et and a node of its target's type. An edge of type $wlpair_{src}(et)$ denotes an obtained write lock representing the creation of graph elements matching this forbidden pair. Analogously, for a forbidden pair consisting of an incoming edge of edge type et and a node if its source type, $rlpair_{tgt}(et)$ and $wlpair_{tgt}(et)$ denote its locking edge types.

As with delete-use and use-delete conflicts, the locking mechanism for the prevention of produce-forbid and forbid-produce conflicts is realized via induced start and end rules. Therefore, we now extend these rules to support durative graph transformation systems where NACs do exist.

Definition 5.4.2 (Induced start rule supporting NACs)**.** Given a durative graph transformation rule with NACs $\mathcal{D}' = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}}, \mathcal{N}_{\mathcal{D}}, name, d)$, the *induced start rule (supporting NACs)* of $\mathcal{D}'$ is a timed rule $sr' = (L, R', r, \mathcal{N}', z, V_{res})$ that differs from the induced start rule $sr = (L, R, r, \mathcal{N}, z, V_{res})$ of a durative graph transformation rule $\mathcal{D} = (L_{\mathcal{D}}, R_{\mathcal{D}}, r_{\mathcal{D}}, name, d)$ in that

- $E_{G,R'} = E_{G,R} \cup E_{RL.fedge,R} \cup E_{WL.fedge,R} \cup E_{RL.fpair,R} \cup E_{WL.fpair,R}$,
- $\mathcal{N}' = \mathcal{N} \cup \mathcal{N}_{\mathcal{D}} \cup \mathcal{N}_{WL.fedge} \cup \mathcal{N}_{RL.fedge} \cup \mathcal{N}_{WL.fpair} \cup \mathcal{N}_{RL.fpair}$,

- $E_{RL.fedge,R} = \{le | \exists (N_{\mathcal{D}}, n_{\mathcal{D}}) \in \mathcal{N}_{\mathcal{D}} : \exists fe \in E_{N_{\mathcal{D}}} \setminus E_{L_{\mathcal{D}}} :$
  $src(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge tgt(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge$
  $src(le) = r \circ i_L \circ n_{\mathcal{D}}^{-1} \circ src(fe) \wedge$
  $tgt(le) = r \circ i_L \circ n_{\mathcal{D}}^{-1} \circ tgt(fe) \wedge$
  $type(le) = rledge \circ type(fe)\},$

- $E_{WL.fedge,R} = \{le | \exists e \in E_{R_{\mathcal{D}}} \setminus \text{ran}(r_{\mathcal{D}}) :$
  $src(e) \in \text{ran}(r_{\mathcal{D}}) \wedge tgt(e) \in \text{ran}(r_{\mathcal{D}}) \wedge$
  $src(le) = r \circ i_L \circ r_{\mathcal{D}}^{-1} \circ src(e) \wedge$
  $tgt(le) = r \circ i_L \circ r_{\mathcal{D}}^{-1} \circ tgt(e) \wedge$
  $type(le) = wledge \circ type(e)\},$

- $E_{RL.fpair,R} = \{le | \exists (N_{\mathcal{D}}, n_{\mathcal{D}}) \in \mathcal{N}_{\mathcal{D}} : \exists fe \in E_{N_{\mathcal{D}}} \setminus E_{L_{\mathcal{D}}} :$
  $\exists fn \in V_{N_{\mathcal{D}}} \setminus V_{L_{\mathcal{D}}} :$
  $src(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge tgt(fe) = fn \wedge$
  $src(le) = tgt(le) = r \circ i_L \circ n_{\mathcal{D}}^{-1} \circ src(fe) \wedge$
  $type(le) = rlpair_{src} \circ type(fe)\} \cup$
  $\{le | \exists (N_{\mathcal{D}}, n_{\mathcal{D}}) \in \mathcal{N}_{\mathcal{D}} : \exists fe \in E_{N_{\mathcal{D}}} \setminus E_{L_{\mathcal{D}}} :$
  $\exists fn \in V_{N_{\mathcal{D}}} \setminus V_{L_{\mathcal{D}}} :$
  $src(fe) = fn \wedge tgt(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge$
  $src(le) = tgt(le) = r \circ i_L \circ n_{\mathcal{D}}^{-1} \circ tgt(fe) \wedge$
  $type(le) = rlpair_{tgt} \circ type(fe)\},$

- $E_{WL.fpair,R} = \{le | \exists e \in E_{R_{\mathcal{D}}} \setminus \text{ran}(r_{\mathcal{D}}) :$
  $src(e) \in \text{ran}(r_{\mathcal{D}}) \wedge$
  $src(le) = tgt(le) = r \circ i_L \circ r_{\mathcal{D}}^{-1} \circ src(e) \wedge$
  $type(le) = wlpair_{src} \circ type(e)\} \cup$
  $\{le | \exists e \in E_{R_{\mathcal{D}}} \setminus \text{ran}(r_{\mathcal{D}}) :$
  $tgt(e) \in \text{ran}(r_{\mathcal{D}}) \wedge$
  $src(le) = tgt(le) = r \circ i_L \circ r_{\mathcal{D}}^{-1} \circ tgt(e) \wedge$
  $type(le) = wlpair_{tgt} \circ type(e)\},$

- $\mathcal{N}_{WL.fedge} = \{(N, n) | \exists (N_{\mathcal{D}}, n_{\mathcal{D}}) \in \mathcal{N}_{\mathcal{D}} : \exists fe \in E_{N_{\mathcal{D}}} \setminus E_{L_{\mathcal{D}}} :$
  $src(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge tgt(fe) \in \text{ran}(n_{\mathcal{D}}) \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = n \circ i_L \circ n_{\mathcal{D}}^{-1} \circ src(fe) \wedge$
  $tgt(ne) = n \circ i_L \circ n_{\mathcal{D}}^{-1} \circ tgt(fe) \wedge$
  $type(ne) = wledge \circ type(fe)\},$

- $\mathcal{N}_{RL.fedge} = \{(N, n) | \exists e \in E_{R_{\mathcal{D}}} \setminus \text{ran}(r_{\mathcal{D}}) :$
  $src(e) \in \text{ran}(r_{\mathcal{D}}) \wedge tgt(e) \in \text{ran}(r_{\mathcal{D}}) \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = n \circ i_L \circ r_{\mathcal{D}}^{-1} \circ src(e) \wedge$
  $tgt(ne) = n \circ i_L \circ r_{\mathcal{D}}^{-1} \circ tgt(e) \wedge$
  $type(ne) = wledge \circ type(e)\},$

- $\mathcal{N}_{WL.fpair} = \{(N,n) | \exists (N_\mathcal{D}, n_\mathcal{D}) \in \mathcal{N}_\mathcal{D} : \exists fe \in E_{N_\mathcal{D}} \setminus E_{L_\mathcal{D}} :$
  $\exists fn \in V_{N_\mathcal{D}} \setminus V_{L_\mathcal{D}} :$
  $src(fe) \in \text{ran}(n_\mathcal{D}) \wedge tgt(fe) = fn \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = tgt(ne) = n \circ i_L \circ n_\mathcal{D}^{-1} \circ src(fe) \wedge$
  $type(ne) = rlpair_{src} \circ type(fe)\} \cup$
  $\{(N,n) | \exists (N_\mathcal{D}, n_\mathcal{D}) \in \mathcal{N}_\mathcal{D} : \exists fe \in E_{N_\mathcal{D}} \setminus E_{L_\mathcal{D}} :$
  $\exists fn \in V_{N_\mathcal{D}} \setminus V_{L_\mathcal{D}} :$
  $src(fe) = fn \wedge tgt(fe) \in \text{ran}(n_\mathcal{D}) \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = tgt(ne) = n \circ i_L \circ n_\mathcal{D}^{-1} \circ tgt(fe) \wedge$
  $type(ne) = rlpair_{tgt} \circ type(fe)\},$
- $\mathcal{N}_{RL.fpair} = \{(N,n) | \exists e \in E_{R_\mathcal{D}} \setminus \text{ran}(r_\mathcal{D}) :$
  $src(e) \in \text{ran}(r_\mathcal{D}) \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = tgt(ne) = n \circ i_L \circ r_\mathcal{D}^{-1} \circ src(e) \wedge$
  $type(ne) = wlpair_{src} \circ type(e)\} \cup$
  $\{(N,n) | \exists e \in E_{R_\mathcal{D}} \setminus \text{ran}(r_\mathcal{D}) :$
  $tgt(e) \in \text{ran}(r_\mathcal{D}) \wedge$
  $V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = tgt(ne) = n \circ i_L \circ r_\mathcal{D}^{-1} \circ tgt(e) \wedge$
  $type(ne) = wlpair_{tgt} \circ type(e)\},$

The new induced start rule differs from the old one in its RHS and set of NACs. Both are extended to account for new potential conflicts. The new set of NACs, of course, also contains those NACs specified in the durative graph transformation rule in addition to the new locking edges. The locking edge set $E_{RL.fedge,R}$ specifies the creation of a read lock for each forbidden edge. The locking edge set $E_{WL.fedge,R}$ specifies the creation of a write lock for each creation edge. Note that since the locking edges used for forbidden edges are the same as those used for required edges, the creation of an edge is also prevented from being executed concurrently with the deletion or preservation of a parallel edge. However, since the creation of an edge does not imply a read, i.e., the edge is not contained in the LHS, multiple parallel edges can be created concurrently.

The locking edge set $E_{RL.fpair,R}$ specifies the creation of a read lock for each forbidden pair. The creation edge of another rule conflicts with such a forbidden pair if two conditions are met:

1. The connecting node adjacent to the forbidden pair matches the same node in the host graph as a required node adjacent to the creation edge.
2. The edge type and direction of the forbidden pair are the same as that of the creation edge.

Therefore, the locking edge set $E_{WL.fpair,R}$ specifies the creation of a write lock for each pair of creation edge and adjacent required node. Note that it does not matter

whether the node at the second end of the new edge is also new or was available before, i.e., the rule completely builds the forbidden pair into the configuration or the forbidden pair is constructed by connecting two already existing nodes with a new edge.

As with the original definition of the induced start rule in Definition 5.2.10, the rule has NACs to ensure the non-existence of certain locking edges. For each read lock in one of the four sets $E_{RL.fedge,R}$, $E_{WL.fedge,R}$, $E_{RL.fpair,R}$, and $E_{WL.fpair,R}$, there is a write lock in one of the respective sets $\mathcal{N}_{WL.fedge}$, $\mathcal{N}_{RL.fedge}$, $\mathcal{N}_{WL.fpair}$, and $\mathcal{N}_{RL.fpair}$ and vice versa. The sets $\mathcal{N}_{WL.fedge}$ and $\mathcal{N}_{WL.fpair}$ ensure that no forbid-produce conflicts occur, and the sets $\mathcal{N}_{RL.fedge}$ and $\mathcal{N}_{RL.fpair}$ ensure that no produce-forbid conflicts occur.

The induced end rule is extended analogously to the induced start rule, i.e., all locking edges to account for produce-forbid and forbid-produce conflicts are deleted again.

Definition 5.4.3 (Induced end rule supporting NACs)**.** Given a durative graph transformation rule with NACs $\mathcal{D}' = (L_\mathcal{D}, R_\mathcal{D}, r_\mathcal{D}, \mathcal{N}_\mathcal{D}, name, d)$, the *induced end rule (supporting NACs)* of $\mathcal{D}'$ is a timed rule $er' = (L', R, r, \mathcal{N}, z, V_{res})$ that differs from the induced end rule $er = (L, R, r, \mathcal{N}, z, V_{res})$ of a durative graph transformation rule $\mathcal{D} = (L_\mathcal{D}, R_\mathcal{D}, r_\mathcal{D}, name, d)$ in that

- $E_{G,L'} = E_{G,L} \cup E_{RL.fedge,L} \cup E_{WL.fedge,L} \cup E_{RL.fpair,L} \cup E_{WL.fpair,L}$,

- $E_{RL.fedge,L} = \{le | \exists (N_\mathcal{D}, n_\mathcal{D}) \in \mathcal{N}_\mathcal{D} : \exists fe \in E_{N_\mathcal{D}} \setminus E_{L_\mathcal{D}} :$
  $src(fe) \in \text{ran}(n_\mathcal{D}) \wedge tgt(fe) \in \text{ran}(n_\mathcal{D}) \wedge$
  $src(le) = i_L \circ n_\mathcal{D}^{-1} \circ src(fe) \wedge$
  $tgt(le) = i_L \circ n_\mathcal{D}^{-1} \circ tgt(fe) \wedge$
  $type(le) = rledge \circ type(fe)\}$,

- $E_{WL.fedge,L} = \{le | \exists e \in E_{R_\mathcal{D}} \setminus \text{ran}(r_\mathcal{D}) :$
  $src(e) \in \text{ran}(r_\mathcal{D}) \wedge tgt(e) \in \text{ran}(r_\mathcal{D}) \wedge$
  $src(le) = i_L \circ r_\mathcal{D}^{-1} \circ src(e) \wedge$
  $tgt(le) = i_L \circ r_\mathcal{D}^{-1} \circ tgt(e) \wedge$
  $type(le) = wledge \circ type(e)\}$,

- $E_{RL.fpair,L} = \{le | \exists (N_\mathcal{D}, n_\mathcal{D}) \in \mathcal{N}_\mathcal{D} : \exists fe \in E_{N_\mathcal{D}} \setminus E_{L_\mathcal{D}} :$
  $\exists fn \in V_{N_\mathcal{D}} \setminus V_{L_\mathcal{D}} :$
  $src(fe) \in \text{ran}(n_\mathcal{D}) \wedge tgt(fe) = fn \wedge$
  $src(le) = tgt(le) = i_L \circ n_\mathcal{D}^{-1} \circ src(fe) \wedge$
  $type(le) = rlpair_{src} \circ type(fe)\} \cup$
  $\{le | \exists (N_\mathcal{D}, n_\mathcal{D}) \in \mathcal{N}_\mathcal{D} : \exists fe \in E_{N_\mathcal{D}} \setminus E_{L_\mathcal{D}} :$
  $\exists fn \in V_{N_\mathcal{D}} \setminus V_{L_\mathcal{D}} :$
  $src(fe) = fn \wedge tgt(fe) \in \text{ran}(n_\mathcal{D}) \wedge$
  $src(le) = tgt(le) = i_L \circ n_\mathcal{D}^{-1} \circ tgt(fe) \wedge$
  $type(le) = rlpair_{tgt} \circ type(fe)\}$,

- $E_{WL.fpair,L} = \{le | \exists e \in E_{R_\mathcal{D}} \setminus \text{ran}(r_\mathcal{D}) :$
  $src(e) \in \text{ran}(r_\mathcal{D}) \wedge$
  $src(le) = tgt(le) = i_L \circ r_\mathcal{D}^{-1} \circ src(e) \wedge$
  $type(le) = wlpair_{src} \circ type(e)\} \cup$
  $\{le | \exists e \in E_{R_\mathcal{D}} \setminus \text{ran}(r_\mathcal{D}) :$
  $tgt(e) \in \text{ran}(r_\mathcal{D}) \wedge$
  $src(le) = tgt(le) = i_L \circ r_\mathcal{D}^{-1} \circ tgt(e) \wedge$
  $type(le) = wlpair_{tgt} \circ type(e)\},$

The addition of the support for NACs does not invalidate the properties formalized in Section 5.3. It affects only those lemmas that reason over possible conflicts between induced start and end rules, i.e., Lemmas 5.3.3 to 5.3.5. Most of their proofs work via the existence of locking edges and rely on the fact that each creation of a read lock [write lock] is accompanied by a NAC that realizes a check for a write lock [read lock] and vice versa. This holds for both, use and deletion, see Definition 5.2.10, as well as forbiddance and creation, see Definition 5.4.2. Technically, some of the proofs have to be extended to handle possible produce-forbid (and forbid-produce) conflicts. This extension can be made analogously to the handling of delete-use and (use-delete) conflicts.

5.5 Concurrency Rules

The application of a durative graph transformation rule limits which other durative graph transformation rules may be applied concurrently. This excluding effect on the applicability of rules can be seen as a *negative* kind of influence between durative graph transformations. Specifying a *positive* kind of influence between durative graph transformations is not possible with durative graph transformation rules alone. This is what concurrency rules are used for.

A concurrency rule provides a means of specifying that certain durative graph transformations *enable* the execution of other durative graph transformations. Concurrency rules are not applied in the sense of a graph transformation; they simply specify a dependency between different durative rules. This specification modifies the induced rules of all involved durative rules such that the application of some of these durative rules requires the concurrent application of others.

The use of concurrency rules increases the expressiveness of the DGTS formalism. With concurrency rules, it is possible to specify systems that require the concurrent application of durative rules instead of only allowing their concurrent application.

As with durative rules, the semantics of concurrency rules is specified using the TGTS formalism. From the perspective of the TGTS formalism, concurrency rules make required concurrency explicit. This is beneficial for modeling concurrent systems because reconfiguration and concurrency can thus be dealt with orthogonally.

We consider an example from the RailCab domain. As stated in Section 1.4, this domain includes base stations, which monitor track segments of the railway network. At any time during the movement of a RailCab, it has to be registered at such a

base station. More precisely, a RailCab has to be registered at a base station that monitors the track segment that the RailCab is currently occupying. The real-time communication between a RailCab and the base station it is registered at is specified by the RTCP Publication, which is represented within configurations and rules as a node of type `Pub`.

When a RailCab moves from a track segment that is monitored by one base station to a track segment that is monitored by another base station, it has to change its publication. This is captured by the durative rule `changePublication`, which is shown in Figure 5.13. It specifies the revocation of a publication at one base station and the announcement of a publication at another base station.

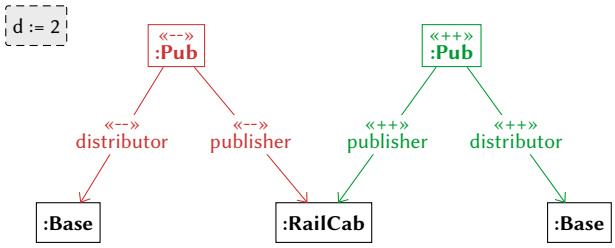


Figure 5.13: Durative rule `changePublication`

A condition for the application of the durative rule `changePublication` is the concurrent application of another durative rule that moves the RailCab from one track segment to the next. Besides `moveRailCab`, possible candidates providing such a reconfiguration are `moveConvoy` and all durative rules related to membership change, e.g., `formConvoy` or `joinConvoy`, since they also change the position of RailCabs. All these durative rules are modeled independently from `changePublication` and their applications exist independently from applications of `changePublication`.

Including the movement of a RailCab in `changePublication` is not advisable due to two reasons. First, there is more than one reconfiguration that moves a RailCab to the next track segment. A modeler would have to model a separate rule for each such reconfiguration. Second, changing a publication and moving a RailCab are different concerns, and modeling them as one reconfiguration can be considered bad development style.

Since reconfigurations addressing different concerns are modeled independently from one another, we need an external means of specifying requirements for their concurrent execution. Concurrency rules provide this means by referencing durative rules and specifying how these rules have to match relatively to one another.

5.5.1 Syntax

A concurrency rule specifies a dependency between two sets of durative rules: an application of a durative rule in the first set requires a concurrent application of a durative rule in the second set. Vice versa, the application interval of a durative rule in the second set can be seen as a window of opportunity for rules in the first

set. The involved durative rules of both sets have to match in a certain way for this dependency to be fulfilled. This matching constraint is formalized in the syntax of concurrency rules via two interface graphs and graph morphisms to the involved durative rules.

Definition 5.5.1 (Concurrency rule)**.** Let $\mathcal{DR}$ be a set of durative rules. A *concurrency rule* $\mathcal{C} = (G^T, D^T, S^T, D, S, name)$ consists of

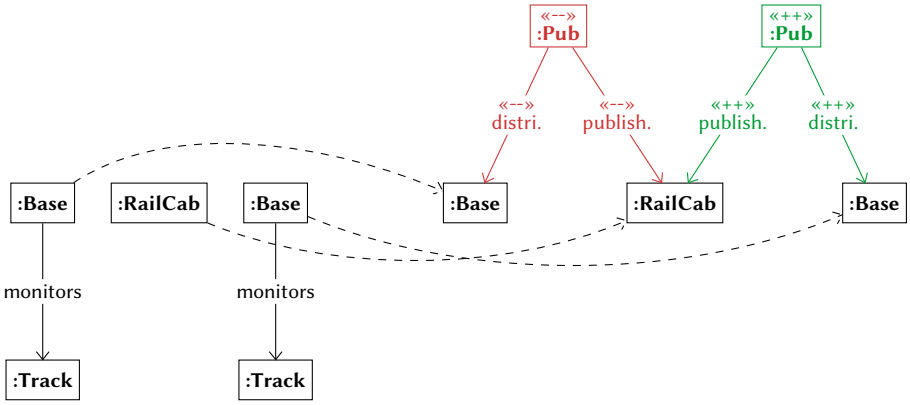
- a typed graph G^T, called *connecting graph*, with two subgraphs D^T and S^T, called *(concurrency) demander interface* and *(concurrency) satisfier interface*, respectively,
- a non-empty set of tuples D, called *(concurrency) demander tuples*, where each tuple $(\mathcal{D}, d) \in D$ references a durative rule $\mathcal{D} \in \mathcal{DR}$ and determines a subgraph of its LHS via an injective morphism $d : D^T \to L_{\mathcal{D}}$, called *(concurrency) demander constraint morphism*,
- a non-empty set of tuples S, called *(concurrency) satisfier tuples*, where each tuple $(\mathcal{D}, s) \in S$ references a durative rule $\mathcal{D} \in \mathcal{DR}$ and determines a subgraph of its LHS via an injective morphism $s : S^T \to L_{\mathcal{D}}$, called *(concurrency) satisfier constraint morphism*, and
- a distinct name *name*.

For a durative graph transformation system $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR})$ with a set of concurrency rules $\mathcal{CR}$, we also write $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR})$.

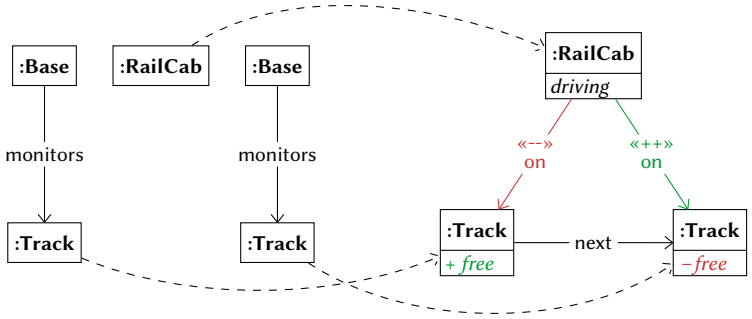
A connecting graph has two dedicated subgraphs, which serve as interfaces for the durative rules involved with a concurrency rule. They are called *demander interface* and *satisfier interface*. So-called *constraint morphisms* from these subgraphs to durative rules define which durative rules fulfill these interfaces and how they have to match relatively to each other. All these constraint morphisms are – together with the rules they map to – contained in the set of demander and satisfier tuples. Each rule referenced by a demander or satisfier tuple fulfills the demander or satisfier interface, respectively.

An application of a durative rule referenced by a demander tuple *demands* a concurrent transformation, and an application of a durative rule referenced by a satisfier tuple *satisfies* this demand. Therefore, we call a durative rule that is referenced by a demander tuple a *demanding rule*. If it is referenced by a satisfier tuple, we call it a *satisfying rule*.

The demander and satisfier constraint morphisms constitute matching constraints for all involved durative rules. For a demanding and a satisfying rule to be applicable concurrently, elements contained in the images of their demander and satisfier constraint morphism that originate from the same element in the connecting graph G^T also have to match to the same element in the host graph. Elements that have a preimage in only one of the interface subgraphs D^T and S^T also restrict the matching: if an element of D^T is somehow connected in G^T to an element of S^T, then the same connection has to exist for their images in the host graph.



(a) A demander constraint morphism of `allowChangePublication` to `changePublication`



(b) A satisfier constraint morphism of `allowChangePublication` to `moveRailCab`

Figure 5.14: A demander and a satisfier constraint morphism of the concurrency rule `allowChangePublication`

As an example, Figure 5.14 shows a demander and a satisfier constraint morphism for the concurrency rule `allowChangePublication`. A RailCab is only allowed to change its publication if it is moving from one track segment to the next. Therefore, `changePublication` is a demanding rule and `moveRailCab` a satisfying rule. To be precise, `changePublication` is the *only* demanding rule, while `moveRailCab` is one of *multiple* satisfying rules. All durative rules that move a RailCab from one track segment to the next are valid satisfying rules for `allowChangePublication`. Here, `moveRailCab` is exemplary for all satisfying rules of `allowChangePublication`.

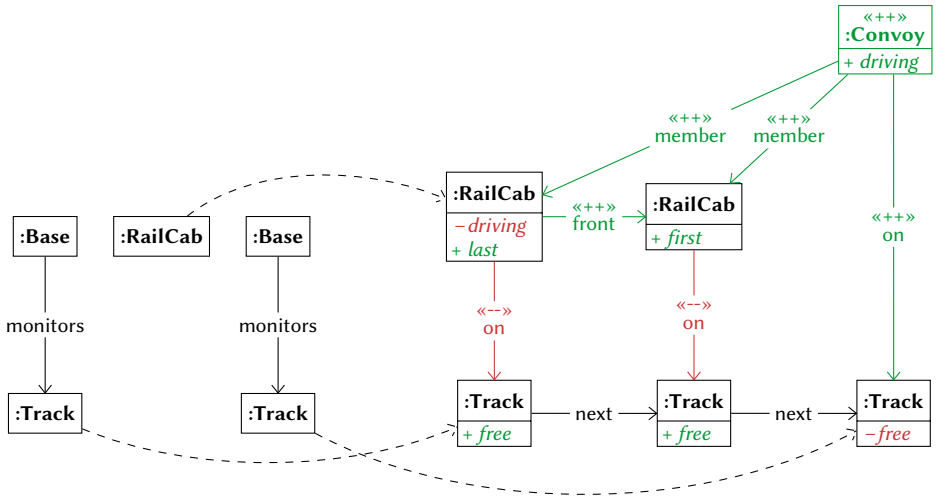
The relevant node in this example is the `RailCab` node, which is why it is contained in both interface subgraphs and thus defined under both demander and satisfier constraint morphisms. However, the fact that the `RailCab` nodes of both rules have to match the same node in the host graph is not the only matching constraint for the concurrent application of both rules. The new base station also has to monitor the track segment that the RailCab is moving to. This is neither specified in `changePublication` nor in `moveRailCab`. It is not specified in `changePublication`,

because changePublication is not concerned with the movement of RailCabs at all, and it is not specified in moveRailCab, because moveRailCab is not concerned with base stations and publications. Instead, this constraint is expressed via the structure of the connecting graph and both interface subgraphs: each Base node is connected to one of the Track nodes via a monitors edge, and while the Base nodes are contained in the demander interface, the Track nodes are contained in the satisfier interface. For the concurrent application of both rules, this structure also has to exist in the host graph.

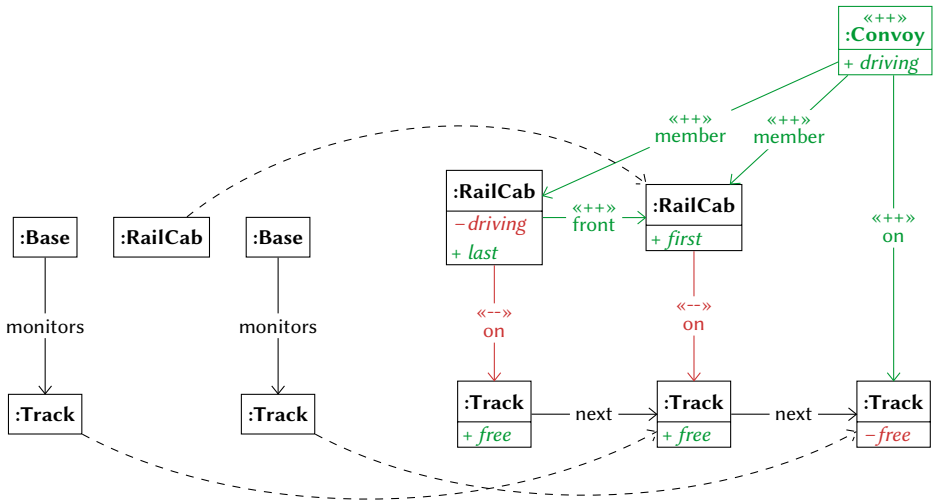
Note that there can be multiple demander or satisfier constraint morphisms to the same demanding or satisfying rule. If there are multiple demander constraint morphisms to a single demanding rule, this means that the demander interface is fulfilled in multiple different ways, each with a different matching constraint, and an application of this rule causes multiple demands. If there are multiple satisfier constraint morphisms to a single satisfying rule, this means that the satisfier interface can be fulfilled in multiple different ways, i.e., multiple matches can lead to a satisfaction of the demand in concurrent execution. An example for this is the rule formConvoy, which models two RailCabs driving in the same direction. In this rule, a rearward RailCab catches up to a frontward RailCab by covering a distance of two track segments. Figure 5.15 shows two satisfier constraint morphisms mapping to this rule. The first constraint morphism maps the concurrency rule's RailCab node to the rearward RailCab and the second constraint morphism to the frontward RailCab. Note that the second Track node does not have to be the direct successor of the first Track node for the constraint morphisms to match.

When employing concurrency rules in software development, a modeler potentially has to define a lot of demander and satisfier constraint morphisms. While these constraint morphism can be defined conveniently using colors or highlighting, a graphical representation of a concurrency rule involving multiple constraint morphisms, like the ones in Figures 5.14 and 5.15, is rather impractical as an overview because it is not presented coherently in a single diagram. Fortunately, the latter can be done: by employing object names in durative rules, we allow to reference their nodes directly from the connecting graph of a concurrency rule.

Figure 5.16 illustrates such a compact representation for the constraint morphisms of Figures 5.14 and 5.15. As an example, consider the RailCab node in the center of the connecting graph. The label *#dc::'rc'* means that the node maps to the node with the object name rc under the constraint morphism *dc*, which is a demander constraint morphism to the durative rule changePublication. The other three labels define images of this node under the three satisfier constraint morphisms of Figures 5.14(b), 5.15(a) and 5.15(b).



(a) First satisfier constraint morphism of `allowChangePublication` to `formConvoy`



(b) Second satisfier constraint morphism of `allowChangePublication` to `formConvoy`

Figure 5.15: Two satisfier constraint morphisms of concurrency rule `allowChangePublication` mapping to the same durative rule

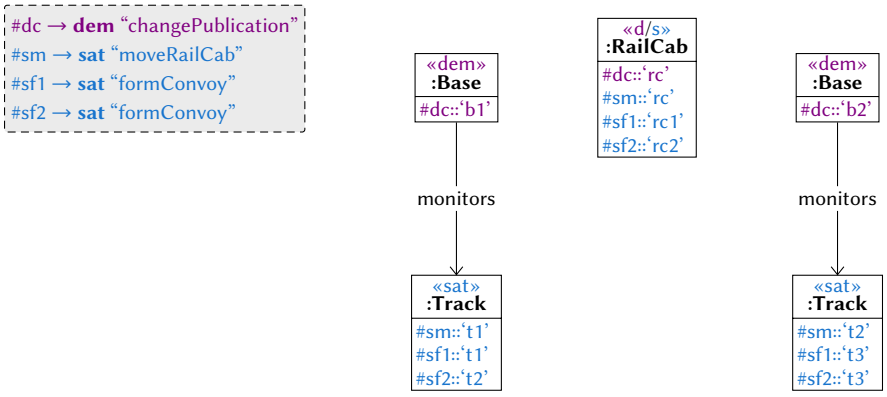


Figure 5.16: Compact representation of demander and satisfier constraint morphisms of Figures 5.14 and 5.15 for concurrency rule `allowChangePublication`

5.5.2 Semantics

The semantics of concurrency rules is defined by extending those timed graph transformation rules that have been induced by durative graph transformation rules. This can be seen as modifying the semantics of all durative rules involved in concurrency rules. Each start or end rule whose inducing durative rule is referenced by a concurrency rule is extended. To motivate how these rules are extended, we take a look at their interaction.

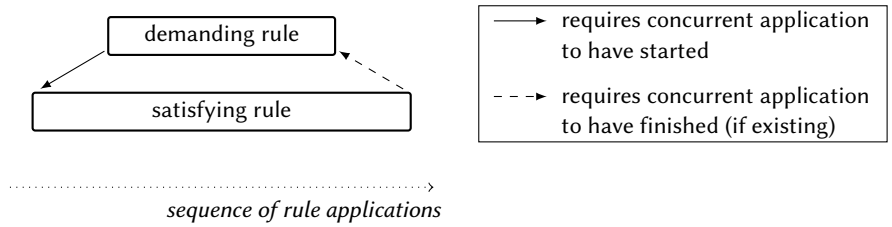


Figure 5.17: Concurrent execution of a demanding and a satisfying rule

Figure 5.17 illustrates the dependency between a demanding and a satisfying rule, e.g., `changePublication` and `moveRailCab`. In terms of time, the demanding rule has the "inner" and the satisfying rule the "outer" application interval. The semantics of a concurrency rule extends the induced start and end rules of all involved durative rules such that their application times have to be temporally ordered as in the figure. More precisely, the demanding rule requires the application of the satisfying rule both to have started earlier and to end later. To require the former, we extend the induced rules of both rules such that the demanding transformation checks whether the satisfying transformation is currently being applied. When requiring the latter, we have to make sure that the current satisfying transformation is indeed the same transformation as before. To prevent a second satisfier transformation (of the same or a different rule) from taking the place of the first, we apply a lock and reverse the direction of the dependency, i.e., by checking for potential locks, the satisfying rule guarantees that no demanding transformation is being applied concurrently. Note that a satisfying rule can still be applied independently of a demanding rule, which is why the right arrow in Figure 5.17 has a different meaning than the left.

To properly extend the induced rules, we have to be able to check whether a demand in concurrent execution, as specified by a concurrency rule, is satisfied. The satisfaction of such a demand is indicated by a *satisfaction indicator*, which is a concept that is analogous to an application indicator. Since concurrency rules are not applied in the sense of graph transformations, satisfaction indicators are not created and deleted by concurrency rules but by their referenced satisfying rules.

As a consequence of the use of satisfaction indicators, the induced TGTS type graph has to be extended. This extension is made analogously to that of application indicators. The TGTS type graph has to include a type for each satisfaction indicator. For a concurrency rule with the name *name*, its satisfaction indicator type is given by $siType(name)$. Furthermore, there has to be a distinct satisfaction indicator edge

type for each node in the satisfier interface of the concurrency rule. For a node v, its satisfaction indicator edge type is given by $siEdgeType(v)$.

For a satisfying rule, a satisfaction indicator is simply attached via satisfaction indicator edges to those nodes that are in the range of its satisfier constraint morphism. Unfortunately, the appearance of satisfaction indicators in demanding rules is slightly more complicated than in satisfying rules. Since demander and satisfier constraint morphisms are defined under different subgraphs of the connecting graph, those nodes that the satisfaction indicator has to be attached to do not necessarily exist in a demanding rule's LHS. Therefore, the LHSs of a demanding rule's induced start and end rule are extended with those elements in the connecting graph that are not defined under the demander constraint morphism. Technically, this is implemented via a pushout.

Adding these elements to the induced rules' LHSs does not restrict the applicability of the demanding rule. The added elements have to exist in the host graph in either case because the satisfying rule, which is applied concurrently, requires them.

Next, we give definitions that state how the induced rules of demanding and satisfying rules are extended. After each definition, we given an example of the extended induced rule. These extensions follow the example of the concurrency rule `allowChangePublication` with `changePublication` as demanding rule and `moveRailCab` as satisfying rule.

First, we extend the induced start and end rule of a concurrency demanding rule such that they require the existence of a satisfaction indicator in the host graph.

Definition 5.5.2 (Extension of a concurrency demander's induced start rule)**.** Let $\mathcal{C} = (G^T, D^T, S^T, D, S, name)$ be a concurrency rule and $\mathcal{DR}$ a set of durative rules. For each concurrency demander tuple $(\mathcal{D}, d) \in D$, the induced start rule $sr = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $sr' = (L', R', r', \mathcal{N}', z, V_{res})$ where

- (L^x, g, i^*), with $g : G^T \to L^x$ and $i^* : L \to L^x$, is the pushout over $d : D^T \to L$ and the subgraph isomorphism $i : D^T \to G^T$,
- $V_{SI} = \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{SI} = \{e | src(e) = si \wedge tgt(e) \in g(S^T) \wedge$
  $type(e) = siEdgeType \circ g^{-1} \circ tgt(e)\}$,
- $V_{G,L'} = V_{G,L^x} \cup V_{SI} \wedge E_{G,L'} = E_{G,L^x} \cup E_{SI}$,
- $V_{G,R'} = V_{G,R} \cup V_{SI} \wedge E_{G,R'} = E_{G,R} \cup E_{SI} \cup E_{RL.node,R'}$,
- $V_{CI,L'} = V_{CI,L^x} \wedge E_{CI,L'} = E_{CI,L^x} \wedge V_{CI,R'} = V_{CI,R} \wedge E_{CI,R'} = E_{CI,R}$
- $r' = r \cup \{\forall x \in V_{g(G^T \setminus D^T)} \cup E_{g(G^T \setminus D^T)} : x \mapsto x\} \cup$
  $\{si \mapsto si\} \cup \{\forall y \in V_{g(S^T)} : (si, y) \mapsto (si, y)\}$, and
- $\mathcal{N}'$ is defined analogously to Definition 5.2.10 such that $\forall (N, n) \in \mathcal{N}'$: $\mathrm{dom}(n) = L'$, and
- $E_{RL.node,R'} = \{e | src(e) = tgt(e) = si \wedge type(e) = rlnode \circ type(si)\}$

The pushout adds those elements that are existing in the connecting graph but not in the subgraph constituting the demander interface to the LHS of the extended start rule. This is necessary so that we can attach the satisfaction indicator at its appropriate place. Besides requiring this satisfaction indicator and its satisfaction indicator edges, the extended start rule creates a read lock on the satisfaction indicator. The read lock is used to ensure that the execution of a demanding rule finishes before the execution of a satisfying rule. Apart from that, the extended start rule is almost identical to the original rule. The rule morphism and the NAC morphisms are changed such that they are total on their new domain L'. This is necessary only for reasons of technical correctness; it does not change their intended purpose.

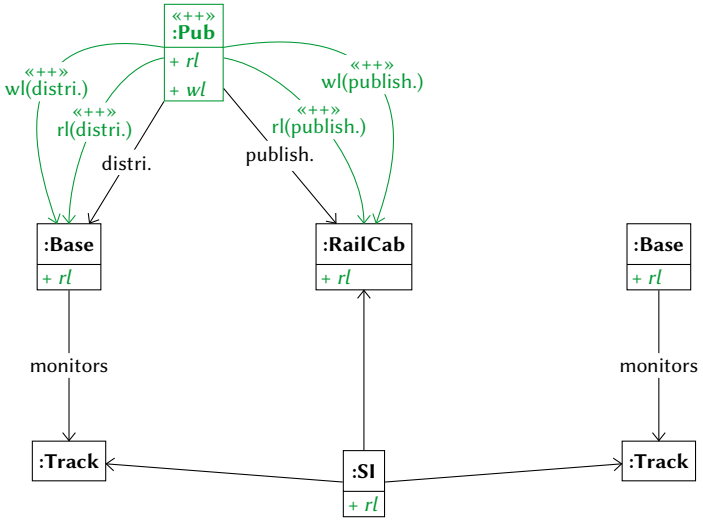


Figure 5.18: Demanding rule `changePublication`'s induced start rule extended according to concurrency rule `allowChangePublication`

Figure 5.18 shows the extended induced start rule of `changePublication`. The extension was done according to the demander constraint morphism of Figure 5.14(a). For reasons of clarity, the extended rule does not show any NACs or locks that have been generated to support NACs on the level of durative rules. The two `Track` nodes along with the two `monitors` edges, which connect the `Track` nodes to the `Base` nodes, have been added into the rule by the pushout. Then, the satisfaction indicator is connected to these `Track` nodes and the only `RailCab` node. The satisfaction indicator also receives a read lock. Note that the new `Track` nodes do not have any locks, because they were not present in the original rule.

The end rule of a concurrency demanding rule is extended in a similar manner as the start rule. The LHS and RHS include those elements from the connecting graph needed for the satisfaction indicator, the satisfaction indicator itself, and its satisfaction indicator edges. The LHS also includes a read lock on the satisfaction indicator.

Definition 5.5.3 (Extension of a concurrency demander's induced end rule)**.** Let $\mathcal{C} = (G^T, D^T, S^T, D, S, name)$ be a concurrency rule and $\mathcal{DR}$ a set of durative rules. For each concurrency demander tuple $(\mathcal{D}, d) \in D$, the induced end rule $er = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $er' = (L', R', r', \mathcal{N}, z, V_{res})$ where

- (L^x, g, i^*), with $g : G^T \to L^x$ and $i^* : L \to L^x$, is the pushout over $d : D^T \to L$ and the subgraph isomorphism $i : D^T \to G^T$,
- $V_{SI} = \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{SI} = \{e | src(e) = si \wedge tgt(e) \in g(S^T) \wedge$
  $type(e) = siEdgeType \circ g^{-1} \circ tgt(e)\}$,
- $V_{G,L'} = V_{G,L^x} \cup V_{SI} \wedge E_{G,L'} = E_{G,L^x} \cup E_{SI} \cup E_{RL.node,L'}$,
- $V_{G,R'} = V_{G,R} \cup V_{SI} \wedge E_{G,R'} = E_{G,R} \cup E_{SI}$,
- $V_{CI,L'} = V_{CI,L^x} \wedge E_{CI,L'} = E_{CI,L^x} \wedge V_{CI,R'} = V_{CI,R} \wedge E_{CI,R'} = E_{CI,R}$
- $r' = r \cup \{\forall x \in V_{g(G^T \setminus D^T)} \cup E_{g(G^T \setminus D^T)} : x \mapsto x\} \cup$
  $\{si \mapsto si\} \cup \{\forall y \in V_{g(S^T)} : (si, y) \mapsto (si, y)\}$, and
- $E_{RL.node,L'} = \{e | src(e) = tgt(e) = si \wedge type(e) = rlnode \circ type(si)\}$.

Figure 5.19 shows the extended induced end rule of `changePublication`. Its extension is done analogously to that of the induced start rule in Figure 5.18.

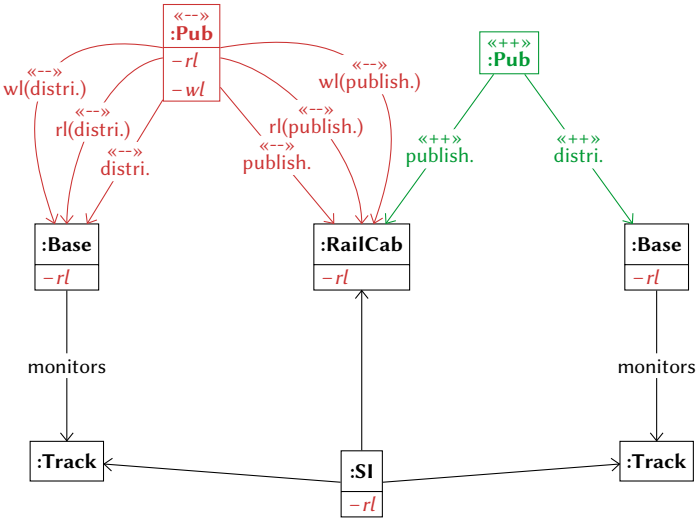


Figure 5.19: Demanding rule `changePublication`'s induced end rule extended according to concurrency rule `allowChangePublication`

The induced rules of a demanding rule require the existence of a satisfaction indicator. The only rules able to create (and delete) this specific satisfaction indicator are induced start (and end) rules of those satisfying rules that are referenced by the same concurrency rule as the demanding rule.

Now, we define how the induced start and end rules of satisfying rules are extended such that they create and delete appropriate satisfaction indicators.

Definition 5.5.4 (Extension of a concurrency satisfier's induced start rule)**.** Let $\mathcal{C} = (G^T, D^T, S^T, D, S, name)$ be a concurrency rule and $\mathcal{DR}$ a set of durative rules. For each concurrency satisfier tuple $(\mathcal{D}, s) \in S$, the induced start rule $sr = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $sr' = (L, R', r, \mathcal{N}, z, V_{res})$ where

- $V_{G,R'} = V_{G,R} \cup \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{G,R'} = E_{G,R} \cup \{e | src(e) = si \wedge tgt(e) \in \text{ran}(s) \wedge$
  $type(e) = siEdgeType \circ s^{-1} \circ tgt(e)\}$,
- $V_{CI,R'} = V_{CI,R} \wedge E_{CI,R'} = E_{CI,R}$

Figure 5.20 shows the extended induced start rule of `moveRailCab`. This extension is done according to the satisfier constraint morphism of Figure 5.14(b). Here, the satisfaction indicator is simply added to the available `Track` and `RailCab` nodes.

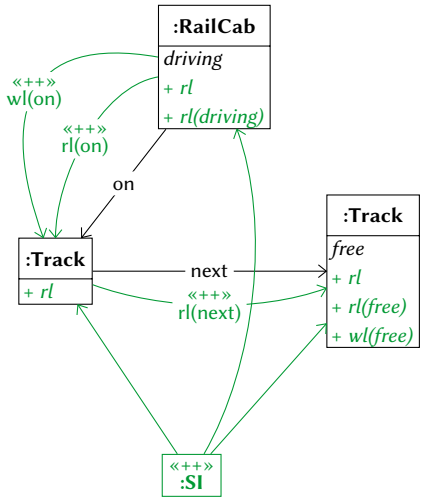


Figure 5.20: Satisfying rule `moveRailCab`'s induced start rule extended according to concurrency rule `allowChangePublication`

In addition to a satisfaction indicator, the induced end rule of a satisfying rule is extended with a NAC, which forbids the existence of a read lock on the satisfaction indicator. This ensures that the execution of each applied demanding rule finishes before that of the satisfying rule does.

Definition 5.5.5 (Extension of a concurrency satisfier's induced end rule)**.** Let $\mathcal{C} = (G^T, D^T, S^T, D, S, name)$ be a concurrency rule and $\mathcal{DR}$ a set of durative rules. For each concurrency satisfier tuple $(\mathcal{D}, s) \in S$, the induced end rule $er = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $er' = (L', R, r, \mathcal{N}', z, V_{res})$ where

- $V_{G,L'} = V_{G,L} \cup \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{G,L'} = E_{G,L} \cup \{e | src(e) = si \wedge tgt(e) \in \text{ran}(s) \wedge$
  $type(e) = siEdgeType \circ s^{-1} \circ tgt(e)\}$,

- $V_{CI,L'} = V_{CI,L} \wedge E_{CI,L'} = E_{CI,L}$, and

- $\mathcal{N}' = \{(N, n)\} \wedge V_N = V_{G,L} \wedge E_N = E_{G,L} \cup \{ne\} \wedge$
  $src(ne) = tgt(ne) = n(si) \wedge$
  $type(ne) = wlnode \circ type(si) \wedge n$ is injective.

Figure 5.21 shows the extended induced end rule of `moveRailCab`. Its extension is done almost analogously to that of the induced start rule in Figure 5.20. In addition, the satisfaction indicator is extended with a NAC that ensures that there is no concurrent application of a demanding rule anymore.

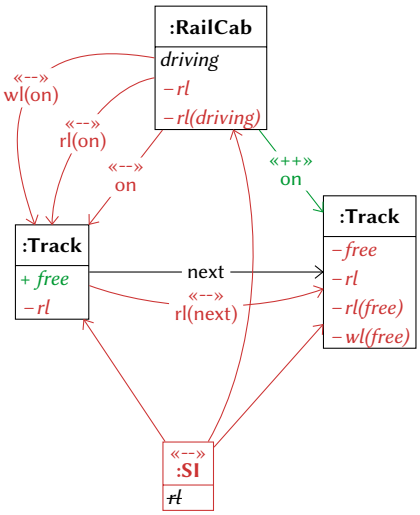


Figure 5.21: Satisfying rule `moveRailCab`'s induced end rule extended according to concurrency rule `allowChangePublication`

Finally, we can define the semantics of a durative graph transformation system with concurrency rules. As with a durative graph transformation system without concurrency rules, it is given by its induced timed graph transformation system. Since Definition 5.2.18 does not regard concurrency rules, we have to provide a new definition for a durative graph transformation system with concurrency rules.

Definition 5.5.6 (Induced timed graph transformation system respecting concurrency rules)**.** Let $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR})$ be a durative graph transformation system that contains a set of concurrency rules $\mathcal{CR}$ and $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ its induced timed graph transformation system according to Definition 5.2.18. Its *induced timed graph transformation system respecting concurrency rules* $\mathcal{TS}' = (TG', TiG_0, TR', IR, CR)$ differs from $\mathcal{TS}$ in that

- the induced TGTS type graph TG has been extended into a type graph TG' that contains a satisfaction indicator type for each concurrency rule in $\mathcal{CR}$ as well as their satisfaction indicator edge types and
- each timed rule $tr \in TR$ whose inducing durative rule $\mathcal{D}$ is referenced by a (concurrency) demander or (concurrency) satisfier tuple of a concurrency rule in $\mathcal{CR}$ has been extended into a timed rule $tr' \in TR'$ as defined in Definitions 5.5.2 to 5.5.5, and if $\mathcal{D}$ is referenced by multiple (concurrency) demander or (concurrency) satisfier tuples (of one or more concurrency rules in $\mathcal{CR}$), then the timed rule is extended successively.

5.6 Urgency Rules

With durative graph transformation rules and concurrency rules, we have dealt with two different kinds of influence among durative graph transformations: potential conflicts and required concurrency. The locking mechanism of durative rules takes care of potential conflicts between transformations, and concurrency rules allow to specify when a concurrent application of certain durative rules is mandatory for the application of others. A third kind of influence between durative graph transformations is urgency, which can be specified in the DGTS formalism via urgency rules.

An urgency rule provides a means of specifying that certain durative graph transformations require an urgent execution of subsequent durative graph transformations. Between these transformations, only a certain amount of time is allowed to pass. Like concurrency rules, urgency rules are not applied in the sense of graph transformations. They modify some existing induced rules and induce further rules.

The benefits of using urgency rules are similar to those of concurrency rules. First, like concurrency rules, urgency rules increase the expressiveness of the DGTS formalism. Second, while the semantics of urgency rules is completely expressible via rules of the TGTS formalism, using urgency rules makes the specification of urgency explicit, which improves the readability of the specification.

Again, we consider an example from the RailCab domain. In this domain, a RailCab is not allowed to stop abruptly if it is in driving motion. Such an abrupt stop would be unrealistic because slowing down a RailCab takes time. Therefore, we differentiate accelerating and braking from the ordinary movement of a RailCab (or a convoy). Figures 5.22 and 5.23 show the durative graph transformation rules `moveRailCab` and `brakeRailCab`. Both rules move a RailCab from on track segment to the next. Besides having different execution times, these rules differ in the state of the RailCab: in `moveRailCab`, the RailCab is in driving motion in both its LHS and RHS; in `brakeRailCab`, it is in driving motion only in its LHS. In the rule `accelerateRailCab`, which was shown in Figure 5.12, the RailCab is in driving motion only in its RHS.

Modeling whether a RailCab is in driving motion or not does not yet remove paths from the state space where RailCabs stop abruptly. Such paths still exist because it is still possible to apply a delay transition in a state where a RailCab is in

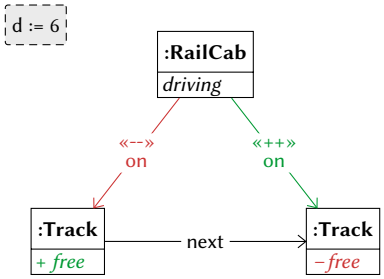


Figure 5.22: Durative rule `moveRailCab`

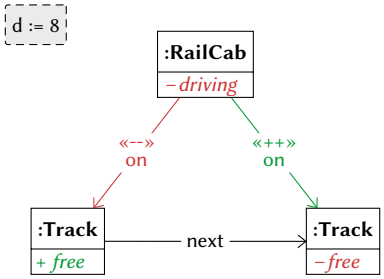


Figure 5.23: Durative rule `brakeRailCab`

driving motion. To prevent the application of a delay transition to such a state, the consumption of time has to be disallowed. Technically, this can be done by means of invariant rules.

Urgency rules lift the functionality of invariant rules to the level of durative rules by providing a convenient means of specifying the requirement that the time that is allowed to pass between the successive application of two durative rules may not exceed a certain amount of time units. As opposed to invariant rules, urgency rules also explicitly indicate all involved durative rules.

The concept of urgency rules is inspired by that of urgent locations, cf. [BDL04], and that of urgent transitions, cf. [BST99; BT04]. However, it neither corresponds *precisely* to that of urgent locations nor urgent transitions.

An urgent location is a location where time is not allowed to pass. Its semantics is implemented via an invariant on a clock specific to the location, which is reset on all incoming edges. Therefore, the incoming and outgoing transitions of a location do not matter; firing any incoming transition leads to the urgent location, and firing any outgoing transition is sufficient for leaving it in time. The difference of urgency rules to urgent locations is that an urgency rule explicitly indicates all involved durative rules.

An urgent transition, as defined in [BST99], is a transition that is fired as soon as it is enabled. This is similar to urgency satisfying rules. However, they are not applied urgently to every configuration but only to certain configurations, which are defined by the urgency rule. There are other approaches where the semantics for urgent transitions deviates from that of [BST99]. In [BT04], an urgent transition

has to be fired within a given time frame from being enabled, and urgent transitions have priority over non-urgent transitions. In the case of urgency rules, specifying a time frame is possible, but there are no priorities.

5.6.1 Syntax

While semantically different, the syntax of urgency rules is very similar to that of concurrency rules. As with concurrency rules, an urgency rule specifies a dependency between two sets of durative graph transformation rules, a set of demanding rules and a set of satisfying rules, and they have to match in a certain way for the dependency to be fulfilled. An application of a demanding rule requires an urgent application of a satisfying rule, i.e., only delays up to a certain maximum of time units, which can possibly be zero, between the two rule applications are allowed.

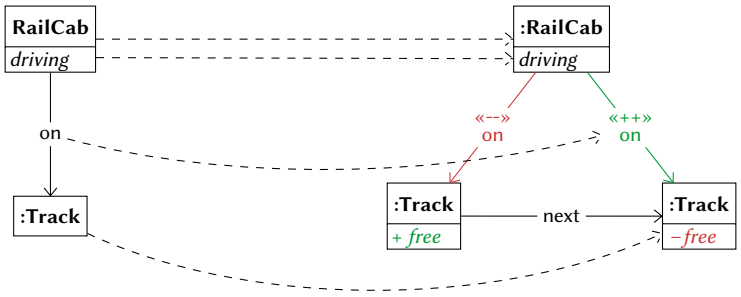
Definition 5.6.1 (Urgency rule)**.** Let $\mathcal{DR}$ be a set of durative rules. An *urgency rule* $\mathcal{U} = (G^T, D^T, S^T, D, S, name, dl)$ consists of

- a typed graph G^T, called *connecting graph*, with two subgraphs D^T and S^T, called *(urgency) demander interface* and *(urgency) satisfier interface*, respectively,
- a non-empty set of tuples D, called *(urgency) demander tuples*, where each tuple $(\mathcal{D}, d) \in D$ references a durative rule $\mathcal{D} \in \mathcal{DR}$ and determines a subgraph of its RHS via an injective morphism $d : D^T \to R_\mathcal{D}$, called *(urgency) demander constraint morphism*,
- a non-empty set of tuples S, called *(urgency) satisfier tuples*, where each tuple $(\mathcal{D}, s) \in S$ references a durative rule $\mathcal{D} \in \mathcal{DR}$ and determines a subgraph of its LHS via an injective morphism $s : S^T \to L_\mathcal{D}$, called *(urgency) satisfier constraint morphism*,
- a distinct name *name*, and
- a deadline $dl \in \mathbb{N}$.

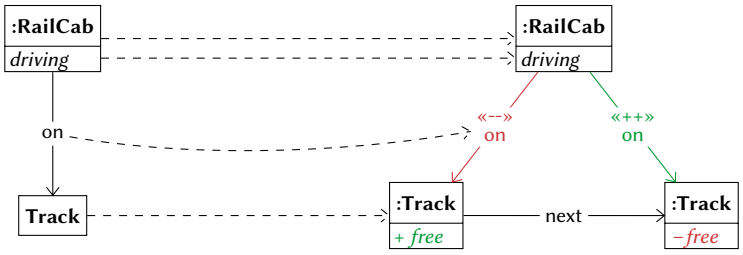
For a durative graph transformation system $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR})$ with a set of concurrency rules $\mathcal{CR}$ and a set of urgency rules $\mathcal{UR}$, we also write $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR})$.

Like a concurrency rule, the connecting graph of an urgency rule has two dedicated subgraphs, which serve as interfaces for the demanding and satisfying rules and as domains for their constraint morphisms. However, as opposed to concurrency rules, each demander constraint morphism maps to the RHS of its demanding rule, not its LHS. Abstractly speaking, this is because the moment in which an urgent execution of a subsequent durative graph transformation is necessary is past the execution of the former durative graph transformations has ended. Remember that the purpose of these constraint morphisms is to formalize matching constraints for the involved durative rules. Such matching constraints have to be taken into consideration by the application of the durative rule that starts

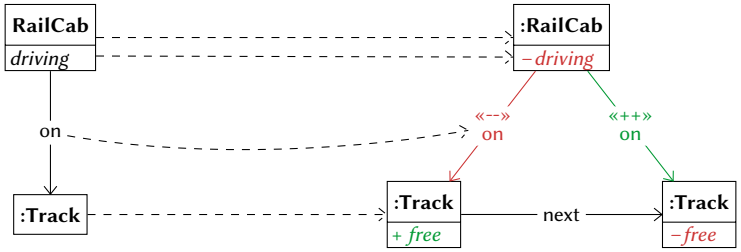
second. In the case of urgency rules, the satisfying rule starts second. Therefore, the matching constraints are regarded when the application of the satisfying rule starts. Since this is after the application of the demanding rule has finished already, the demander constraint morphism maps to its RHS.



(a) A demander constraint morphism of `immediatelyMoveRailCab` to `moveRailCab`



(b) A satisfier constraint morphism of `immediatelyMoveRailCab` to `moveRailCab`



(c) A satisfier constraint morphism of `immediatelyMoveRailCab` to `brakeRailCab`

Figure 5.24: Demander and satisfier constraint morphisms of urgency rule `immediatelyMoveRailCab`

An example is the urgency rule `immediatelyMoveRailCab`. Figure 5.24 shows a demander and two satisfier constraint morphisms for this rule. If a RailCab is in driving motion, it is not possible to stop it abruptly. First, the RailCab needs to perform a braking maneuver. Therefore, each durative rule that has an RHS with a RailCab in driving motion is a demanding rule, and each durative rule that has an LHS with a RailCab in driving motion is a satisfying rule. The constraint morphisms shown in Figure 5.24 are exemplary for all these durative rules. Note that rules can be referenced as both demanding rules and satisfying rules, as is done with

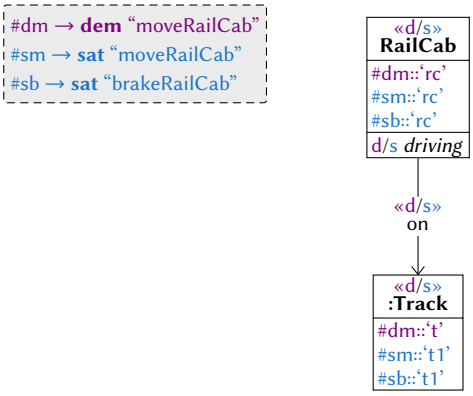


Figure 5.25: Compact representation of demander and satisfier constraint morphisms of Figure 5.24 for urgency rule `immediatelyMoveRailCab`

`moveRailCab` in Figure 5.24. When referencing the same rule as demanding and satisfying rule, the demander and satisfier constraint morphisms should be different, which is the case here; otherwise, a rule application would satisfy its demand itself.

A compact representation for these three constraint morphisms is shown in Figure 5.25. As opposed to the constraint morphisms of the concurrency rule `allowChangePublication`, for which a compact representation was shown in Figure 5.16, the constraint morphisms here also involve edges. Fortunately, we do not have to state the image of an edge under each constraint morphism. It is sufficient to state which edge is involved in the demander and satisfier interface because the correct source and target nodes of the edge's image under each constraint morphism can be deduced from the context, i.e., from the connecting graph and the nodes' images.

In the example given here, the demander and satisfier interface are identical. In general, the demander and satisfier interface of urgency rules are, of course, also allowed to be different. An example where they are required to be different might be the release of a driver's safety belt and the subsequent unlocking of the driver's door.

5.6.2 Semantics

As in concurrency rules, the semantics of urgency rules are defined by extending those start and end rules whose durative rules are referenced by urgency rules. In contrast to concurrency rules, urgency rules also induce new timed rules, invariant rules, and clock instance rules directly. To motivate their purpose, we take an abstract look at how the semantics of an urgency rule is implemented.

Figure 5.26 illustrates how the application of a satisfying rule, e.g., `brakeRailCab`, is enforced by the application of a demanding rule, e.g., `moveRailCab`. First, the demanding rule indicates a demand in urgent execution. This is done by adding a demand indicator into the host graph. To require that the demand is satisfied,

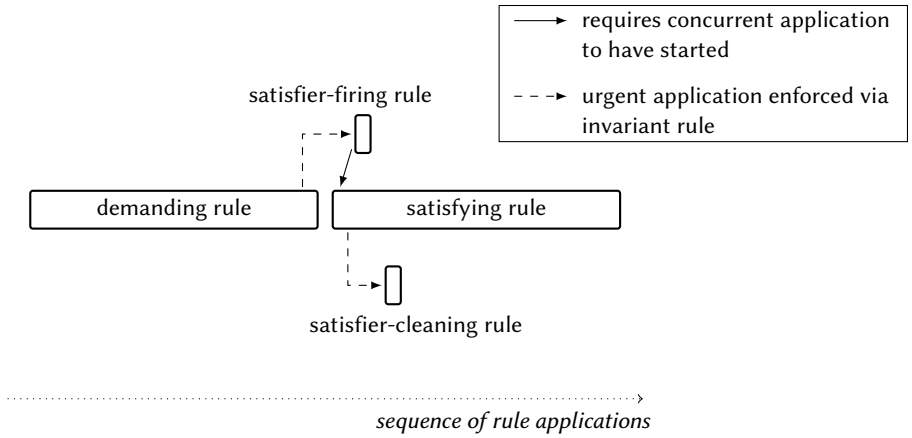


Figure 5.26: Urgent execution of a satisfying rule after a demanding rule

i.e., the demand indicator is deleted again, within the time frame specified by the urgency rule, we use an invariant rule over the demand indicator. The only rule able to delete this demand indicator is the timed rule shown above the satisfying rule in Figure 5.26. This rule is called *satisfier-firing timed rule*. Its application is enforced via the invariant rule. The purpose of the satisfier-firing timed rule is to enforce the application of a satisfying rule. To do so, the rule requires a satisfaction indicator to exist in the host graph. Since the application of the satisfier-firing timed rule is itself enforced via an invariant rule, a compatible satisfaction indicator has to be created before its application. This is what causes a satisfying rule to be applied.

Note that a satisfying rule can also be applied when there is no demand in urgent execution. In such a case, the satisfying rule creates a satisfaction indicator that is not deleted by a satisfier-firing timed rule. If left behind in the host graph, a satisfaction indicator might cause a problem when an urgency demanding rule is applied a second time: since the old satisfaction indicator is still available, there is no need for a satisfying rule to be applied. Therefore, satisfaction indicators are deleted by another timed rule, called *satisfier-cleaning timed rule*. This rule, shown below the satisfying rule in Figure 5.26, has to be applied for every satisfaction indicator in the host graph that is not deleted by an application of the satisfier-firing timed rule. As with the satisfier-firing timed rule, we enforce the application of the satisfier-cleaning timed rule via an invariant rule.

The demand and satisfaction indicator can simply be attached to the RHS of the demanding rule and the LHS of the satisfying rule, respectively. Their proper relative positioning in a configuration, i.e., the matching constraints formalized via the demander and satisfier constraint morphisms, is guaranteed by the satisfier-firing timed rule. There is no need to extend the demanding rule with additional nodes and edges as done in the case of concurrency rules. Extending the demanding rule was necessary for concurrency rules because concurrency rules do not have demand indicators.

The induced TGTS type graph is extended by urgency rules to support their demand and satisfaction indicators. For each demand and satisfaction indicator, it includes a separate type. For an urgency rule with the name *name*, its demand and satisfaction indicator type are given by $diType(name)$ and $siType(name)$, respectively. Furthermore, there are distinct demand and satisfaction indicator edge types for each node in the interface subgraphs of the urgency rule. For a node v, its demand and satisfaction indicator edge type are given by $diEdgeType(v)$ and $siEdgeType(v)$, respectively.

Next, we give the formal definitions for the semantics of urgency rules. Between those definitions, we give examples of the extended induced rules and the timed rules directly induced by urgency rules. They follow the example of urgency rule `immediatelyMoveRailCab` with `moveRailCab` as demanding rule and `brakeRailCab` as satisfying rule.

First, we extend the induced end rule of an urgency demanding rule such that it creates a demand indicator in the host graph.

Definition 5.6.2 (Extension of an urgency demander's induced end rule)**.** Let $\mathcal{U} = (G^T, D^T, S^T, D, S, name, dl)$ be an urgency rule and $\mathcal{DR}$ a set of durative rules. For each urgency demander tuple $(\mathcal{D}, d) \in D$, the induced end rule $er = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $er' = (L, R', r, \mathcal{N}, z, V_{res})$ where

- $V_{DI} = \{di\} \wedge type(di) = diType(name) \wedge$
  $E_{DI} = \{e | src(e) = di \wedge tgt(e) \in \text{ran}(d) \wedge$
  $type(e) = diEdgeType \circ d^{-1} \circ tgt(e)\}$,
- $V_{G,R'} = V_{G,R} \cup V_{DI} \wedge E_{G,R'} = E_{G,R} \cup E_{DI}$, and
- $V_{CI,R'} = V_{CI,R} \wedge E_{CI,R'} = E_{CI,R}$.

Figure 5.27 shows the extended induced end rule of `moveRailCab`. This extension is done according to the demander constraint morphism of Figure 5.24(a). As with the last section, the extended rule does not show any NACs or locks that have been

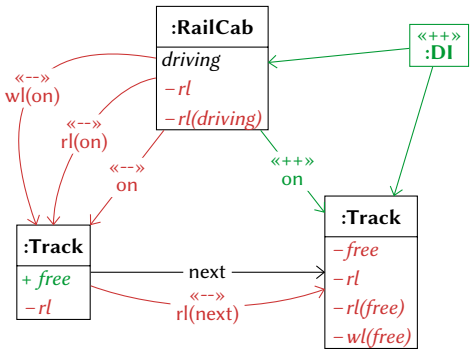


Figure 5.27: Demanding rule `moveRailCab`'s induced end rule extended according to urgency rule `immediatelyMoveRailCab`

generated to support NACs on the level of durative rules. The only new element is a demand indicator, which is connected to the `RailCab` node and the right `Track` node.

To enforce the application of the satisfying rule, the urgency rule induces a satisfier-firing timed rule.

Definition 5.6.3 (Induced satisfier-firing timed rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name, dl)$, the *induced satisfier-firing timed rule* of $\mathcal{U}$ is a timed rule $sftr = (L, R, r, \mathcal{N}, z, V_{res})$ where

- $V_{DI} = \{di\} \wedge type(di) = diType(name) \wedge$
  $E_{DI} = \{e | src(e) = di \wedge tgt(e) \in V_{D^T} \wedge type(e) = diEdgeType \circ tgt(e)\}$,
- $V_{SI} = \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{SI} = \{e | src(e) = si \wedge tgt(e) \in V_{S^T} \wedge type(e) = siEdgeType \circ tgt(e)\}$,
- $V_{G,L} = V_{G^T} \cup V_{DI} \cup V_{SI} \wedge E_{G,L} = E_{G^T} \cup E_{DI} \cup E_{SI}$,
- $V_{G,R} = V_{G^T} \wedge E_{G,R} = E_{G^T}$,
- $V_{CI,L} = V_{CI,R} = \emptyset \wedge E_{CI,L} = E_{CI,R} = \emptyset$,
- $r : L \to R$, with $\mathrm{dom}(r) = R$, is the identity morphism on R,
- $\mathcal{N} = \emptyset$, and
- $z = \emptyset \wedge V_{res} = \emptyset$.

The placement of the demand and satisfaction indicator in a satisfier-firing timed rule is determined by the demander and satisfier interface of its inducing urgency rule. This ensures that the satisfying rule is applied at a compatible match, i.e., a match that is compatible with the structure specified in the urgency rule.

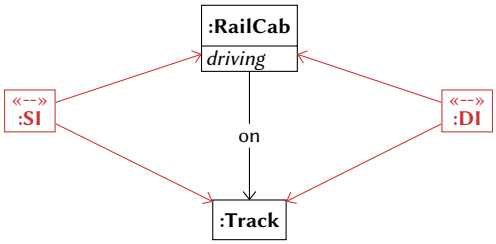


Figure 5.28: Induced satisfier-firing timed rule of urgency rule `immediatelyMoveRailCab`

Figure 5.28 shows the satisfier-firing timed rule that has been directly induced by `immediatelyMoveRailCab`. It consists of the connecting graph of `immediatelyMoveRailCab` with an additional demand and satisfaction indicator in its LHS. Here, both indicators are connected to both nodes because both interface subgraphs are identical to the connecting graph.

The application of a satisfier-firing timed rule is coupled to the application of a demanding rule's induced end rule via a satisfier-firing invariant rule. Note that the interplay between a satisfier-firing timed rule and a satisfier-firing invariant rule is analogous to that of a durative rule's induced end rule and invariant rule. However, instead of a duration given by a durative rule, this invariant rule specifies a deadline for firing the satisfier-firing timed rule according to the deadline given in the urgency rule.

Definition 5.6.4 (Induced satisfier-firing invariant rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name, dl)$, the *induced satisfier-firing invariant rule* of $\mathcal{U}$ is an invariant rule $sfir = (L, z)$ where

- $V_{G,L} = \{di\} \wedge type(di) = diType(name) \wedge E_{G,L} = \emptyset$,
- $V_{CI,L} = \{ci\} \wedge E_{CI,L} = \{(ci, di)\}$, and
- $z = \{ci \leq dl\}$.

Both a satisfier-firing timed rule and a satisfier-firing invariant rule need a clock instance to operate on. Such a clock instance is created by a satisfier-firing clock instance rule.

Definition 5.6.5 (Induced satisfier-firing clock instance rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name, dl)$, the *induced satisfier-firing clock instance rule* of $\mathcal{U}$ is a clock instance rule $sfcr = (L, R, r, \mathcal{N})$ where

- $V_{G,L} = V_{G,R} = \{di\} \wedge type(di) = diType(name) \wedge E_{G,L} = E_{G,R} = \emptyset$ and
- $V_{CI,L} = \emptyset \wedge E_{CI,L} = \emptyset \wedge V_{CI,R} = \{ci\} \wedge E_{CI,R} = \{(ci, di)\}$.

Now, we extend the induced start rule of an urgency satisfying rule such that it creates a satisfaction indicator in the host graph. This extension is done analogously to the extension of the urgency demanding rule's end rule.

Definition 5.6.6 (Extension of an urgency satisfier's induced start rule)**.** Let $\mathcal{U} = (G^T, D^T, S^T, D, S, name)$ be an urgency rule and $\mathcal{DR}$ a set of durative rules. For each urgency satisfier tuple $(\mathcal{D}, s) \in S$, the induced start rule $sr = (L, R, r, \mathcal{N}, z, V_{res})$ of $\mathcal{D} \in \mathcal{DR}$ is extended into a timed rule $sr' = (L, R', r, \mathcal{N}, z, V_{res})$ where

- $V_{SI} = \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{SI} = \{e | src(e) = si \wedge tgt(e) \in \text{ran}(s) \wedge$
  $type(e) = siEdgeType \circ s^{-1} \circ tgt(e)\}$,
- $V_{G,R'} = V_{G,R} \cup V_{SI} \wedge E_{G,R'} = E_{G,R} \cup E_{SI}$, and
- $V_{CI,R'} = V_{CI,R} \wedge E_{CI,R'} = E_{CI,R}$.

Figure 5.29 shows the extended induced start rule of (urgency) satisfying rule `brakeRailCab`. The extension is done according to the satisfier constraint morphism of Figure 5.24(c). It is analogous to the extension of the induced end rule of the

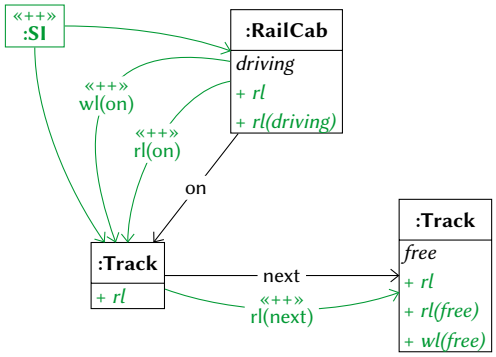


Figure 5.29: Satisfying rule `brakeRailCab`'s induced start rule extended according to urgency rule `immediatelyMoveRailCab`

demanding rule. The only new element is a satisfaction indicator, which is connected to the `RailCab` node and the left `Track` node.

If the induced start rule of a satisfying rule is applied in a situation where there was no demand in urgent execution, it creates a satisfaction indicator that is not needed and thus not consumed by a satisfier-firing timed rule. To prevent such a satisfaction indicator from remaining in the system until an unrelated satisfier-firing timed rule is applied, we delete it immediately. This is done by a satisfier-cleaning timed rule.

Definition 5.6.7 (Induced satisfier-cleaning timed rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name)$, the *induced satisfier-cleaning timed rule* of $\mathcal{U}$ is a timed rule $sctr = (L, R, r, \mathcal{N}, z, V_{res})$ where

- $V_{SI} = \{si\} \wedge type(si) = siType(name) \wedge$
  $E_{SI} = \{e | src(e) = si \wedge tgt(e) \in V_{S^T} \wedge type(e) = siEdgeType \circ tgt(e)\}$,
- $V_{G,L} = V_{S^T} \cup V_{SI} \wedge E_{G,L} = E_{S^T} \cup E_{SI}$,
- $V_{G,R} = V_{S^T} \wedge E_{G,R} = E_{S^T}$,
- $V_{CI,L} = V_{CI,R} = \emptyset \wedge E_{CI,L} = E_{CI,R} = \emptyset$,
- $r : L \to R$, with $\mathrm{dom}(r) = R$, is the identity morphism on R,
- $\mathcal{N} = \emptyset$, and
- $z = \emptyset \wedge V_{res} = \emptyset$.

Figure 5.30 shows the satisfier-cleaning timed rule that has been directly induced by `immediatelyMoveRailCab`. While this rule looks similar to the satisfier-firing timed rule, except for the missing demand indicator, this does not have to be the case for an arbitrary urgency rule. Instead of the complete connecting graph, satisfier-cleaning timed rules only use the subgraph constituting the satisfier interface. The demander interface is irrelevant for satisfier-cleaning timed rules, because there was no application of a demanding rule when a satisfier-cleaning timed rule is applied.

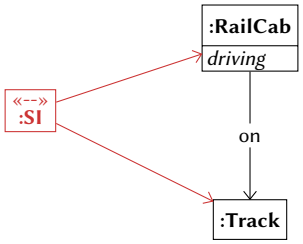


Figure 5.30: Induced satisfier-cleaning timed rule of urgency rule `immediatelyMoveRailCab`

The application of a satisfier-cleaning timed rule is coupled to the application of a satisfying rule's induced start rule via a satisfier-cleaning invariant rule.

Definition 5.6.8 (Induced satisfier-cleaning invariant rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name)$, the *induced satisfier-cleaning invariant rule* of $\mathcal{U}$ is an invariant rule $scir = (L, z)$ where

- $V_{G,L} = \{si\} \wedge type(si) = siType(name) \wedge E_{G,L} = \emptyset$,
- $V_{CI,L} = \{ci\} \wedge E_{CI,L} = \{(ci, si)\}$, and
- $z = \{ci = 0\}$.

Both a satisfier-cleaning timed rule and a satisfier-cleaning invariant rule operate on a clock instance that is created by a satisfier-cleaning clock instance rule.

Definition 5.6.9 (Induced satisfier-cleaning clock instance rule)**.** Given an urgency rule $\mathcal{U} = (G^T, D^T, S^T, D, S, name)$, the *induced satisfier-cleaning clock instance rule* of $\mathcal{U}$ is a clock instance rule $sccr = (L, R, r, \mathcal{N})$ where

- $V_{G,L} = V_{G,R} = \{si\} \wedge type(si) = siType(name) \wedge E_{G,L} = E_{G,R} = \emptyset$ and
- $V_{CI,L} = \emptyset \wedge E_{CI,L} = \emptyset \wedge V_{CI,R} = \{ci\} \wedge E_{CI,R} = \{(ci, si)\}$.

As with a durative graph transformation system without urgency rules, the semantics of a durative graph transformation system with urgency rules is given by its induced timed graph transformation system. Since Definitions 5.2.18 and 5.5.6 do not regard urgency rules, we have to provide a new definition for a durative graph transformation system with urgency rules.

Definition 5.6.10 (Induced timed graph transformation system respecting urgency rules)**.** Let $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR})$ be a durative graph transformation system that contains a set of concurrency rules $\mathcal{CR}$ and a set of urgency rules $\mathcal{UR}$ and $\mathcal{TS} = (TG, TiG_0, TR, IR, CR)$ its induced timed graph transformation system respecting concurrency rules according to Definition 5.5.6. Its *induced timed graph transformation system respecting urgency rules* $\mathcal{TS}' = (TG', TiG_0, TR', IR', CR')$ differs from $\mathcal{TS}$ in that

- the induced TGTS type graph TG has been extended into a type graph TG' that contains a demand indicator type and a satisfaction indicator type for each urgency rule in $\mathcal{UR}$ as well as their demand indicator edge types and satisfaction indicator edge types,

- each timed rule $tr \in TR$ whose inducing durative rule $\mathcal{D}$ is referenced by an (urgency) demander or (urgency) satisfier tuple of an urgency rule in $\mathcal{UR}$ has been extended into a timed rule $tr' \in TR'$ as defined in Definitions 5.6.2 and 5.6.6, and if $\mathcal{D}$ is referenced by multiple (urgency) demander or (urgency) satisfier tuples (of one or more urgency rules in $\mathcal{UR}$), then the timed rule is extended successively, and

- in addition to the extended variants of those timed rules in TR, the invariant rules in IR, and the clock instance rules in CR, the induced timed graph transformation system $\mathcal{TS}'$ also contains those timed rules, invariant rules, and clock instance rules that have been induced by $\mathcal{UR}$ according to Definitions 5.6.3 to 5.6.5 and 5.6.7 to 5.6.9.

Note that if there is no compatible satisfying rule that can be applied within the urgency rule's deadline after the demanding rule's application (and there is no timed rule making a satisfying rule applicable without passing more time than allowed), a time-stopping deadlock occurs. During operation of the system, this is not a problem per se, because the system does not have to take a path of the state space that leads into a time-stopping deadlock. After all, it is the task of the system's planning component to find a path leading to a certain goal specification, and if such a path exists, it it obviously free of deadlocks.

5.7 Related Work

Gyapay et al. [GHV02] proposed an approach to *graph transformation with time* that annotates codes with timestamps, called *chronos* values. Such chronos values can be read and written upon application of a graph transformation rule. When this is done, all written chronos values are set to the same time, i.e., the firing time of the graph transformation, which has to be higher than all chronos values read. Whether or not a node has a chronos value is defined via the type graph, i.e., either all nodes of a certain type have a chronos value or none of them. By assigning chronos values (via the RHS) relatively to their values read (via the LHS), a graph transformation rule can be seen as having some sort of firing duration, although its application is atomic.

There are vital differences between graph transformations with time and durative graph transformations. If there are types without chronos values or chronos values of some nodes in the RHS are not updated by a rule application, then there can be nonsensical sequences of graph transformations, where firing times are not monotonically increasing. If not, then no concurrent application of two rules is possible if the matches of both rules overlap. The reason for this is that each rule

application increases the chronos values of nodes in its match by its firing duration. This is clearly more restrictive than the DGTS formalism. Since graph transformation with time also do not have any concepts analogous to time guard and invariant rules, they would not have been a suitable alternative to TGTS for implementing the various concepts of the DGTS formalism.

Syriani and Vangheluwe [SV11; SV08] developed a modular language for timed graph transformation, called *MoTif*. Its semantics is based on *Discrete EVent system Specification (DEVS)* [Zei84], where graph are embedded in events being transmitted between scheduling units, called *atomic DEVS models*, which run in parallel. In MoTif, atomic DEVS models are transformation entities, which can have different execution semantics, e.g., applying a rule once, at all matches, or as many times as possible. These transformation entities have a so-called *time advance* specifying a delay after which the rule application occurs, i.e., graph transformation rules are applied instantaneously.

As with most related approaches, MoTif is more similar to timed graph transformation systems than to durative graph transformation systems. However, a downside compared to the TGTS formalism is that it is not possible to maintain a single consistent representation of the host graph when rules are being applied concurrently, because each transformation entity works on its own copies transmitted via events. As a consequence, concurrency issues are difficult to handle.

In the approach of de Lara et al. [Lar+14; Lar+10], graph transformation rules can schedule the application of other graph transformation rules at a later point in time. The approach is based on *discrete event simulation*, i.e., the application of a graph transformation rule is considered an *event*, and the scheduling of events is defined in a structure similar to an *event graph*. This structure contains so-called *invocation edges* and *canceling edges*. An invocation edge schedules future rule applications of its target rule when its source rule has been applied. Canceling edges discard scheduled rule applications of their target rules. A scheduled rule application is also discarded when its match is invalidated by another rule application. As a result of this, there is no guarantee that a scheduled rule application will be executed.

On the plus side, scheduled rule applications may also include matching constraints. Formally, these matching constraints are defined similar to constraint morphisms in the DGTS formalism. However, there is no connecting graph with individual interface subgraphs. As a consequence, matching constraints can only be formalized via elements existing in both rules.

Boronat and Ölveczky [BÖ10] presented *MOMENT2*, a model transformation framework supporting timed behavior. It is based on *Maude* [Cla+07], which is a specification language and tool based on rewriting logic and capable of verifying invariants and LTL properties. MOMENT2 introduces several timed constructs: a *clock*, which increases its value according to the elapsed time, a *timed value*, which is a clock with a (positive or negative) weighting factor, and a *timer*, which is a clock running backwards. Timers can be deactivated or reset by graph transformation rules. If a timer reaches zero, time is not allowed to pass anymore, i.e., a graph

transformation rule has to be applied before the passing of time may continue. The purpose of timers is thus similar to that of invariant rules in the TGTS formalism.

The approach of Rivera et al. [RDV10] is similar to MOMENT2 in that it extends in-place model transformations with timed behavior. In their tool *e-Motions*, durations of graph transformation rules are specified as intervals representing the minimum and maximum amount of time needed to execute the graph transformation. Its semantics is given by a mapping to *Real-Time Maude* [ÖM07]. Similar to a durative graph transformation rule in DGTS, a rule in e-Motions is compiled into two rewrite rules, the so-called *triggering* and *realization rule*. The use of timers ensures that the amount of time consumed between executing these two rewrite rules satisfies the duration interval. As opposed to MOMENT2, this approach is more high-level because timers do not have to be managed manually.

Checking the rule's applicability is implemented both in the triggering and realization rule. Additional invariant checks are optional, cf. [RVV09]. The rule's execution is implemented in the realization rule. If the LHS match does not exist anymore when the realization rule is scheduled to be applied, its execution is being canceled. This might lead to erroneous behavior if the execution of a concurrent graph transformation relied on this rule. In the DGTS formalism, such a cancellation of durative rules is prevented by the use of a locking mechanism.

A distinguishing feature of e-Motions is the possibility to refer to past and concurrent rule applications, called *action executions*, in graph transformation rules. As there is no equivalent to a locking mechanism, this feature has to be used by a designer to prevent conflicting graph transformations from being executed concurrently. Using this feature to require a concurrent rule application shares similarities with concurrency rules in the DGTS formalism. However, it is less expressive for three reasons:

1. While the match of a required concurrent rule application can be restricted to contain certain nodes, it is not possible to specify the position of these nodes in the concurrent rule's LHS. As a result, the concurrency rule `allowChangePublication` cannot be specified correctly in e-Motions, because its satisfying rule `formConvoy` has multiple nodes of the `RailCab` type but only one of them satisfies the demand in concurrent execution.

2. Matching constraints can only be formalized for nodes appearing in both rules. In the DGTS formalism, matching constraints are expressed via the structure of the connecting graph, which enables to relate `Base` nodes in `changePublication` to `Track` nodes in `moveRailCab` although `moveRailCab` has no `Base` nodes and `changePublication` has no `Track` nodes.

3. Unlike concurrency rules in the DGTS formalism, it is not possible to specify a disjunction of satisfying concurrent rule applications in e-Motions.

Baldan et al. [Bal+08] provide a theoretical framework for the definition of *transactional graph transformation systems*. A transactional graph transformation systems differentiates between graph elements that are *stable* and *unstable*. The stable

part of a graph or state is that portion that is visible to an external observer, the unstable part is hidden. A *transaction* is then a sequence of graph transformations starting and ending in completely stable states but traversing through states with unstable elements. If two transactions are parallel independent, then the graph transformations they contain can be interleaved arbitrarily. Unfortunately, this approach requires the non-existence of NACs. If NACs exist, there is no guarantee that these graph transformations can be interleaved arbitrarily.

Comparing this to the DGTS formalism, locking edges and application indicators can be seen as unstable elements and durative graph transformations as transactions. An important difference is that transactional graph transformation systems do not provide a notion of time, which is essential for durative graph transformation systems. Due to the non-existence of NACs in transactional graph transformation systems, they are also not suitable as semantic basis for the locking mechanism of the DGTS formalism. In essence, transactional graph transformation systems are more similar to refinement or module concepts of graph transformation systems, see [Hec+99] for an overview of such approaches, than to durative or timed graph transformation systems.

Corradini et al. [CFR09] employed transactional graph transformation systems to implement *graph transformations with dependencies*. In a graph transformation system with dependencies, graph transformation rules are supplemented with so-called *dependency relations*, which express additional relationships between the deleted and created elements of a graph transformation. The specification of these dependencies itself underlies certain conditions, which guarantee that a suitable sequence of graph transformations implementing the graph transformation with dependencies exists (and can be constructed automatically). An application area for graph transformations with dependencies are reactive systems: graph transformations with dependencies provide a means to abstractly specify interaction patterns between the reactive system and its environment. While dependency relations specify restrictions affecting the refinement of a single graph transformation, a durative graph transformation brings along restrictions regarding the execution of multiple concurrent graph transformations, i.e., it implies forbidden as well as mandatory interactions.

5.8 Discussion

The design of durative graph transformation rules has mostly been motivated by the wish for a graph transformation approach with time that feels natural and intuitive to a modeler. Therefore, the application of a durative graph transformation rule was defined such that it mostly corresponds to that of an untimed rule, with the only difference that it consumes time and allows the concurrent application of other durative rules. For the same reason, no interleaving of start and end times of durative graph transformations stops an ongoing durative graph transformation from terminating, and every interleaving results in the same graph when each involved rule has terminated. These properties of durative rules have already been addressed and proven in Sections 5.3.1 and 5.3.2.

In this section, we want to discuss design decisions regarding the syntax and semantics of concurrency and urgency rules. This section mainly addresses three things:

1. syntactic sugar added into the syntax of concurrency and urgency rules,
2. why the semantics of concurrency and urgency rules does not restrict the application of durative rules any more than necessary, and
3. why the semantics of concurrency and urgency rules has been designed differently although their syntax is almost identical.

Concurrency and urgency rules both have a non-empty set of demander tuples as well as a non-empty set of satisfier tuples. In the case of demander tuples, having only a single tuple would have been sufficient. The option of defining more than one tuple in the set of demander tuples can be seen as syntactic sugar. For the set of satisfier tuples, the option of having more than one tuple in the set is necessary for the expressiveness of the rules.

The necessity for having more than one tuple in the set of satisfier tuples can be seen easily. The application of a demanding rule creates a demand that can be satisfied by an application of a satisfying rule. In cases where we have multiple options to satisfy this demand, e.g., multiple durative rules moving a RailCab and thus satisfying the demand of the concurrency rule `allowChangePublication`, we need the possibility of having multiple tuples in the set of satisfier tuples.

To see that the option of defining more than one tuple in the set of demander tuples is only syntactic sugar, consider the urgency rule `immediatelyMoveRailCab`. Each durative rule that moves a RailCab and does not come to a halt is referenced by `immediatelyMoveRailCab` as a demanding rule. Besides `moveRailCab`, this can be `accelerateRailCab` or durative rules related to convoy membership change, e.g., `leaveConvoy`. However, instead of referencing every demanding rule from the same urgency rule, we can flatten each reference to a demanding rule into its own urgency rule. These flattened urgency rules each have the same connecting graph and the same set of satisfier tuples. The difference in semantics between such a set of flattened rules and an integrated rule is only a technical one. Each urgency rule in the set of flattened rules gives rise to its own demand and satisfaction indicator type. As a consequence, each demanding rule works with a different demand indicator and each satisfying rule delivers multiple satisfaction indicators, i.e., one for each flattened rule. However, as far as the application of durative rules is concerned, this makes no difference. Whether each demanding rule creates the same or a distinct demand does not matter, because the demand can be satisfied by the same satisfying rules in both cases. For all practical matters, the meaning of a set of flattened urgency rules is the same as that of the integrated urgency rule.

Next, we give reasons as to why the semantics of concurrency and urgency rules does not restrict the application of durative rules any more than necessary. A certain restriction of a demanding rule's applicability is of course intended: a demanding rule whose demand is not satisfied is not applicable. However, if a demand can

be satisfied, i.e., a satisfying rule can be applied with a compatible match, the concurrent or urgent execution of both rules is possible unless it is prevented by locking conflicts. For satisfying rules, there is no such restriction: if a durative rule is applicable at a match, it is also applicable at the same match if referenced as a satisfying rule by a concurrency or urgency rule.

No unnecessary restrictions from concurrency rules In the case of concurrency rules, the induced start and end rule of a demanding rule are extended via a pushout construction. The elements added by this pushout cause matches of the demanding rule's induced rules to be less likely. However, this is not unjustified and does not restrict the applicability of demanding rules any more than necessary: the elements added to the induced rules have to exist in the host graph anyway because every compatible satisfying rule requires them.

The applicability of a satisfying rule's induced end rule is only restricted by read locks on its satisfaction indicator, which are acquired and released by demanding rules. Therefore, it is not restricted by any concurrency rule if no application of a demanding rule is involved.

Note that it is technically possible to specify a concurrency rule involving incompatible rules, i.e., the demand can never be satisfied due to locking conflicts. While such a specification is syntactically correct, it is of course not sensible. Fortunately, such unreasonable specifications can easily be found via a critical pair analysis, i.e., by generating all overlapping graphs of the LHSs of a demanding and a satisfying rule and checking whether there is a conflict if both rules are applied, cf. [Plu93; LE08; LEO06]. If there is a conflict for each overlapping graph, the specification of the concurrency rule is at fault.

No unnecessary restrictions from urgency rules The case of urgency rules is more simple. Here, induced rules are extended only in that they create demand and satisfaction indicators, which are consumed again by satisfier-firing and satisfier-cleaning timed rules. The demand for a satisfying rule to be applied is caused indirectly via a satisfier-firing timed rule. In doing so, the applicability of the demanding rule itself is technically not restricted by the urgency rule. Furthermore, the satisfier-firing timed rule can always be applied if a compatible satisfying rule has been applied.

The different approaches used in the semantics of concurrency and urgency rules leads to the question as to why they have been realized so differently. Recall that the semantics of concurrency rules uses a satisfaction indicator and a pushout to allow for a proper placement of the satisfaction indicator in induced rules of demanding rules, while the semantics of urgency rules simply uses a demand indicator in addition to the satisfaction indicator. Implementing the semantics of concurrency rules with a demand indicator instead of the pushout construction or the semantics of urgency rules via a pushout construction instead of one of the indicators would have had disadvantages compared to the semantics realized.

Argument against demand indicators in concurrency rules The main idea of using a pushout construction in the semantics of concurrency rules is that we can require a satisfaction indicator at the correct place in the demanding rules' induced rules. Without the pushout, we would not have all elements needed to attach the satisfaction indicator properly. This is why we need to have a second indicator, in this case for the demanding rule, when no pushout is used.

Now, if we used a demand indicator for concurrency rules, we would need another timed rule to check whether the demanding and satisfying rule's transformations are compatible. This is what the satisfier-firing timed rule does in the case of urgency rules. However, introducing such a rule into the semantics of concurrency rules and ensuring its application via an invariant rule leads to more states in the state space, i.e., in each path where a concurrency rule has been effective, there is a state before applying the concurrency-rule equivalent of a satisfier-firing timed rule and a state afterwards. This is obviously a drawback compared to the semantics realized.

Argument against a pushout construction in urgency rules In the case of concurrency rules, the pushout construction is used to allow a proper attachment of a satisfaction indicator to elements of the induced rules of a demanding rule. This avoids the need for a demand indicator and thus the need for another timed rule whose application ensures that the demanding and satisfying rule's transformations are compatible. Their compatibility is given directly by the placement of the satisfaction indicators.

There are two options to try when adapting such a use of a pushout to the semantics of urgency rules: either we use a pushout in the semantics of demanding rules to enable the attachment of satisfaction indicators, or we use a pushout in the semantics of satisfying rules to enable the attachment of demand indicators.

Using a pushout in the semantics of demanding rules, i.e., for extending the induced end rule of a demanding rule, would mean trying to avoid the need for a demand indicator by directly attaching the satisfaction indicator at a place ensuring compatibility of the demanding and satisfying rule's transformations. However, if the satisfying rule creates the satisfaction indicator, and the demanding rule checks whether its demand is satisfied by requiring the existence of a satisfaction indicator, this would mean that the satisfying rule has to start before the demanding rule ends. This would impose multiple problems. First, such a semantics would not support deadlines and thus no applications of other durative rules during the interval between the application of the demanding rule ends and that of the satisfying rule starts. Second, and this is the more important problem, such a semantics would not support transformations where the application of the satisfying rule sequentially depends on the demanding rule's application.

The troublesome ordering of the demanding rule's induced end rule and satisfying rule's induced start rule could potentially be fixed by letting the demanding rule create the satisfaction indicator and the satisfying rule consume it. In doing so, this "satisfaction" indicator would essentially indicate a demand, like a demand indicator, but be placed like a satisfaction indicator, i.e., the pushout is still used in the demanding rule's semantics. However, this would fix the problems mentioned above only in parts. Due to the pushout, the satisfier interface subgraph would still have to be available in the configuration when the application of the demanding rule ends. This would be bad because it would still restrict the applicability of the demanding rule's induced end rule more than necessary. With the realized semantics, the satisfier interface subgraph does not have to be available when the application of the demanding rule ends but can be established afterwards, i.e., after the application of the demanding rule ended but before that of the satisfying rule starts, or by an independent parallel transformation.

The second option, i.e., using a pushout in the semantics of satisfying rules, would mean trying to avoid the need for a satisfaction indicator by directly attaching the demand indicator in the induced start rule of a satisfying rule at a place ensuring compatibility of the demanding and satisfying rule's transformations. In doing so, the elements added by the pushout to the satisfying rule's induces start rule obviously cause matches of this rule to be less likely. While unproblematic for concurrency demanding rules, such a restriction is not appropriate for urgency satisfying rules: each satisfying rule should still be applicable independently of any demanding rule, and its application should not be restricted by additional elements in the LHS of its induced start rule.

Finally, we conclude with an observation on the interface subgraphs of a connecting graph. Interface subgraphs can also be seen as interfaces in the sense of abstract durative rules, i.e., durative rules that cannot be applied but define common parts of concrete durative rules. Therefore, specifying the demanding and satisfying rules of a concurrency or urgency rule is similar to rule inheritance approaches, which have already been established for model-to-model transformation languages, cf. [Wim+11]. In this sense, demander and satisfier constraint morphisms can be seen as declarations of interface implementations. As an alternative to the interface-centric view for these declarations, i.e., the compact representations for demander and satisfier constraint morphisms of Figures 5.16 and 5.25, they can also be shown from the perspective of a durative rule. Referencing the nodes of the connecting graph in such a view can be done by employing object names in connecting graphs, analogously to the compact representation of concurrency and urgency rules.

6 Temporal PDDL-Based Planning for Durative Graph Transformation Systems

This chapter presents an approach for planning software architecture reconfiguration with time and concurrency. A key component of this approach is a translation scheme that builds a propositional planning domain representation in PDDL from a durative graph transformation system. This PDDL representation allows for making use of off-the-shelf temporal planning systems, which yield plans that exactly state when and how long reconfigurations are going to be executed. By employing durative graph transformations and the translation scheme presented in this chapter, temporal planning techniques can be integrated with component-based software development approaches to architectural reconfiguration.

Most current planning approaches to architectural reconfiguration do not support time, and consequently only generate non-temporal plans. To the best of our knowledge, the only planning approach that does support time (besides ours) was developed by Tichy and Klöpper [TK11] as part of the CRC 614. Like our approach, their approach builds on graph transformation rules for modeling reconfiguration behavior. They assign each rule a duration and temporal annotations, like «`at_start`» and «`at_end`», as stereotypes to the elements of rules to specify when conditions have to hold and when effects are carried out. However, this is cumbersome from a modeling perspective for two reasons:

1. The domain modeler has to exercise caution not to enable potentially dangerous conflicts, e.g., the deinstantiation and use of a software component at the same time. In safety-critical environments, like real-time mechatronic systems, such conflicts might result in system crashes or the loss of lives.

2. Requirements regarding the concurrent or urgent execution of reconfigurations cannot easily be expressed. Instead of explicitly expressing such requirements in language constructs dedicated for this purpose, such requirements indirectly result from the interplay of multiple graph transformation rules.

Our approach to generate temporal reconfiguration plans can be seen as an exten-

sion of [TK11]. We solve planning tasks by translating durative graph transformation systems into PDDL – more precisely PDDL 2.1 level 3 [FL03], which supports discrete durative actions – and feeding them into an off-the-shelf planning system, like SGPlan [CWH06] or POPF [Col+10]. However, our solution renounces from assigning start and end annotations. Instead, we adapt the locking mechanism implemented into the semantics of durative graph transformation rules, cf. [ZW15; ZW13], and the concepts of concurrency and urgency rules to PDDL's propositional representation. A planning domain model originating from a durative graph transformation system thus ensures that plans do not contain any conflicting transformations, but comply with the requirements formalized as concurrency and urgency rules.

Together with the language provided in the last chapter, i.e., the DGTS formalism, the translation scheme constitutes a convenient method for specifying temporal planning domains, for which planning tasks can be solved by state-of-the-art planning systems. Apart from the software engineering perspective, this translation scheme can also be seen as a knowledge engineering contribution to the AI planning research community. While knowledge engineering approaches usually cover multiple phases of the design cycle of a planning application, e.g., domain specification, analysis, and plan visualization, this approach concerns only the initial phase, i.e., the construction of a planning domain model. Its strength lies in its high level of abstraction, which relieves a designer of explicitly addressing undesired concurrency, and its explicit representation of required concurrency and urgency.

The translation scheme provides different options for the translation of certain features of a durative graph transformation system. Since the choice of translation variant has a huge impact on the computation time needed by the employed planning system, we also provide an evaluation comparing their performance.

The deletion of nodes can follow the

- DPO approach, where a graph transformation cannot be applied if it results in a dangling edge, or
- SPO approach, where dangling edges are deleted.

Both approaches have been discussed in Chapter 2. The choice between these two options depends on the application domain and its intended semantics.

Furthermore, there are two different variants for the translation of negative application conditions. We can

- translate each NAC into a negative existential quantification or
- employ functions counting the number of adjacent edges for each node, which enables to translate each NAC into a simple numeric fact.

Unfortunately, the technique employing counting functions, which proved to be more efficient, is restricted to specific kinds of NACs, i.e., forbidden edges and forbidden pairs.

The next section introduces notions of temporal planning problems and temporal plans that are appropriate for durative graph transformation systems. Section 6.2

presents a variant of the RailCab system as application example, which is simpler than that in the last chapter. The application example differs in that it does not model base stations or acceleration and braking of RailCabs and does not employ concurrency of urgency rules, but makes an extensive use of NACs as compensation. Using this variant of the RailCab system as running example, we explain most parts of our translation scheme in Section 6.3. A prototype implementing this translation scheme is shortly introduced in Section 6.4. Sections 6.5 and 6.6 provide an evaluation of planner performance on the generated PDDL domains. Section 6.5 covers the alternative translation variants mentioned above, and Section 6.6 checks for the feasibility of employing concurrency and urgency rules by evaluating their impact on planner performance. The sections on concurrency and urgency rules and their evaluation are based on the application example of the last chapter.

Related work in the area of graph transformation planning has already been discussed in Section 4.5. In Section 6.7, we cover related work that is specifically concerned with translations into PDDL, whether from graph transformation systems or other model-based or object-oriented representations. Eventually, we conclude this chapter with a discussion and suggestions for optimizing planner performance on our models in Section 6.8.

6.1 Problem Statement

The planning problem on a durative graph transformation system is similar to that on an untimed graph transformation system. Like the graph transformation planning problem, the temporal graph transformation planning problem uses a target graph pattern as a goal specification. The task is to find a subgraph isomorphism from this target graph pattern to a state that can be found from the initial state by applying durative graph transformation rules. In doing this, the application of durative graph transformation rules also has to respect concurrency and urgency rules.

Definition 6.1.1 (Temporal graph transformation planning problem)**.** A *temporal graph transformation planning problem* $\mathcal{TP} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR}, P_{tgt})$ consists of

- a type graph $\mathcal{TG}$,
- an initial timed graph G_0^T,
- a set of durative graph transformation rules $\mathcal{DR}$,
- a set of concurrency rules $\mathcal{CR}$,
- a set of urgency rules $\mathcal{UR}$, and
- a target graph pattern $P_{tgt} = (L_{tgt}, \mathcal{N}_{tgt})$.

In the non-temporal case, a plan is simply a sequence of graph transformations. However, since we are facing durative graph transformations, the application intervals of multiple graph transformations might overlap. A temporal plan is thus a set of tuples of points in time and durative graph transformations, i.e., a *schedule* of

durative graph transformations. This schedule of durative graph transformations respects the sets of concurrency and urgency rules given in the temporal graph transformation planning problem.

Definition 6.1.2 (Temporal graph transformation plan)**.** Given a temporal graph transformation planning problem $\mathcal{TP} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR}, P_{tgt})$ with a target graph pattern $P_{tgt} = (L_{tgt}, \mathcal{N}_{tgt})$, a *temporal graph transformation plan* for $\mathcal{TP}$ is a finite set of tuples $(t, \overset{\mathcal{D},m}{\Longrightarrow}) \in SKED$, where $t \in \mathbb{N}$ is a point in time on a global clock and $\overset{\mathcal{D},m}{\Longrightarrow}$ is a durative graph transformation, such that

- for each tuple $(t, \overset{\mathcal{D},m}{\Longrightarrow}) \in SKED$, the match m of durative rule $\mathcal{D}$ exists at its respective point in time t and satisfies the NACs $\mathcal{N}_\mathcal{D}$ of $\mathcal{D}$,
- for each tuple $(t, \overset{\mathcal{D},m}{\Longrightarrow}) \in SKED$ where $\mathcal{D}$ is referenced by concurrency rule $\mathcal{C}$ via demander constraint morphism $d : D^T \to L_\mathcal{D}$, there is another tuple $(t', \overset{\mathcal{D}',m'}{\Longrightarrow}) \in SKED$ where
  - $\mathcal{D}'$ is referenced by $\mathcal{C}$ via satisfier constraint morphism $s : S^T \to L_{\mathcal{D}'}$,
  - there exists a match $g : G^T \to G$ from the connecting graph of $\mathcal{C}$ to the host graph at time t such that $m \circ d(D^T) = g(D^T)$ and $m' \circ s(S^T) = g(S^T)$,
  - and for t' hold $t' \leq t$ and $t' + d_{\mathcal{D}'} \geq t + d_\mathcal{D}$.
- for each tuple $(t, \overset{\mathcal{D},m}{\Longrightarrow}) \in SKED$ where $\mathcal{D}$ is referenced by urgency rule $\mathcal{U}$ via demander constraint morphism $d : D^T \to R_\mathcal{D}$, there is another tuple $(t', \overset{\mathcal{D}',m'}{\Longrightarrow}) \in SKED$ where
  - $\mathcal{D}'$ is referenced by $\mathcal{U}$ via satisfier constraint morphism $s : S^T \to L_{\mathcal{D}'}$,
  - there exists a match $g : G^T \to G$ from the connecting graph of $\mathcal{C}$ to the host graph at time t' such that $m^* \circ d(D^T) = g(D^T)$ and $m' \circ s(S^T) = g(S^T)$,
  - and for t' hold $t' \geq t + d_\mathcal{D}$ and $t' \leq t + d_\mathcal{D} + dl_\mathcal{U}$.
- the application of all durative graph transformations from $SKED$ at their respective points in time results in a graph G_k^T such that L_{tgt} has an injective match m in G_k^T and m satisfies $\mathcal{N}_{tgt}$.

This definition explains what a temporal plan is from the perspective of a durative graph transformation system. From the perspective of its induced timed graph transformation system, a temporal plan is simply a sequence of transitions with certain properties. Recall that the semantics of durative graph transformation systems is given by action transitions and delay transitions. An action transition results from the application of a timed graph transformation rule. Even when there is no delay transition between two action transitions, the action transitions are applied in sequence and thus have an ordering. However, this ordering is not indicated by a temporal graph transformation plan as defined above.

The reason for this is that the ordering of transformations on the level of its induced timed graph transformation systems is not relevant from the perspective of

a durative graph transformation system. When multiple durative graph transformations start at the same time, the actual ordering of their induced start transformations does not matter, because they are parallel independent. The same holds for multiple end transformations. As a consequence, there is no need to indicate this ordering in a temporal graph transformation plan. The ordering of transformations occurring at the same point in time matters only in cases with a sequential dependency, which can only appear between end and start transformations. In all such cases, the end transformation is applied first.

A transition sequence of an induced timed graph transformations system that ends in a state satisfying the target graph pattern is not necessarily a valid plan. This is because there might be a durative graph transformation that did not finish its execution. In order to constitute a plan, the transition sequence also needs to finish each durative graph transformation with an end transformation. The definition given above meets this requirement because it considers each durative rule in its entirety.

Given a temporal graph transformation plan, the duration of the complete graph transformation process from the initial state to that satisfying the target graph pattern is given by the plan's *makespan*, i.e., the interval from the beginning of the first durative graph transformation to the end of the last one.

6.2 Application Example: RailCab System (Emphasis on NACs)

To explain most parts of our translation scheme, we again use the RailCab system as a running example. Unlike the RailCab domain used in the last chapter, the variant used here makes extensive use of NACs. While this is more expensive for the employed planning system, it avoids modeling many of the self edges needed in the last chapter. Furthermore, this domain does not include base stations or the acceleration and braking of RailCabs. Both would not contribute in understanding the translation scheme but simply increase the size of the resulting PDDL listings. As a result of this, the domain is rather simple. Figure 6.1 shows its type graph.

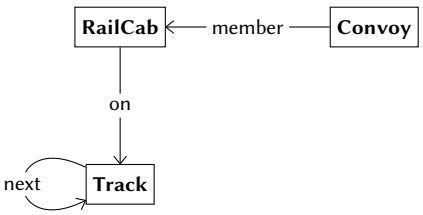


Figure 6.1: Type graph of the RailCab-NACs domain

Figure 6.2 provides all graph transformation rules for this domain. The rule `joinConvoy`, see Figure 6.2(a), shows a RailCab joining a convoy of RailCabs. It specifies the creation of a `member` edge representing the RailCab's participation in the convoy operation simultaneously with its movement to the next track segment.

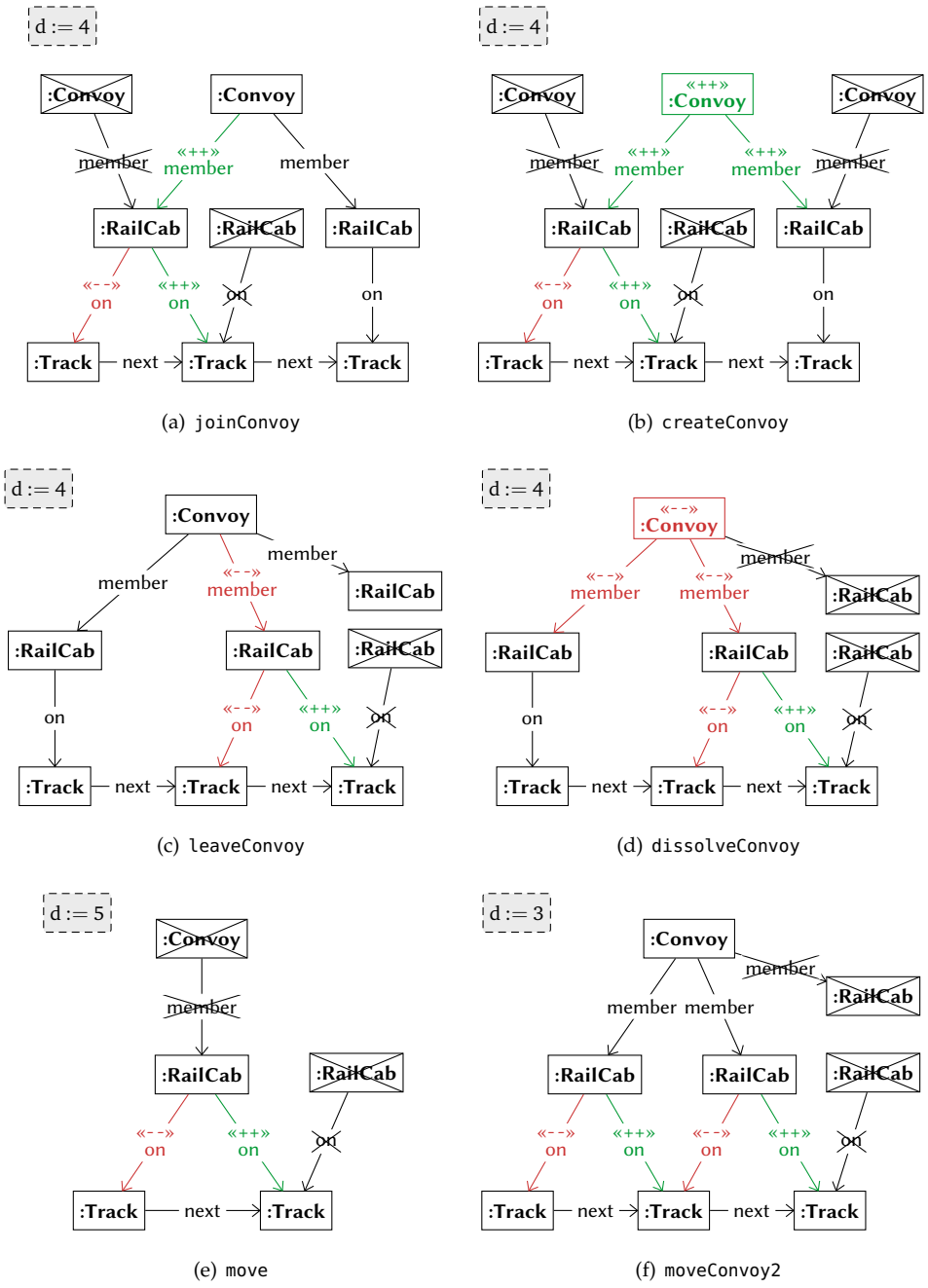


Figure 6.2: Durative graph transformation rules of the RailCab-NACs domain

The rule `createConvoy`, see Figure 6.2(b), is similar. In addition to the RailCab's movement and creation of a `member` edge, it specifies the creation of a `Convoy` node. Thus, it can be applied to configurations where the `Convoy` node is not yet available.

The rules `leaveConvoy` and `dissolveConvoy`, see Figures 6.2(c) and 6.2(d), specify reconfigurations inverse to that of `joinConvoy` and `createConvoy`. In `leaveConvoy`, only a single `member` edge from a RailCab to the `Convoy` node is deleted, whereas in `dissolveConvoy` the `Convoy` node itself is deleted.

The rule `move`, see Figure 6.2(e), simply specifies the movement of a RailCab to the next track segment. It can only be applied if the RailCab does not participate in a convoy operation. For the movement of RailCabs that do, there are separate rules. The rules `moveConvoy2`, see Figure 6.2(f), moves a convoy of two RailCabs, the rule `moveConvoy3`, which is not shown, a convoy of 3 RailCabs.

Since RailCabs drive slower if they are on their own, `move` has a duration of 5 time units, while `moveConvoy2` and `moveConvoy3` each have a duration of 3 time units. All the other rules have a duration of 4 time units.

The initial configuration of a planning task defines the complete railway network as well as initial positions for the RailCabs available in the system. The goal specification is given by the RailCabs' desired target positions. During operation of the RailCab system, initial configurations for the planning subsystem can be generated from actual runtime states of the system. Goal specifications are either constructed from user input or predefined by the system designer.

The different problem instances used for evaluation are illustrated in Sections 6.5 and 6.6 along with the results of the experiments. For these problem instances, the employed planning systems computed temporal reconfiguration plans. In accordance with the domains that have been generated from the graph transformation rules shown above, these plans take advantage of parallel execution of actions when possible while guaranteeing that concurrently executed actions do not interfere with each other. With regard to the application scenario, this means that multiple RailCabs are driving at the same time if they are both members of the same convoy or operating sufficiently apart from each other, but wait for other RailCabs if necessary, e.g., to clear a common track segment.

Listing 6.1 shows an excerpt of a plan computed by SGPlan_6 [CWH06; HW08] for a problem instance involving 4 RailCabs. As stated in Section 6.1, the plan corresponds to a set of tuples of points in time and durative graph transformations, which are represented as ground actions. During the interval from 39 to 42, `railcab4` and `railcab2` operate in a convoy. From 42 to 46, they break up the convoy operation because the underlying domain specifies a Y junction between `maintrack_6`, `endtrack_1_1`, and `endtrack_2_1`, and they need to move along different routes to arrive at their target locations. In order to do so, `railcab4` has to fall back, i.e., it still occupies `maintrack_5` at 46. From 46 to 50, `railcab4` instantiates a new convoy with `railcab3`, which needs to move along the same route as `railcab4` and waited on `endtrack_2_1` since 39 for `railcab4`.

Listing 6.1: Excerpt of a plan for 4 RailCabs

```
1: 30.016: (MOVECONVOY2 convoy1 railcab4 railcab2 maintrack_3 maintrack_4   ↙
           ↪    maintrack_5)  [3.0000]
2: 30.017: (BREAKCONVOY convoy2 railcab3 railcab1 maintrack_6 endtrack_1_1   ↙
           ↪    endtrack_1_2)  [4.0000]
3: 34.018: (MOVE railcab3 maintrack_6 endtrack_2_1)  [5.0000]
4: 39.019: (MOVECONVOY2 convoy1 railcab4 railcab2 maintrack_4 maintrack_5   ↙
           ↪    maintrack_6)  [3.0000]
5: 42.020: (BREAKCONVOY convoy1 railcab4 railcab2 maintrack_5 maintrack_6   ↙
           ↪    endtrack_1_1)  [4.0000]
6: 46.021: (CREATECONVOY convoy2 railcab4 railcab3 maintrack_5 maintrack_6   ↙
           ↪    endtrack_2_1)  [4.0000]
7: 46.022: (MOVE railcab1 endtrack_1_2 endtrack_1_3)  [5.0000]
8: 50.023: (MOVECONVOY2 convoy2 railcab4 railcab3 maintrack_6 endtrack_2_1   ↙
           ↪    endtrack_2_2)  [3.0000]
9: 51.024: (CREATECONVOY convoy1 railcab2 railcab1 endtrack_1_1   ↙
           ↪    endtrack_1_2 endtrack_1_3)  [4.0000]
```

6.3 Translation Scheme

This section explains the construction of a PDDL domain file out of a given durative graph transformation system $\mathcal{DS} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR})$ or temporal graph transformation planning problem $\mathcal{TP} = (\mathcal{TG}, G_0^T, \mathcal{DR}, \mathcal{CR}, \mathcal{UR}, P_{tgt})$. Roughly speaking, the type graph $\mathcal{TG}$ yields the declarations (of types and predicates) in the domain file and each durative rule in $\mathcal{DR}$ yields an action schema. In doing so, the LHS of a durative rule makes up the largest part of the action's precondition. The difference between the RHS and LHS yields the effect of the action. Each concurrency rule in $\mathcal{CR}$ and each urgency rule in $\mathcal{UR}$ causes further literals within those preconditions and effects. Each urgency rule in $\mathcal{UR}$ also yields a separate action schema, called *clip*, which causes actions between which no time is allowed to pass to be scheduled closely to one another. Like envelopes, clips are a concept known in PDDL domain modeling, see [HLF03].

A quick overview of the complete translation scheme is given in Table 6.1. Serving as a reference, this overview includes all different variants developed for the support of forbidden pairs and the treatment of dangling edges. Details of this translation scheme and its variants are given in the following subsections.

Note that our running example does not make use of attribute conditions or manipulations. Attribute expressions for boolean and numerical attributes are conceptually simple to support in PDDL and have already been presented in [TK11]. Boolean attribute expressions can be supported by treating them exactly like self edges, and numerical attribute expressions can be supported by translating them into numeric facts and numeric assignments.

Remember that a planning task consists of a domain and a problem file. While domain files are generated according to the translation scheme, problem files are

Table 6.1: Overview of the translation scheme. Table rows with an abbreviation to the right of the table indicate entries relevant for certain translation variants only. The abbreviations are introduced in Section 6.5 and not needed before.

Type level		
type	declaration of object	
type / edge type	declaration of predicate	
edge type	declaration of two counting functions	*-C
Rule level		
node	parameter (with inequality checks in case of injective matching of LHSs)	
preserved node/edge	positive literal	
deletion node/edge	positive literal $\rightarrow$ negative literal	
forbidden edge	negative literal	
creation node/edge	negative literal $\rightarrow$ positive literal	
forbidden pair	negative existential quantification (with inequality checks in case of injective matching of NACs)	*-Q
forbidden pair	numeric fact on counting function	*-C
deletion/creation edge	numeric assignments on counting functions	*-C
deletion node	absence of dangling edges ensured analogously to forbidden pairs	DPO-*
deletion node	dangling edge deletion realized via universal quantification	SPO-Q
deletion node	dangling edge deletion realized via new action	SPO-C
creation node	non-existence of leftover dangling edges ensured via numeric fact	SPO-C
Locking on type level		
type / edge type	declaration of read lock as predicate and write lock as function	
edge type	declaration of adjacency read and write locks as functions	
Locking on rule level		
any node/edge	check for write lock, acquire and release of read lock	
deletion/creation node/edge	check for read lock, acquire and release of write lock	
forbidden pair	check for adjacency write lock, acquire and release of adjacency read lock	
creation edge	check for adjacency read lock, acquire and release of adjacency write lock (each for source and target node)	

Table 6.1 Continued: Overview of the translation scheme

Concurrency and urgency rules on type level	
concurrency rule	declaration of envelope predicate and envelope locking function
urgency rule	declaration of clip predicates
Concurrency rules on rule level	
concurrency demanding rule	check for concurrency satisfaction, acquire and release of envelope lock
concurrency satisfying rule	check for envelope lock, indication of concurrency satisfaction during application interval
Urgency rules on rule level	
urgency rule	clip action
urgency demanding rule	indication of urgency demand via clip literal in `at_end` condition
urgency satisfying rule	indication of urgency satisfaction via clip literal in `at_start` effect

constructed from runtime data. Each problem file contains a set of objects in the world, an init section, and a goal section. For each problem file associated with the generated domain, the set of objects in the world and the init section constitute the counterpart to the initial timed graph G_0^T. The goal section results from the target graph pattern P_{tgt}.

6.3.1 Type Graph

The translation process begins with the declaration of types, predicates, and functions. All these declarations can be derived from the type graph of the durative graph transformation system. For the type graph of the RailCab-NACs domain, see Figure 6.1, the generated declarations are shown in Listing 6.2. For now, we omit the declaration of locking literals and functions.

The declarations are spread into three sections of the generated PDDL domain file: the type declaration section `:types`, the predicate declaration section `:predicates`, and the function declaration section `:functions`. Each type in the type graph gives rise to a type in the type declaration section of the file. Every such type is a subtype of `Object`, the root of the type hierarchy in PDDL. Since PDDL does not allow for object creation or deletion, we use a predicate `active` for the supertype `Object` stating whether a given object exists. Each edge type in the type graph is translated into a predicate that is parameterized by its source and target type. Since there are only three edge types in the type graph, we yield exactly three predicate declarations. Their names each are a concatenation of the edge type's label and its source and target type's label; the different labels are separated from each other via underscores.

Listing 6.2: Generated declaration of types, predicates, and functions

```
1:  (:types  Convoy - Object  RailCab - Object  Track - Object)
2:  (:predicates
3:      (active  ?object - Object)
4:      (member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
5:      (on_RailCab_Track  ?railcab - RailCab  ?track - Track)
6:      (next_Track_Track  ?track1 - Track  ?track2 - Track)
7:      ...  % declarations for locking functionality
8:  )
9:  (:functions
10:     ...  % declarations for locking functionality
11: )
```

6.3.2 Durative Graph Transformation Rules

We first cover the translation of durative graph transformation rules without addressing forbidden pairs or the locking functionality. For the durative rule `joinConvoy`, see Figure 6.2(a), Listing 6.3 shows its generated durative action. Since an LHS match has to exist at the beginning of a durative rule's application interval, we require all conditions in the action to hold `at_start` (lines 5 to 25). The changes made by the rule's application, i.e., the creation and deletion of nodes and edges, take effect `at_end` (lines 30 to 35).

Every node in the rule – regardless of whether it is preserved or going to be created or deleted – is mapped to a parameter of the action (line 2). These parameters are checked for inequality in lines 6 to 9 because we assume injective matchings in the running example, i.e., each node in the LHS has to be mapped to a different node in the host graph.

As alternatives to injective matching, the translation scheme also supports implementing inequality checks according to the DPO or SPO identification condition. The DPO implementation only employs an inequality check when at least one of two nodes of the same type is being deleted. The SPO implementation never employs any inequality checks. Since a preserved node does not generate any effect whereas a deletion node does, this favors the deletion of literals.

In PDDL, all objects have to be defined in the initial configuration of the problem file. Since it is not possible to instantiate objects dynamically, the existence of objects is worked into PDDL's propositional state representation by means of the predicate `active`. An object for which this predicate maps to *true* is considered to be instantiated; if it maps to *false*, the object is considered to be non-existing. In doing so, all those objects for which `active` maps to *true* correspond to available nodes in the LHS of the rule.

In order to specify all potential instances in the problem file, a maximum number of potential instances has to be known for each type. In some domains, this number can easily be calculated, e.g., if nodes of instantiable types are always connected to

Listing 6.3: Generated durative action for the durative rule `joinConvoy`

```
(:durative-action joinConvoy
  :parameters (?c1 - Convoy  ?r1 - RailCab  ?r2 - RailCab  ?t1 - Track
        ↪    ?t2 - Track  ?t3 - Track)
  :duration (= ?duration 4)
  :condition
    (at start (and
      (not (= ?r1 ?r2))
      (not (= ?t1 ?t2))
      (not (= ?t1 ?t3))
      (not (= ?t2 ?t3))
      (active ?c1)
      (active ?r1)
      (active ?r2)
      (active ?t1)
      (active ?t2)
      (active ?t3)
      (member_Convoy_RailCab ?c1 ?r2)
      (on_RailCab_Track ?r1 ?t1)
      (on_RailCab_Track ?r2 ?t3)
      (next_Track_Track ?t1 ?t2)
      (next_Track_Track ?t2 ?t3)
      (not (member_Convoy_RailCab ?c1 ?r1))
      (not (on_RailCab_Track ?r1 ?t2))
      ...  % conditions caused by forbidden pairs
      ...  % checking for locks
    ))
  :effect (and
    (at start (and
      ...  % locking
    ))
    (at end (and
      (not (on_RailCab_Track ?r1 ?t1))
      (member_Convoy_RailCab ?c1 ?r1)
      (on_RailCab_Track ?r1 ?t2)
      ...  % unlocking
    ))
  )
)
```

nodes of non-instantiable types, and it is not possible to have more than one instance connected to the same non-instantiable node. This is the case in the RailCab-NACs domain: `Convoy` nodes are always connected to non-instantiable `RailCab` nodes, and no `RailCab` node may be connected to more than one `Convoy` node. In other domains, the domain designer's expert knowledge has to be used to choose a number sufficiently large to yield a valid plan for the given domain.

To specify that each of the rule's LHS' nodes has to exist, the literals in lines 10 to 15 of Listing 6.3 are generated. Instantiation and deinstantiation of nodes can be realized by changing such literals from *false* to *true* and vice versa, respectively. In this case, there is no such literal in the effect of the action, because no node is deleted by `joinConvoy`.

Required edges, i.e., preserved edges and deletion edges, cause the remaining literals in the condition of Listing 6.3. If there were forbidden edges in `joinConvoy`, they would have been translated into negative literals. Although there are none, our translation produces a few negative literals. This is because we assume that no parallel edges, i.e., edges with identical source, target, and label, are allowed. To prevent the creation of parallel edges, every edge that is going to be created implies a forbidden edge. These implicit forbidden edges produce the negative literals in lines 21 and 22.

A modeling approach based on a semantics that allows parallel edges could easily be supported by changing the predicates that state the existence of edges into functions that return the number of their instances. We decided to disallow parallel edges in order to be able to implement the existence of edges via predicates, which are a more basic construct of PDDL than functions. Another alternative to support parallel edges has already been given in Section 5.2.4: since graphs employing parallel edges can be simulated by graphs without parallel edges, cf. [Bon+07], we can transform a durative graph transformation system with parallel edges into an equivalent durative graph transformation system without parallel edges.

The negative literal in the effect (line 31) is caused by the `on` edge being deleted and the two positive literals (lines 32 and 33) by the `member` and `on` edge being created.

6.3.3 Forbidden Pairs

While a forbidden edge can simply be mapped into a negative literal, a forbidden pair, e.g., one of the NACs in the durative graph transformation rule `dissolveConvoy`, see Figure 6.2(d) on page 144, cannot be realized as easily. One solution is to map a forbidden pair into a negative existential quantification over the conjunction of node type and adjacent edge that connects the node to the LHS. This is shown in Listing 6.4 for one of the forbidden pairs in `dissolveConvoy`. The forbidden `RailCab` node is declared in line 4 as part of the existential quantification. The literals in lines 7 and 8 specify that the `RailCab` node is not allowed to exist if it is connected via a `member` edge to the `Convoy` node. An isolated forbidden node, i.e., when no connecting edge exists, can simply be mapped into a negative existential quantification involving only the `active` predicate. In both cases, inequality conditions can be added (lines 5 and 6) if the node type has already been matched as a parameter, to be in accordance with employing injective matchings for NACs. Note that the choice whether or not to employ injective matchings for NACs can be made independent of the choice made for LHSs.

Unfortunately, preconditions containing quantifications are not supported by

Listing 6.4: Generated negative existential quantification for a forbidden pair

```
1:  :condition
2:    (at start (and
3:      ...  % more literals/facts
4:      (not (exists (?r - RailCab) (and
5:        (not (= ?r ?r1))
6:        (not (= ?r ?r2))
7:        (active ?r)
8:        (member_Convoy_RailCab ?c1 ?r)
9:      ))))
10:     ...  % further negative existantial quantifications caused by
                ↪    forbidden pairs
11:     ...  % checking for locks
12:   ))
```

every PDDL-based planning system. State-of-the-art temporal planners that support required concurrency, like POPF [Col+10] or YAHSP2-MT [Vid11], usually do not support them. Fortunately, forbidden pairs can also be realized in PDDL as numeric facts. We can define two counting functions $f_{et,src} : V \to \mathbb{N}$ and $f_{et,tgt} : V \to \mathbb{N}$ for each edge type et, one for its source type and one for its target type. Then, we count for each node its number of incoming and outgoing edges of each edge type in the initial configuration and update these counters each time an edge is created or deleted. For a `Convoy` node v with two outgoing `member` edges, the counting function for the source type returns $f_{\texttt{member},src}(v) = 2$. For each target `RailCab` node u_i, the counting function for the target type returns $f_{\texttt{member},tgt}(u_i) = 1$ because each RailCab is member of only one convoy. Knowing these numbers, when there is a node connected to a forbidden pair, e.g., the `Convoy` node in `dissolveConvoy`, we can implement the forbidden pair via a numeric fact on one of the counting functions: we simply require that the node is adjacent to exactly that many edges of the edge type involved in the forbidden pair as specified in the LHS of the rule. In case of the `Convoy` node in `dissolveConvoy`, we thus require that it is adjacent to two `member` edges. By employing these counting functions, checking for a forbidden pair comes down to checking for one numeric fact.

Listing 6.5 shows the function declarations of the counting functions that result from the rule `dissolveConvoy`. There are two counting functions for each edge type: one for counting the outgoing edges of source nodes and one for counting the incoming edges of target nodes. The numeric facts and update assignments that result from `dissolveConvoy` are shown in Listing 6.6. The numeric fact in line 4 is generated for the forbidden pair adjacent to the `Convoy` node. Since the `Convoy` node has two outgoing `member` edges that each connect to a `RailCab` node, the value of the numeric fact is 2. The numeric fact in line 5 is generated for the forbidden pair adjacent to the rightmost `Track` node. It has no incoming `on` edge that connects to a `RailCab` node. Thus, the numeric fact has a value of 0. The update assignments in

Listing 6.5: Generated declarations for the counting functionality

```
(:functions
    ...  % more declarations
    (counterAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
    (counterAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
    (counterAdjacentToSource_on_RailCab_Track  ?railcab - RailCab)
    (counterAdjacentToTarget_on_RailCab_Track  ?track - Track)
    (counterAdjacentToSource_next_Track_Track  ?track - Track)
    (counterAdjacentToTarget_next_Track_Track  ?track - Track)
    ...  % declarations for locking functionality
)
```

Listing 6.6: Generated numeric facts and assignments for forbidden pairs

```
:condition
  (at start (and
    ...  % more literals/facts
    (= (counterAdjacentToSource_member_Convoy_RailCab ?c1) 2)
    (= (counterAdjacentToTarget_on_RailCab_Track ?t3) 0)
    ...  % checking for locks
  ))
:effect (and
  (at start (and
    ...  % locking
  ))
  (at end (and
    ...  % more literals/assignments
    (decrease (counterAdjacentToSource_member_Convoy_RailCab ?c1) 2)
    (decrease (counterAdjacentToTarget_member_Convoy_RailCab ?r1) 1)
    (decrease (counterAdjacentToTarget_member_Convoy_RailCab ?r2) 1)
    (decrease (counterAdjacentToTarget_on_RailCab_Track ?t2) 1)
    (increase (counterAdjacentToTarget_on_RailCab_Track ?t3) 1)
    ...  % unlocking
  ))
```

lines 14 to 18 are generated for the creation and deletion edges. For the rightmost `RailCab` node, an outgoing `on` edge is deleted while another outgoing `on` edge is created. Since they balance out, there is no update assignment for the number of `on` edges of this `RailCab` node.

We expected the first variant, i.e., negative existential quantifications, to be more costly than the second, i.e., counting functions, which is why we specifically compare these two variants during the experiments in Section 6.5.

6.3.4 Dangling Edges

When a rule with one or more deletion nodes is applied, the problem of dangling edges can occur. This problem is handled differently by DPO and SPO. In DPO, the application of a rule is not allowed at a match if it raises one or more dangling edges. In SPO, dangling edges are simply deleted; there is no further restriction on the rule's applicability. Both variants can be transferred into PDDL.

The DPO variant results either in negative existential quantifications or in numeric facts on counting functions. This is no choice; it depends on how forbidden pairs are translated. For each edge type whose edges can be adjacent to a ode being deleted, a condition analogous to that of forbidden pairs is generated. If there already is a forbidden pair connected to the deletion node, this condition is redundant.

How the SPO variant is realized also depends on whether negative existential quantifications or counting functions have been chosen for supporting forbidden pairs. In the case of negative existential quantifications, the deletion of dangling edges is realized via universal quantifications in the `at_end` effect of the generated action. Each quantification contains a negative literal representing the potential deletion of an edge that is adjacent to the node being deleted and the quantified node. Listing 6.7 shows the generated universal quantification for the deletion of dangling edges in the rule `dissolveConvoy`. Note that in PDDL, a literal does not have to be *true* when being asserted as *false*. Therefore, the action can still be applied when there are no dangling edges. The non-existence of dangling edges is simply reasserted by the action's effect.

Listing 6.7: Generated universal quantification for deleting dangling edges (in the SPO variant with quantifications)

```
1:  :effect (and
2:    ...  % at start effect
3:    (at end (and
4:      ...  % more literals/assignments
5:      (forall (?r - RailCab)
6:        (not (member_Convoy_RailCab ?c1 ?r))  % ?c1 is a parameter
7:      )
8:      ...  % unlocking
9:    ))
```

In the case of counting functions, we also have to update the values of the counters for the dangling edges being deleted. Simply decreasing the counter value for every potential deletion of an edge would result in incorrect values where no dangling edge existed in the first place. The solution is to delete dangling edges via a new action, which ensures that the danging edge exists. Listing 6.8 shows such an action for the dangling edges of `dissolveConvoy`. Since the application of such an action is optional, this can result in dangling edges remaining in the configuration. Leftover dangling edges could pose a problem if a deleted node was recreated later,

Listing 6.8: Generated durative action for deleting dangling edges (in the SPO variant with counting functions)

```
1:  (:durative-action deleteDanglingEdges_member_Convoy_RailCab
2:    :parameters (?c1 - Convoy ?r1 - RailCab)
3:    :duration (= ?duration 0.01)
4:    :condition (at start (and
5:      (not (active ?c1))
6:      (member_Convoy_RailCab ?c1 ?r1)
7:    ))
8:    :effect (at end (and
9:      (not (member_Convoy_RailCab ?c1 ?r1))
10:     (decrease (counterAdjacentToSource_member_Convoy_RailCab ?c1) 1)
11:     (decrease (counterAdjacentToTarget_member_Convoy_RailCab ?r1) 1)
12:   ))
13: )
```

reconnecting to a dangling edge. To prevent this from happening, we simply require no dangling edge to exist each time a node is created. This is done via a numeric fact checking that the counting function's value for the creation node is zero.

6.3.5 Locking Functionality

Now, we extend the translation scheme to integrate a locking mechanism analogously to that in the semantics of durative graph transformation rules. First, the declaration of predicates and functions has to be extended to include the declaration of locks. For the RailCabs-NACs domain, the generated predicate and function declarations for the locks are shown in Listing 6.9. For the sake of clarity, locking declarations are only shown for one of the edge types. There is one pair of locks (read and write lock) for each node (lines 3 and 9) and one pair of locks for each edge in the type graph (lines 4 and 10). Write locks on nodes and edges are realized as predicates (exclusive lock, *true* means locked) because a node or edge may not be accessed in any way if it is being deleted at the moment (or created in case of a forbidden edge). Concurrent access to a node or an edge is allowed if the rule does not manipulate it. Therefore, read locks are realized as functions (shared lock, greater than zero means locked).

The idea of the remaining locking predicates (lines 11 to 14) is more subtle. Nodes within NACs cannot be locked via any of the aforementioned locking predicates, because such node do not exist in a configuration, i.e., there is no explicit object in PDDL's propositional state representation that constitutes the node in the NAC. Instead, locking information is added to connecting nodes, i.e., those nodes of the LHS that the NAC is connected to. This, of course, restricts our approach to specific kinds of NACs, namely forbidden edges and forbidden pairs.

Locking of forbidden edges is already supported via the locking predicates for edges. Locking of forbidden pairs is supported by the functions in lines 11 and 12,

Listing 6.9: Generated declarations for the locking functionality

```
1:  (:predicates
2:      ...  % basis declarations
3:      (writeNode_active  ?object - Object)
4:      (writeEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
5:      ...  % further declarations for the locking functionality
6:  )
7:  (:functions
8:      ...  % basis declarations
9:      (readNode_active  ?object - Object)
10:     (readEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
11:     (readAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
12:     (readAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
13:     (writeAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
14:     (writeAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
15:     ...  % further declarations for the locking functionality
16: )
```

which – pictorially speaking – add locking information to the connecting node. For each edge predicate, there is a pair of locking predicates: the first locking predicate is used to prevent the creation of forbidden pairs with outgoing edges, the second one prevents forbidden pairs with incoming edges.

Nodes that are being added along with edges connecting it to already existing nodes in the LHS are locked in a similar fashion via the functions in lines 13 and 14. However, for nodes that are being added, locking information is attached to *all* existing nodes that the new nodes are going to be connected to. Consider a node that is created along with two edges that are both adjacent to a different node in the LHS. In this case, both nodes in the LHS are connecting nodes and both acquire a write lock disallowing a certain forbidden pair. If there is no connecting node for a newly created node, then there is no need to lock anything since isolated nodes do not interfere with any forbidden pair.

Note that it is possible that multiple new nodes get connected to the same existing node simultaneously – possibly due to the concurrent application of multiple rules. For this reason, write locks are realized as functions instead of predicates. Also note that while this approach supports only two specific kinds of NACs, i.e., forbidden edges and forbidden pairs, it supports any kind of RHS.

6.3.6 Locks in Durative Rules

We will now treat the generation of locking literals for the conditions and effects of action schemata and turn to the example of `joinConvoy` again. Listing 6.10 shows the locking literals generated to support the locking of required nodes. For every positive literal that represents the existence of a node, a negative locking literal is added to the condition of the durative action (lines 4 to 9). These literals ensure

Listing 6.10: Generated locks to support (required) nodes

```
1:  :condition
2:    (at start (and
3:      ...  % basis literals/facts
4:      (not (writeNode_active ?c1))  % preserved node
5:      (not (writeNode_active ?r1))  % preserved node
6:      (not (writeNode_active ?r2))  % preserved node
7:      (not (writeNode_active ?t1))  % preserved node
8:      (not (writeNode_active ?t2))  % preserved node
9:      (not (writeNode_active ?t3))  % preserved node
10:     ...  % further literals/facts for checking for locks
11:   ))
12: :effect (and
13:   (at start (and
14:     (increase (readNode_active ?c1) 1)  % preserved node
15:     (increase (readNode_active ?r1) 1)  % preserved node
16:     (increase (readNode_active ?r2) 1)  % preserved node
17:     (increase (readNode_active ?t1) 1)  % preserved node
18:     (increase (readNode_active ?t2) 1)  % preserved node
19:     (increase (readNode_active ?t3) 1)  % preserved node
20:     ...  % further literals/assignments for locking
21:   ))
22:   (at end (and
23:     ...  % basis literals/assignments
24:     (decrease (readNode_active ?c1) 1)  % preserved node
25:     (decrease (readNode_active ?r1) 1)  % preserved node
26:     (decrease (readNode_active ?r2) 1)  % preserved node
27:     (decrease (readNode_active ?t1) 1)  % preserved node
28:     (decrease (readNode_active ?t2) 1)  % preserved node
29:     (decrease (readNode_active ?t3) 1)  % preserved node
30:     ...  % further literals/assignments for unlocking
31:   ))
```

that none of the required nodes, i.e., preserved nodes and deletion nodes, has been locked for writing. If applied, the action locks the required nodes for reading, i.e., it acquires a shared read lock when it is scheduled to begin (lines 14 to 19) and releases it when it ends (lines 24 to 29). In general, deletion nodes also have to be locked for writing. However, there is no deletion node in `joinConvoy`, so there is no checking of read locks in the condition or acquiring and releasing of write locks in the effect.

The locking literals of required edges, i.e., preserved edges and deletion edges, are realized similarly. For `joinConvoy`, they are shown in Listing 6.11. As with deletion nodes, deletion edges also have to be locked for writing. An example is the `on` edge from the left RailCab to the leftmost track segment. Since it is being deleted,

Listing 6.11: Generated locks to support required and forbidden edges

```
1:  :condition
2:    (at start (and
3:      ...  % basis literals/facts
4:      ...  % more literals/facts for checking for locks
5:      (not (writeEdge_member_Convoy_RailCab ?c ?r2))  % preserved edge
6:      (not (writeEdge_on_RailCab_Track ?r2 ?t3))  % preserved edge
7:      (not (writeEdge_next_Track_Track ?t1 ?t2))  % preserved edge
8:      (not (writeEdge_next_Track_Track ?t2 ?t3))  % preserved edge
9:      (not (writeEdge_on_RailCab_Track ?r1 ?t1))  % deletion edge
10:     (= (readEdge_on_RailCab_Track ?r1 ?t1) 0)   % deletion edge
11:     ...  % further literals/facts for checking for locks
12:   ))
13: :effect (and
14:   (at start (and
15:     ...  % more literals/assignments for locking
16:     (increase (readEdge_member_Convoy_RailCab ?c ?r2) 1)  % preserved edge
17:     (increase (readEdge_on_RailCab_Track ?r2 ?t3) 1)  % preserved edge
18:     (increase (readEdge_next_Track_Track ?t1 ?t2) 1)  % preserved edge
19:     (increase (readEdge_next_Track_Track ?t2 ?t3) 1)  % preserved edge
20:     (increase (readEdge_on_RailCab_Track ?r1 ?t1) 1)  % deletion edge
21:     (writeEdge_on_RailCab_Track ?r1 ?t1)              % deletion edge
22:     ...  % further literals/assignments for locking
23:   ))
24:   (at end (and
25:     ...  % basis literals/assignments
26:     ...  % literal/assignments for unlocking, analogous to 'at start'
27:   ))
```

it is locked for writing in line 10. It is also locked for reading in line 9 because each edge that is being deleted is also required in the LHS.

Note that the implementation of the locking mechanism does not depend on the choice of semantics to handle the problem of dangling edges. Even though dangling edges have not been locked by the start effect when they are deleted, this cannot lead to conflicts with actions requiring those edges. This is already prevented by the locking of nodes: each action accessing the dangling edge would also have to access both its adjacent nodes; however, at least one of these nodes is write locked because it is going to be deleted by the action, thus causing the dangling edge.

Since object instantiation is realized via a workaround involving literals that state whether or not a node exists, creation nodes need some special treatment in PDDL. Consider the situation when two nodes of the same type are instantiated simultaneously by two different actions. In the DGTS formalism, both nodes do not yet exist at the beginning of executing the actions and thus cannot be the reason for any conflicts. However, since we are working on existing objects in PDDL, there

can be a conflict when both actions "instantiate" the same node, i.e., both set the same node's `active` literal to *true*. Therefore, creation nodes lock their `active` literal analogously to deletion nodes.

Although forbidden edges result in negative literals in an action's condition, their locking functionality works analogously to that of preserved edges. Forbidden edges even use the same locking predicates as preserved edges. Note that as stated in Section 6.3.2, every creation edge also implies a forbidden edge.

Similar to conflicts between a required node or edge of one rule application and the deletion of this node or edge by another, concurrent rule application, there can be conflicts between forbidden pairs and the creation of nodes and edges. A forbidden pair might even interfere with creating *only* an edge, because the edge could be adjacent to an already available node of the same type as the node in the forbidden pair. To avoid such conflicts between forbidden pairs and the creation of edges, we need further locking functionality.

Listing 6.12 shows the numeric facts and assignments that implement this locking functionality. The numeric facts in lines 5 and 6 are generated for the two forbidden pairs of `joinConvoy`. Each numeric fact guarantees that no action has already been started that creates an edge that is of the same type and direction as the edge contained in the forbidden pair of `joinConvoy` and adjacent to the connecting node. The locking literals in lines 7 to 10 are generated for the creation edges of `joinConvoy`. They state that no forbidden pair lock may be acquired for any of the nodes that are adjacent to the creation edges. If such a lock is acquired by a concurrently applied action, that action contains a forbidden pair connected to the same node. The locking literals in the effect itself acquire read locks for forbidden pairs to ensure that no creation edge conflicts with them (lines 15 and 16) and write locks to ensure that no concurrent action with a conflicting forbidden pair will be applied (lines 17 to 20).

Listing 6.12: Generated adjacency locks to support forbidden pairs

```
1:  :condition
2:    (at start (and
3:      ...  % basis literals/facts
4:      ...  % more literals/facts for checking for locks
5:      (= (writeAdjacentToTarget_member_Convoy_RailCab ?r1) 0)  %  ↙
                ↪    forbidden pair
6:      (= (writeAdjacentToTarget_on_RailCab_Track ?t2) 0)  % forbidden pair
7:      (= (readAdjacentToSource_member_Convoy_RailCab ?c1) 0)  % creation edge
8:      (= (readAdjacentToTarget_member_Convoy_RailCab ?r1) 0)  % creation edge
9:      (= (readAdjacentToSource_on_RailCab_Track ?r1) 0)  % creation edge
10:     (= (readAdjacentToTarget_on_RailCab_Track ?t2) 0)  % creation edge
11:   ))
12: :effect (and
13:   (at start (and
14:     ...  % more literals/assignments for locking
15:     (increase (readAdjacentToTarget_member_Convoy_RailCab ?r1) 1)  %  ↙
                ↪    forbidden pair
16:     (increase (readAdjacentToTarget_on_RailCab_Track ?t2) 1)  %  ↙
                ↪    forbidden pair
17:     (increase (writeAdjacentToSource_member_Convoy_RailCab ?c1) 1)  %  ↙
                ↪    creation edge
18:     (increase (writeAdjacentToTarget_member_Convoy_RailCab ?r1) 1)  %  ↙
                ↪    creation edge
19:     (increase (writeAdjacentToSource_on_RailCab_Track ?r1) 1)  %  ↙
                ↪    creation edge
20:     (increase (writeAdjacentToTarget_on_RailCab_Track ?t2) 1)  %  ↙
                ↪    creation edge
21:   ))
22:   (at end (and
23:     ...  % basis literals/assignments
24:     ...  % literal/assignments for unlocking, analogous to 'at start'
25:   ))
```

6.3.7 Concurrency Rules

The implementation of concurrency rules in PDDL works similarly to that in the semantics. In both cases, the satisfying rule signals the satisfaction of the other rule's demand. In the DGTS formalism, this is done using a satisfaction indicator, which is locked during the application of a demanding rule to prevent that the satisfying rule finishes before the demanding rule finishes. In PDDL, we use designated literals to achieve a similar effect.

The semantics of concurrency rules requires that each application of a demanding rule is enclosed by that of a satisfying rule. In PDDL, an action that is required to enclose the application of another action is called *envelope*. Figure 6.3 illustrates the interplay between an envelope and its enclosed action.

Figure 6.3: Interplay between an envelope action and its enclosed action. The left end of an action's line represents the moment when the action's execution starts; the right end represents the end of its execution. Literals written above the action's line denote conditions; literals below the line denote effects.

The literal x in Figure 6.3 plays the role of the satisfaction indicator. It is set and unset by the envelope at the beginning and ending of its execution, respectively, and required by the enclosed action. The literal is further parameterized via the nodes in the satisfier interface subgraph. Note that the satisfier interface subgraph has to be available in the enclosed action for this to work. For this reason, we translate the demanding rule extended by the connecting graph of the concurrency rule, see Definitions 5.5.2 and 5.5.3, instead of the original demanding rule.

To guarantee that the envelope finishes its execution after the enclosed action finished its execution, we need a means to check whether or not the enclosed action is still being executed. In the semantics, this is done by applying a lock to the satisfaction indicator. Here, we simply use a numeric fact y, which is parameterized via the same nodes as x. It is increased and decreased by the enclosed rule and required not to be existing when the envelope finishes its execution.

There are two concurrency rules in the RailCab domain. The concurrency rule `allowChangePublication` requires the application of a durative rule moving a RailCab from one track segment to the next, e.g., `moveRailCab` or `formConvoy`, to happen concurrently to that of the durative rule `changePublication`. The second concurrency rule is `allowChangePublicationWhileInConvoy`. Here, the demanding rule is `changePublicationWhileInConvoy`, which differs from `changePublication`

in that it requires the RailCab to drive in a convoy, and satisfying rules are all those rules that move a convoy from one track segment to the next.

The generated declarations for both concurrency rules are shown in Listing 6.13. Locks are realized as functions because there can be different demanding rules relying on the same satisfying rule at the same time.

Listing 6.13: Generated declarations to support concurrency rules

```
1:  (:predicates
2:      ...  % basis declarations
3:      ...  % declarations for the locking functionality
4:      (envelope_allowChangePub  ?railcab - RailCab  ?track1 - Track    ↙
                ↪    ?track2 - Track)
5:      (envelope_allowChangePubWhileInConvoy  ?convoy - Convoy  ?track1 -   ↙
                ↪    Track  ?track2 - Track)
6:  )
7:  (:functions
8:      ...  % basis declarations
9:      ...  % declarations for the locking functionality
10:     (lock_envelope_allowChangePub  ?railcab - RailCab  ?track1 - Track    ↙
                ↪    ?track2 - Track)
11:     (lock_envelope_allowChangePubWhileInConvoy  ?convoy - Convoy    ↙
                ↪    ?track1 - Track  ?track2 - Track)
12: )
```

For demanding action `changePublication` and satisfying action `moveRailCab`, Listing 6.14 and Listing 6.15 show those literals, numeric facts, and numeric assignments that are added to implement the functionality of concurrency rule `allowChangePublication`. The literals starting with `envelope_` correspond to the x in Figure 6.3, the numeric assignments and facts starting with `lock_envelope_` to the y.

Listing 6.14: Generated concurrency demand in the rule `changePublication`

```
:condition
  (at start (and
    ...  % basis literals/facts
    ...  % literals/facts for checking for locks
    (envelope_allowChangePub ?r1 ?t1 ?t2)
  ))
:effect (and
  (at start (and
    ...  % literals/assignments for locking
    (increase (lock_envelope_allowChangePub ?r1 ?t1 ?t2) 1)
  ))
  (at end (and
    ...  % basis literals/assignments
    ...  % literals/assignments for unlocking
    (decrease (lock_envelope_allowChangePub ?r1 ?t1 ?t2) 1)
  ))
)
```

Listing 6.15: Generated concurrency satisfaction in the rule `moveRailCab`

```
:condition (and
  (at start (and
    ...  % basis literals/facts
    ...  % literals/facts for checking for locks
  ))
  (at end (= (lock_envelope_allowChangePub ?r1 ?t1 ?t2) 0))
)
:effect (and
  (at start (and
    ...  % literals/assignments for locking
    (envelope_allowChangePub ?r1 ?t1 ?t2))
  ))
  (at end (and
    ...  % basis literals/assignments
    ...  % literals/assignments for unlocking
    (not (envelope_allowChangePub ?r1 ?t1 ?t2)))
  ))
)
```

6.3.8 Urgency Rules

The implementation of urgency rules in PDDL differs more from that in the semantics than it was the case for concurrency rules. In the DGTS formalism, both the demanding and satisfying rule add a demand and a satisfaction indicator, respectively, into the configuration. To ensure the application of the satisfying rule, the semantics employs a satisfier-firing timed and a satisfier-firing invariant rule. The satisfier-firing invariant rule matches as soon as the demanding rule finishes and enforces an application of the satisfier-firing timed rule via a clock instance constraint. The satisfier-firing timed rule, in turn, requires an application of the satisfying rule to be able to find a match.

In PDDL – at least in PDDL 2.1 level 3 – there is no such thing as an invariant rule. Instead, we use a very short action that encloses the end of the demanding action and the begin of the satisfying action. Such an action is called *clip*. Figure 6.4 illustrates the interplay between a clip and two consecutive actions that have been clipped together.

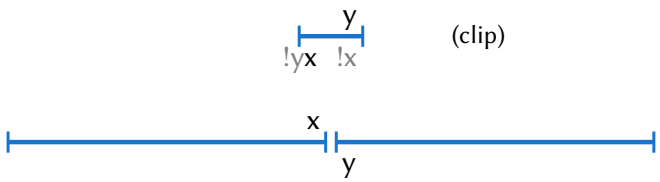


Figure 6.4: Interplay between two consecutive actions and a clip action clipping them together. The left end of an action's line represents the moment when the action's execution starts; the right end represents the end of its execution. Literals written above the action's line denote conditions; literals below the line denote effects.

The literal x plays the role of the demand indicator, y that of the satisfaction indicator. Since x is required when the demanding action ends and set only when the clip starts, the clip has to start before the demanding action ends. The situation with y is similar: this time the clip requires the literal, which forces the satisfying action to start before the clip ends. Both literals are parameterized via the nodes in the satisfier and demander subgraph, respectively.

Note that while the semantics also produces satisfier-cleaning timed and invariant rules to remove needless satisfaction indicators, this is not necessary in PDDL. The reason these satisfaction indicators are removed is that we do not want them to be available later, because they could satisfy an unrelated demand arising from an urgency rule with the same satisfaction subgraph. There is an easier way to prevent this in PDDL. By asserting the literal y as *false* at the beginning of the clip's execution, we ensure that the satisfying action is executed *after* the clip started to satisfy the clip's demand for the literal. This is possible in PDDL because a literal does not have to be *true* when being asserted as *false*.

Since the clip is an ordinary durative action, it is possible to execute this clip at

any time in the plan. Imagine the clip being executed long before a demanding action is executed. Its execution provides a literal x, which is necessary for the demanding action to end. If this literal was not deleted again by the clip, it would stay available in the configuration and allow an unrelated application of a demanding action, which occurs later in the plan, to end without another application of the clip action. For this reason, the clip asserts the literal x as *false* when it ends.

The RailCab domain includes two urgency rules: `immediatelyMoveRailCab` and `immediatelyMoveConvoy`. Autonomously operating RailCabs and convoys of RailCabs both are not allowed to stop abruptly, i.e., no pause is allowed between the application of a durative rule that ends with a RailCab or a convoy in driving motion and another successive rule application that continues this movement. Demanding rules are those rules that end with RailCabs or convoys in driving motion, e.g., `accelerateRailCab` or `moveRailCab`, and satisfying rules are those rules that continue the movement, e.g., `moveRailCab` or `brakeRailCab`.

The generated declarations for both urgency rules are shown in Listing 6.16. Here, all functionality can be realized as predicates.

Listing 6.16: Generated declarations to support urgency rules

```
(:predicates
    ...  % basis declarations
    ...  % declarations for the locking functionality
    ...  % declarations for concurrency rules
    (clipX_immediatelyMoveRailCab  ?railcab - RailCab)
    (clipY_immediatelyMoveRailCab  ?railcab - RailCab)
    (clipX_immediatelyMoveConvoy  ?convoy - Convoy)
    (clipY_immediatelyMoveConvoy  ?convoy - Convoy)
)
```

The clip action generated by urgency rule `immediatelyMoveRailCab` is shown in Listing 6.17. Due to the *no moving targets* rule of PDDL, the end of the demanding action and the start of the satisfying action may not be executed at the exact same point in time. There has to be at least ε units of time between these two time points. The same holds for the start and end of the clip and the end of the demanding action and start of the satisfying action, respectively. For this reason, the duration of a clip action is 3ε higher than the deadline specified by the urgency rule inducing the clip action. Since the deadline specified by `immediatelyMoveRailCab` is 0, the duration of the clip action in Listing 6.17 is 3ε.

For demanding action `accelerateRailCab` and satisfying action `brakeRailCab`, Listing 6.18 and Listing 6.19 show those literals that are added to implement the functionality of urgency rule `immediatelyMoveRailCab`.

Listing 6.17: Clip action to support urgency rule `immediatelyMoveRailCab`

```
(:durative-action CLIP_immediatelyMoveRailCab
  :parameters (?r1 - RailCab)
  :duration (= ?duration 0.003)
  :condition (and
      (at end (clipY_immediatelyMoveRailCab ?r1))
  ) ;end condition
  :effect (and
      (at start (clipX_immediatelyMoveRailCab ?r1))
      (at start (not (clipY_immediatelyMoveRailCab ?r1)))
      (at end (not (clipX_immediatelyMoveRailCab ?r1)))
  ) ;end effect
)
```

Listing 6.18: Generated urgency demand in the rule `accelerateRailCab`

```
:condition (and
  (at start (and
    ...  % basis literals/facts
    ...  % literals/facts for checking for locks
  ))
  (at end (clipX_immediatelyMoveRailCab ?r1))
)
```

Listing 6.19: Generated urgency satisfaction in the rule `brakeRailCab`

```
:effect (and
  (at start (and
    ...  % literals/assignments for locking
    (clipY_immediatelyMoveRailCab ?r1)
  ))
  (at end (and
    ...  % basis literals/assignments
    ...  % literals/assignments for unlocking
  ))
)
```

6.4 Prototype and Translation Workflow

The current prototype of the translation scheme is written in Java. It uses meta-models of durative graph transformation systems and PDDL planning domains, which were both developed using the *Eclipse Modeling Framework (EMF)*[1]. An earlier prototype based on these meta-models was implemented in QVT Relations [OMG11] using *medini QVT*[2]. This prototype was used in an earlier evaluation, see [ZW15], but dropped in favor of the Java-based implementation, which supports more features and variations.

Conceptually, the Java-based prototype executes a translation workflow with multiple passes, some of which are optional. The first mandatory pass creates all basis functionality in the PDDL domain, i.e., predicates resulting from the type graph and (not yet complete) durative actions resulting from durative graph transformation rules. The next two passes depend on the chosen translation variant. They add functionality for forbidden pairs and dangling edges, respectively, into the action schemata generated by the basis pass.

The generation of all locking functionality is encapsulated into a forth pass. The same goes for the functionality of concurrency and urgency rules. Each of these three passes relies on bidirectional mappings generated by the basis pass to generate correct predicates, functions, literals, numeric assignments and facts, as well as their parameters.

Then, there are two post-processing passes, which clean up some redundancies. The first of these two passes removes all functionality related to the locking of read-only elements. The second pass settles additive assignment effects when necessary, i.e., it merges multiple numeric assignments over the same numeric function and parameters into one numeric assignment (or none if their incremental values balance out).

Finally, there are some optional passes to improve the compatibility with planners. Both are motivated by POPF2's missing support for negative preconditions. The first of these two passes transforms all locks that have been realized as predicates and literals into functions and numeric facts and assignments. The second pass exchanges each remaining literal in a precondition into a positive literal over an inverse predicate and extends the actions' effects such that inverse literals are updated in accordance with their originals. While activating the latter is mandatory to support POPF2, activating the former brings two additional benefits. First, it simplifies the specification of problem files because less inverse literals have to be specified for the initial state of the problem. Second, it produces better planning results when employing POPF2.

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://projects.ikv.de/qvt/wiki`

6.5 Evaluation of Translation Variants

This section evaluates the performance of a planning system on PDDL domains that have been generated from the application example introduced in Section 6.2. In particular,

- we compare the two options to support forbidden pairs, i.e., translating them into negative existential quantifications or employing counting functions for the incoming and outgoing edges of each node, see Section 6.3.3, and
- we check whether or not the different graph transformation approaches to handle dangling edges, see Section 6.3.4, result in any performance differences.

To gain meaningful performance results from our experiments, we employed the same planning system on each generated domain, regardless of whether it contained quantifications or counting functions. Unfortunately, planning systems that are specifically targeted at temporal planning domains, like POPF2 [Col+10; Col+11] or YAHSP2-MT [Vid11], do not allow negations[3] or quantifications to appear in the precondition of actions[4]. Therefore, we chose to employ the slightly older SGPlan_6 [CWH06; HW08] because it supports both durative actions as well as quantifications in their preconditions.

Experiment Setup For the planning problems associated with the generated domains, we used 10 different problem sizes (5 different railway network sizes with either 3 or 4 RailCabs) and 10 different problem instances per problem size. They consist of 18, 24, 30, 36, or 42 track sections (depending on the railway network size), 2 Y junctions, and specify initial and target track sections for the RailCabs. The initial and target track sections have been generated randomly for each instance (in such a way that the problem is solvable). We used the same random number generator seed to produce both the instances for the domain resulting from the translation variant employing quantifications in the actions' preconditions and the variant employing counting functions. This allows for a fair comparison of these two translation variants.

The railway topology is defined such that the problems are hard to solve by planning systems, i.e., they need to backtrack a lot. This is because the target track sections lie behind a bottleneck. If RailCabs drive into the bottleneck in a wrong order, the goal cannot be achieved anymore. However, the domain provides no predicate that represents the order of RailCabs on the tracks. This complicates the reasoning of planning systems: without recognizing the order of RailCabs, the planner is prone to search along wrong paths in the state space.[5]

[3] POPF2 does allow the negation of static facts, e.g., equality, though.

[4] See `http://www.plg.inf.uc3m.es/ipc2011-deterministic/ParticipatingPlanners` for information on temporal planners that participated in the 7th International Planning Competition, 2011.

[5] We verified this assumption by a comparison with a modified domain. The modified domain included a predicate for the order of RailCabs, additional goal literals typed over this predicate, and action schemata that allow to create such literals.

The experiments were conducted on an Intel Core i7-2600 compute server with 8 (virtual) cores running at 3.40GHz. Each experiment was given 2 cores and 4GB of RAM. If no plan could be computed within 5 minutes, the job was terminated.

Results Table 6.2 shows the means and standard deviations (σ) of the planning times as well as the number of solved problems for the generated problem instances with the smallest and largest railway network size for 3 and 4 RailCabs. There was no significant difference in average plan lengths between the different translation variants.

Table 6.2: Planning times of SGPlan_6 on domains of different translation variants

Variant	*3 RCs, 18 tr.*		*3 RCs, 42 tr.*		*4 RCs, 18 tr.*		*4 RCs, 42 tr.*	
	mean	σ	*mean*	σ	*mean*	σ	*mean*	σ
DPO-Q	24.54	0.39	—	—	71.79	1.04	—	—
SPO-Q	24.78	0.27	—	—	67.99	0.22	—	—
DPO-C	0.17	0.06	2.02	0.36	1.60	0.83	7.95	0.65
SPO-C	0.18	0.06	1.96	0.19	1.65	0.94	7.82	0.61

DPO-Q and SPO-Q denote those translation variants that realize forbidden pairs as negative existential quantifications. DPO-C and SPO-C denote those variants that employ counting functions and realize forbidden pairs as simple checks on the functions' values for the connecting nodes. Comparing the counter variants with the quantification variants (in either DPO or SPO), it can be seen that the counter variants result in much shorter planning times than the quantification variants. When using the domain with the quantification variant, no instance of the 42-track problem size could be solved within the given time.

Figure 6.5 shows the planning times of the four translation variants in a histogram with a logarithmic scale. The values seen in Table 6.2 correspond to the first, fifth, sixth, and tenth cluster of the histogram. The error bars represent the standard deviation of the measurements. A different view on the results is given in Figure 6.6. It shows a line chart illustrating how many problem instances (of any size) could be solved by each variant over time.

Looking at the line chart, we can see that the variants employing counting functions manage to solved more problem instances than the variant containing negative existential quantifications. In sum, they also managed to solve them in less time. From the histogram, it can be seen that this also holds for each individual problem size.

Interestingly, the two translation variants to handle dangling edges performed equally well. To understand why there is no significant difference in their performance, we first look at the DPO variants in isolation and then discuss the differences to the SPO variants.

In the DPO variants, the dangling condition is realized in the same manner as forbidden pairs: either as negative existential quantifications (in DPO-Q) or via

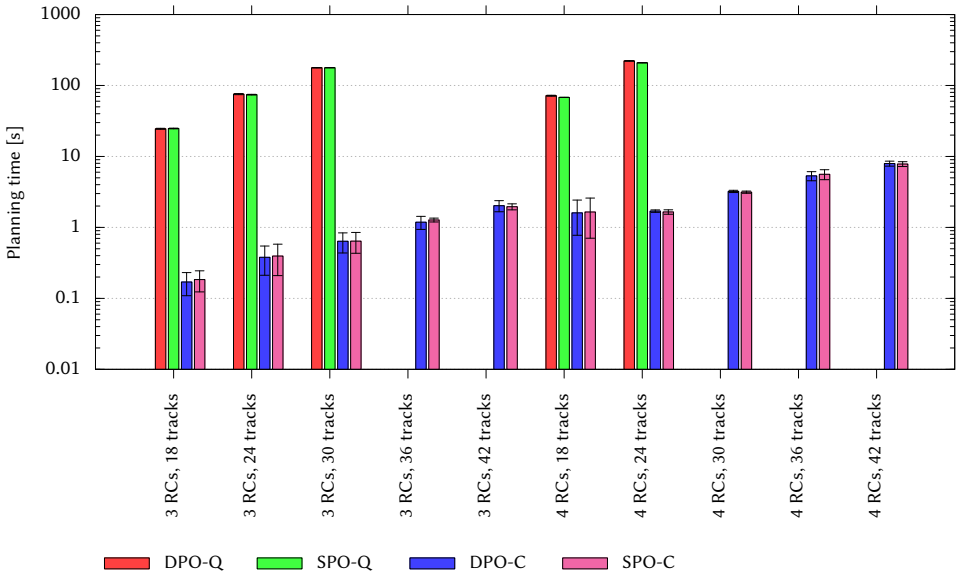


Figure 6.5: Histogram of planning times on domains of different translation variants

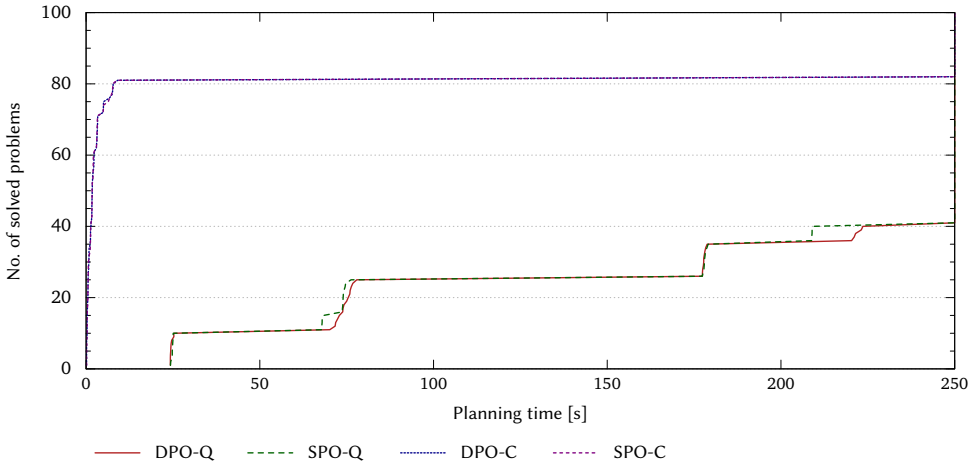


Figure 6.6: Line chart of planning times on domains of different translation variants

counting functions (in DPO-C). Dangling edges can only occur when a rule deletes a node. However, the only rule that does, `dissolveConvoy`, also specifies a forbidden pair with the same edge type as the dangling edge. Therefore, the negative existential quantifications or numeric facts on the countering functions realizing the dangling condition are redundant in the generated domain and do not have an impact on the total planning time.

In the SPO variants, however, dangling edges have to be deleted. In the variant employing quantifications (SPO-Q), this is realized via a quantification in the effect. The quantification is done over nodes of the same type as the node on the second end of the dangling edge and specifies the deletion of all available edges between these nodes. Although this is *not* redundant to any functionality in the generated action schema, it does perform equally well to the DPO-Q domain. The reason behind this is probably that the SPO-Q domain already contains quantifications over the same nodes. For every quantification in the *effect* of a rule, there is – due to a forbidden pair with the same edge type as the dangling edge – already a quantification in the *condition* over the same nodes.

In the SPO variant employing counting functions (SPO-C), an additional action schema is used to optionally delete dangling edges when necessary. The reason why this did not result in any performance losses compared to the DPO-C domain is simple: the additional action was never applied. There was no need to apply the action, because dangling edges do not occur in the RailCab-NACs domain.

6.6 Evaluation of Concurrency and Urgency Rules

The purpose of this section is to validate whether or not the existence of concurrency and urgency rules in a domain causes problems in this domain to be too complex for planning systems to solve. It provides a performance evaluation of a planning system running on PDDL domains that have been generated from a DGTS model of the RailCab domain, which was introduced in Section 5.1. Specifically, it compares the performance of four generated PDDL domains that differ only in whether or not concurrency and urgency rules exist.

Since the translation of concurrency and urgency rules results in required concurrency in the generated PDDL domain, we need to employ a complete temporal planner. For this reason, we chose to employ POPF2 [Col+10; Col+11] on each of the four domains. To convey an idea for how much faster an incomplete temporal planner is compared to a complete one, we also employ SGPlan_6 [CWH06; HW08] on the domain where neither concurrency nor urgency rules exist.

Note that domains without concurrency or urgency rules were not modified in any other way from the original RailCab domain. As a consequence, on domains that come without concurrency rules, the rule `changePublication` can be applied any time, i.e., without the need for the RailCab to move from the track section monitored by the old base station to the track section monitored by the new base station. On domains that come without urgency rules, we still have those variants of `moveRailCab` and `moveConvoy` that accelerate or brake the RailCab and convoy,

respectively. Removed is only the requirement that no time is allowed to pass between the application of a rule accelerating or moving a RailCab and the subsequent application of a rule moving or braking the same RailCab.

All four domains have been generated using the DPO-C translation variant. Since POPF2 does not support negative literals in the precondition of actions, the translation has also been configured to generate inverse predicates for each available predicate in the domain. In doing so, negative literals in a precondition of an action could be substituted by positive literals of their inverse predicates, which allowed POPF2 to support the domain. Note that the domain used by $SGPlan_6$ differs from that of POPF2 in that it does not generate inverse predicates.

Experiment Setup In this evaluation, we used 18 different problem sizes (2 or 3 RailCabs with 2, 4, or 6 consecutive railway line sections per RailCab and 0, 2, or 4 intermediate track sections per line section) and 10 different problem instances per problem size. The problem instances with 2 RailCabs contain 1, 2, and 3 crossroads; those with 3 RailCabs contain 3, 5, and 7 crossroads. As with the last evaluation, the initial and target track sections have been generated randomly for each instance.

The railway topology is defined as a grid. On this topology, the generated planning problems are conceptually easy to solve by planning systems. Bottleneck problems do not occur, because initial and target track sections of all RailCabs lie on different line sections.

The experiments were conducted on the same machine as the experiments in the last section. Here, each experiment was given 2 cores and 4GB or RAM. As with the other evaluation, the job was terminated if no plan could be computed within 5 minutes.

Results Table 6.3 shows the means and standard deviations (σ) of the planning times as well as the number of solved problems for the generated problem instances with the smallest, medium-sized, and largest railway network size for 2 RailCabs and the smallest network size for 3 RailCabs. While some problem instances with the medium-sized network for 3 RailCabs could be solved on domains with concurrency rules, none could be solved if urgency rules were involved. Problem instances with the largest network size for 3 RailCabs could not be solved within the given time and memory limits on any of the four domains. As with the experiments in the last section, there was no significant difference in the average plan lengths.

CR0-UR0 denotes the results of POPF2 on domains where no concurrency or urgency rule exist, CR1-UR0 and CR0-UR1 on domains where only concurrency rules and only urgency rules exist, respectively, and CR1-UR1 on the domain where both kinds of rules exist. Finally, SGPLAN denotes the results of $SGPlan_6$ on domains where neither concurrency nor urgency rules exist. Comparing the different problem sizes and domains, it can be seen that the planning time on domains without urgency rules increases exponentially with the size of the railway network. On domains with urgency rules where solutions were found, the performance is similar to that on domains without urgency rules. However, for the medium-sized and

Table 6.3: Planning times of POPF2 and $SGPlan_6$ on domains with and without concurrency and urgency rules

Domain	*2 RCs, 4 sect.*		*2 RCs, 8 sect.*		*2 RCs, 12 sect.*		*3 RCs, 6 sect.*	
	mean	σ	*mean*	σ	*mean*	σ	*mean*	σ
CR0-UR0	0.10	0.01	1.83	0.16	36.01	13.31	10.13	5.75
CR1-UR0	0.10	0.00	3.47	0.13	34.90	0.29	4.01	2.34
CR0-UR1	0.12	0.03	—	—	—	—	23.63	14.77
CR1-UR1	0.10	0.00	—	—	—	—	3.43	0.63
SGPLAN	0.20	0.00	0.16	0.01	—	—	0.53	0.47

largest networks with urgency rules, no solutions were found within the given time. Interestingly, in case of the problem size with 3 RailCabs, it seems as if problems were easier to solve on domains where concurrency rules exist than on domains without concurrency rules. However, this observation does not persist when looking at all problem sizes.

Figure 6.7 shows the planning times in a histogram with a logarithmic scale. The values seen in Table 6.3 correspond to the first, fifth, ninth, and tenth cluster of the histogram. A line chart illustrating how many problem instances of each domain could be solved over time is given in Figure 6.8.

From the line chart, it can be seen that domains containing urgency rules solved significantly less problem instances than other domains. However, the problem instances that could be solved were solved in very short time. The reason for this can be seen by looking at the histogram: on domains containing urgency rules, only problem instances with a very small railway network could be solved.

Whether or not concurrency rules exist in the domain does not make much of a difference in performance. One of the reasons for this might be that both the demanding and satisfying rules of concurrency rule `allowChangePublication` have to be applied anyway to reach the goal, and scheduling applications of `changePublication` to a specific point in time does rarely conflict with any other rule application.

In the case of urgency rules, the situation is different. The urgency rule `immediateMoveRailCab` affects the application times of rules that change the positions of RailCabs or convoys. However, scheduling applications of such rules to a specific point in time is likely to conflict with other rule applications. The reason for this is that other rule applications might rely on certain positions of RailCabs, e.g., to initiate a convoy operation or simply to prevent conflicts in concurrent execution.

$SGPlan_6$ could solve most of the problem instances on the domain without any concurrency or urgency rules and – thanks to being an incomplete temporal planner – could solve them very fast. However, it could not solve as much problems as POPF2. On all problem instances that $SGPlan_6$ could not solve, it suddenly terminated without giving a reason as to why.

In Section 3.4 we mentioned that POPF2 is based on CRIKEY3, which is able to recognize durative actions as compression-safe, i.e., it detects where end points

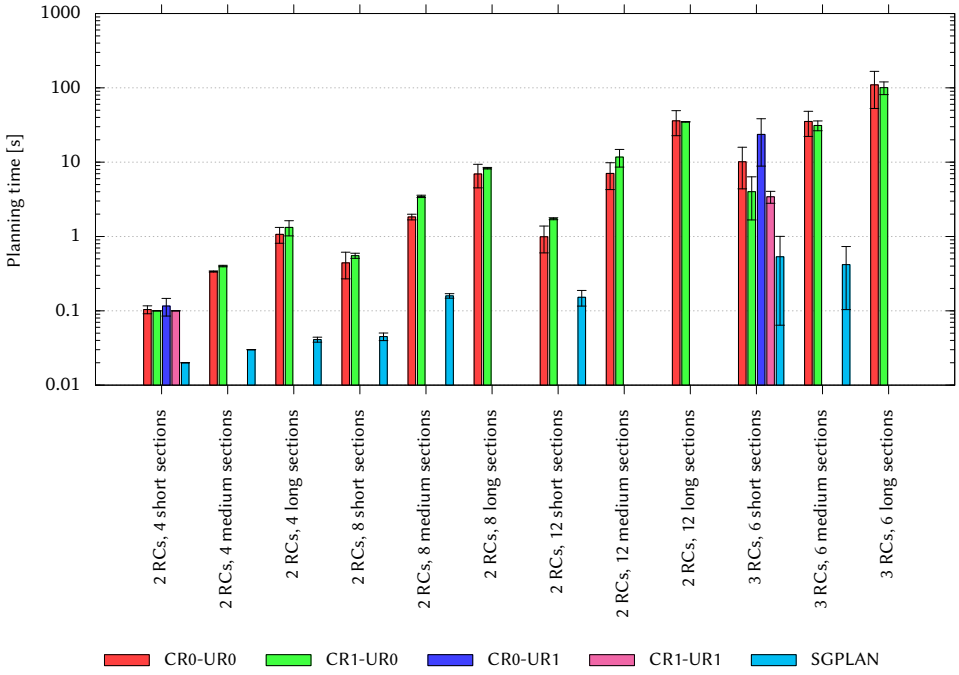


Figure 6.7: Histogram of planning times on domains with and without concurrency and urgency rules

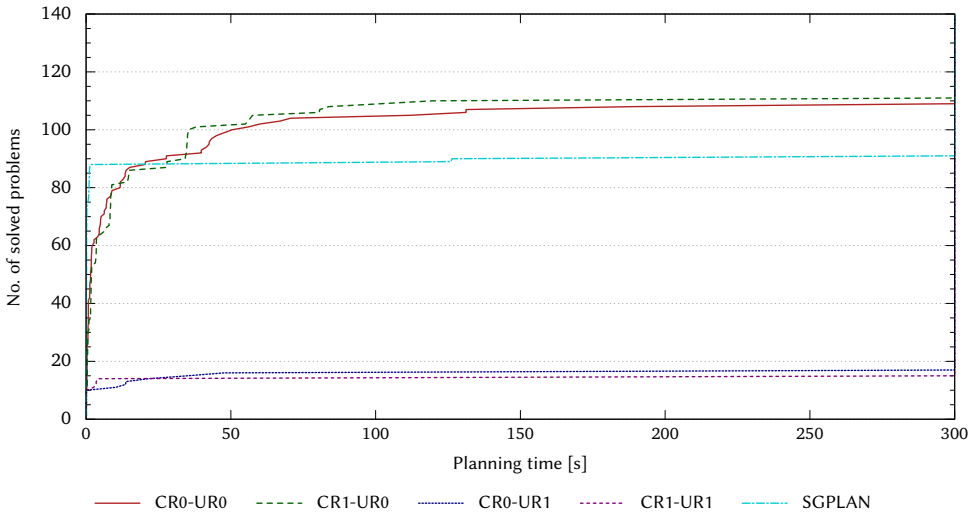


Figure 6.8: Line chart of planning times on domains with and without concurrency and urgency rules

of durative actions do not have to be considered a choice point for state space exploration. Since there is no required concurrency in the CR0-UR0 domain, all actions in this domain are compression-safe; however, they are not recognized as such by POPF2. Unfortunately, the technique employed by POPF2 recognizing actions as compression-safe is sound but not complete, cf. [HLF04].

To reach as much compression-safety in POPF2 as possible, we changed certain characteristics of the generated domain. Implementing these changes into the CR0-UR0 domain led to POPF2 recognizing 100% of all ground actions as compression-safe. On the other three domains, the percentage of recognized compression-safety depended on the problem instance used for evaluation.

Numeric facts The generated domain contains numeric facts for checking the number of read locks and, in case of domains employing counting functions, for checking the number of adjacent edges of a node. In each of these numeric facts, its generated rvalue is the minimal possible value. For this reason, we replaced each numeric fact implemented via an equality check with a numeric fact using a non-strict inequality, i.e., the lvalue is required to be less than or equal to the rvalue. While not changing the domain's behavior, this allowed POPF2 to realize that lower values are preferable to higher values and thus that an increment operation at the beginning of an action cannot enable the application of other actions.

Deletion and increment operations According to the translation scheme, changes specified in a durative graph transformation rule are realized in the `at_end` effect of an action. Since all relevant nodes and edges are locked at the beginning and unlocked at the end of an action's application, certain changes, e.g., the deletion of elements and increment operations on counters, can be realized in the `at_start` effect instead. Implementing this made POPF2 realize that there is no benefit in postponing the application of an action's `at_end` effect, because all "bad" effects have already been executed with the `at_start` effect.

Figures 6.9 and 6.10 show the histogram and line chart on the domains with increased compression-safety, respectively. Whereas taking concurrency rules into account made almost no difference on domains with less compression-safety, there is a significant difference now. More frequently than before, disregarding concurrency rules is now faster than considering them. This can be seen clearly on compression-safe domains without urgency rules (CR0-UR0 vs. CR1-UR0). When comparing the results on compression-safe domains with urgency rules (CR0-UR1 vs. CR1-UR1) to the previous results of Figures 6.7 and 6.8, we can see that POPF2 now found more solutions on the domain disregarding concurrency rules than before and thus also more than on the compression-safe domain that considers concurrency rules. Unlike the previous results, these results conform to what we expected: both kinds of rules make planning harder, but urgency rules more so than concurrency rules.

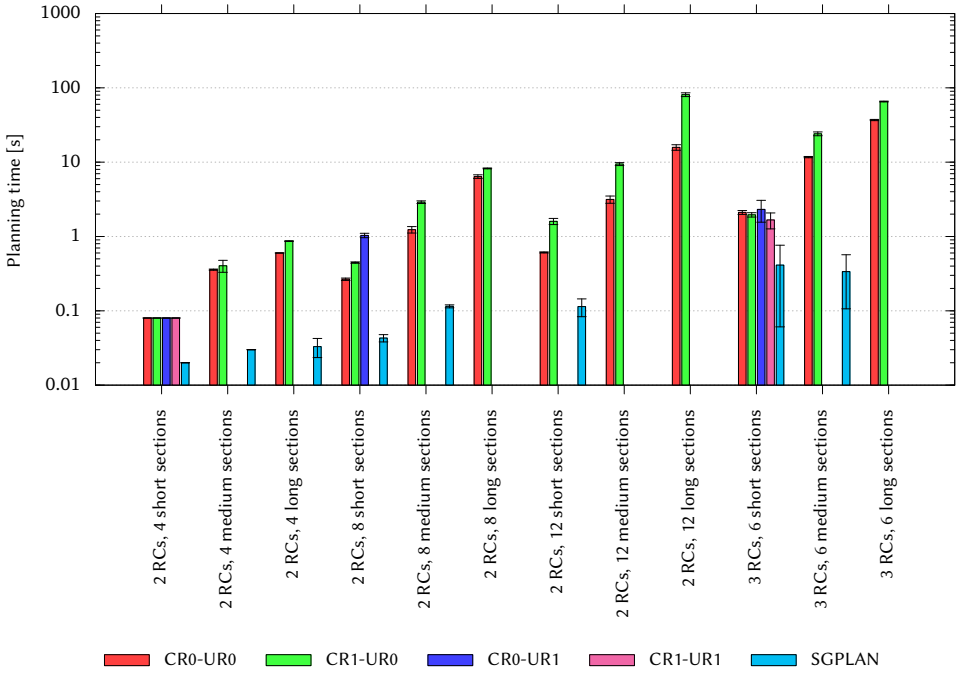


Figure 6.9: Histogram of planning times on domains with and without concurrency and urgency rules, with as much compression-safety as possible

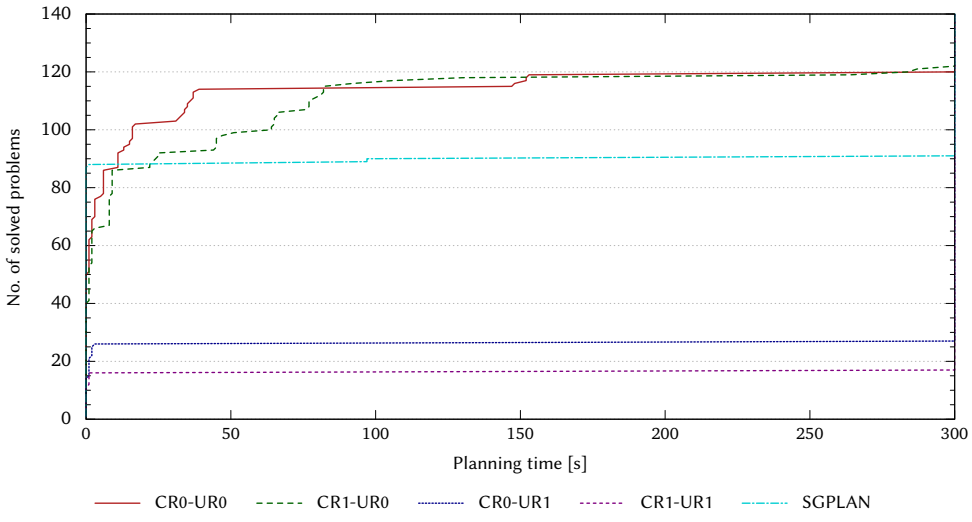


Figure 6.10: Line chart of planning times on domains with and without concurrency and urgency rules, with as much compression-safety as possible

6.7 Related Work

While planning and scheduling is a discipline in artificial intelligence research that has made many advances in the last decades, only few moves have been made to tie planning techniques with the software engineering domain. The benefit in pursuing this direction lies in the ability to reuse models which have been built by software engineers for planning applications. For example, a recent approach by Hoffmann et al. [HWK12] reuses an existing Status and Action Management (SAM) model, which describes status changes of SAP Business Objects, for the creation of new processes in Business Process Management. The processes are created by translating the SAM model into PDDL and employing an adapted version of the planning system FF.

Other promising approaches that combine software engineering models with planning techniques rely on graph transformation systems as an underlying formalism due to its suitability for implementing software architecture reconfiguration. An early attempt in this direction came from Edelkamp and Rensink [ER07]. They showed manual translations from planning tasks specified with graph transformation rules into PDDL and identified some advantages of planning directly on graph transformation systems: the possibility to reduce the state space by representing isomorphic graphs only once and the support for the instantiation and deinstantiation of nodes. These advantages gave reason for techniques that directly use graph transformation systems for planning, notably the ones already discussed in Section 4.5.

Tichy and Klöpper [TK11] were first to present an automatic translation of graph transformation rules into PDDL actions. As noted at the beginning of this chapter, their translation scheme supports time-consuming reconfigurations by means of temporal annotations, but they did not treat potential conflicts in the concurrent application of rules or explicit concurrency and urgency. Meijer [Mei12] also provides a translation between graph transformations and PDDL, but does not cover time or durative actions. The developed translator works in both directions, i.e., planning tasks formulated in PDDL can also be translated back into a graph transformation system. As opposed to our technique, the main focus of this work lies on the backward direction. The employed graph transformation tool, however, needs to support existential quantification on edges to match the semantics of PDDL[6].

Although not reusing an existing software engineering model, there are approaches to knowledge engineering for planning domains that make use of UML or have an object-centric perspective, e.g., the itSIMPLE project by Vaquero et al. [Vaq+09] or the GIPO environment by Simpson et al. [SKM07]. The itSIMPLE project's goal is to develop a knowledge engineering environment that supports its users during the design of planning applications. The tool allows to define UML domains with OCL constraints, which can be translated into Petri net representations for analyses and mapped into PDDL for planning. It also supports durative actions

[6] In PDDL, a literal that is going to be deleted by an action does not have to be present if it is not required in the precondition. In such a case, the action is still applicable, but does not change the literal. In fact, we used this property for the deletion of dangling edges in the SPO-Q translation variant.

via a timing diagram that specifies the timing of a predicate change. Since version 4.0 [Vaq+12], it also allows to express the timing of conditions. The GIPO environment has a similar goal as itSIMPLE. However, its static validation techniques are rather simple, e.g., it checks that declared predicates are actually used in action schemata or finds predicates that are only used in preconditions but never in effects or vice versa. On the plus side, it provides a manual plan stepper, which helps in discovering bugs within a domain definition, and supports hierarchical task network planning [Sac75].

6.8 Discussion

Several approaches to improve the planning performance on the domains used in this thesis are conceivable. Leaving improvements to the internal workings of temporal planning systems aside, we can augment the planning domain with additional information or simplify it by employing abstraction techniques. The remainder of this section presents three ideas in this direction.

The first idea is motivated by the fact that many domains involve multiple agents that have to cooperate to reach their goals. For example, RailCabs each have an own target track section but cooperate when driving there by building convoys or by deciding who may drive first before a bottleneck. The idea is to decompose the planning problem into several parts that differ in whether agents can solve these partial problems individually or have to coordinate with one another in solving them. Based on this premise, Hauck [Hau13] developed a prototype of such a decompositional planning system as part of his master's thesis. This system uses so-called *domain patterns* to identify those points in individual plans where coordination starts or ends to be necessary or beneficial. These domain patterns are provided as a third input file, i.e., along with the domain and problem file, to the planning system. An adaptation of these domains patterns to graph transformation systems is easily feasible.

The second idea is motivated by the fact that planning systems employing action compression, like SGPlan_6, are much faster than complete planning systems, like POPF2. The idea is a smart combination of two such different planning systems. A prototype realizing this combination was developed by Heil [Hei14] as part of his bachelor thesis. In a first step, the need for required concurrency is abstracted away by manipulating the planning domain. Then, the planning problem is solved on this abstract domain by an action-compressing planner. Although the resulting plan is not valid on the original domain, it provides valuable information for solving the original problem. Based on this information, the original domain is manipulated again such that the complete planner benefits from this additional information when solving the problem in a second pass. This is done in such a way that the domain modification ensures the validity of the plan on the original domain.

The third idea is an approach to real-time planning[7], where the generation and execution of plans is performed concurrently. In the operational phase of a system

[7] Real-time planning is also known as online planning.

continuously sensing its environment and adapting to changes therein, one cannot afford a lengthy planning process. For this reason, real-time planning approaches commit to certain state changes before a complete plan has been found, e.g., by restricting their search to states in the neighborhood of their current state. Real-time planning has its roots in real-time heuristic search. While the well-known real-time search algorithm *Learning Real-Time A* (LRTA*)* [Kor90], which is essentially a variant of A* [HNR68] with early commitment and limited lookahead, has been adapted by Bonet et al. [BLG97] to solve STRIPS planning tasks, we are not aware of any adaptations of real-time search techniques to temporal planning.

The idea proposed in the following is inspired by *Windowed Hierarchical Cooperative A* (WHCA*)* [Sil05]. WHCA* is a real-time search algorithm for the cooperative pathfinding problem, i.e., multiple agents have to move in a labyrinth of discrete cells to different target cells without two agents crashing or occupying the same cell at the same time. Each movement to a neighboring cell takes exactly on time unit. The word "cooperative" in WHCA* refers to employing a centralized planning process for all agents; "hierarchical" refers to using two levels of abstraction, i.e., one concrete level, which denotes the search in the planning problem's state space, and one abstract level, which denotes the search performed by the heuristic function employed in the concrete level; and "windowed" refers to a k-step window used as planning horizon. Planning in the abstract level is fast because it uses a much simpler heuristic function than the concrete level, ignores positions of (and thus conflicts with) other agents, and allows to reuse calculations in later invocations. The suitability for real-time planning is given by the k-step window. It is realized by a concrete-level state space for k steps and an abstract-level state space for the remainder of the search.

In its abstract-level state space, WHCA* abstracts away all other agents lying on the path to an agent's goal location. A straightforward adaptation of this approach to generic temporal planning would require that it is always clear what has to be abstracted away and how. Since this is not the case in general, a better idea is a generic relaxation of all action applications in the abstract-level state space, i.e., all actions after a certain point in time k. In the RailCab domains, a simple delete relaxation causes – from the perspective of one RailCab – all other RailCabs to be abstracted away as soon as they perform one move operation. While different in concept from WHCA*, such a relaxation amounts to a similar behavior on appropriate domains.

The use of a concrete and abstract level and the delete relaxation can easily be implemented directly into the planning domain. For each original action schema, we can create a relaxed action schema where delete effects are ignored and numerical facts and assignments are executed on bound variables, i.e., increment operations on upper bounds and decrement operations on lower bounds, instead of their original variables. With the beginning of executing abstract actions, bound variables have the same value as their original counterparts. This can be achieved by adding increment and decrement operations for bound variables to original action schemata for each numeric assignment they contain. The transition from executing concrete actions to

executing abstract actions can be implemented with a special one-shot action whose execution takes k time units and enables the use of abstract actions in its `at_end` effect. In case of the RailCab domain, a side benefit of using a k-time-units-sized window is the option to decompose the planning problem into subproblems based on the RailCabs' locations in each planning iteration.

7 Conclusion and Future Work

This thesis investigated planning techniques working with graph transformation systems as system models. Motivated by different requirements arising from two fundamentally different application examples, two approaches for graph transformation planning have been developed.

The first approach preserves the expressiveness of graph transformation systems by directly working on a graph transformation system's state space. As a result, it can handle system models with an infinite state space. It employs a domain-independent heuristic function that uses the solution length of a relaxed planning problem as heuristic estimate. Taking both the structure of graphs and applicable graph transformations into account, this is a considerable improvement over related work.

The second approach puts its focus on timing aspects and concurrency. It comes with a new formalism for the specification of durative graph transformations, which

- guarantees that multiple durative graph transformations with conflicting behavior cannot be executed concurrently,
- enables the explicit, rule-based specification of required concurrency and urgency, and
- enables making use of available verification procedures by being based on timed graph transformation systems.

System models that have been designed in this formalism can be translated into planning domains, for which problem instances can be solved by employing off-the-shelf PDDL-based planning systems. Evaluation results gave insight on how to decide between different translation variants and showed how certain aspects of planning domains with required concurrency influenced planning performance.

Future work can be divided into future work extending the performance and capabilities of the two approaches presented in this thesis as well as future work related to their integration with other methods or languages. Obvious perspectives

for improving the performance concern the development of planning systems. Section 4.6 already discussed the idea of adapting techniques for the recognition of landmarks, i.e., literals occurring in every possible plan, from PDDL-based planning systems to graph transformation planning. Ideas for improving the performance of temporal planning systems have already been presented in Section 6.8. The support for forbidden patterns has been integrated into planning methods working directly on the state space of a graph transformation system, cf. Section 4.5, but not into approaches translating them into plan constraints of PDDL 3.0.

In the following, we first consider the integration of the planning methods proposed in this thesis with the MDSD approach MechatronicUML. Then, we cover options for extending the DGTS formalism with new concepts and automation.

Integration into MDSD The thesis introduction mentioned the OCM as an architectural model for the development of self-organizing systems and classified the generation of temporal plans into its top layer, the cognitive operator. Executing plan actions was classified into the middle layer, the reflective operator. The question is how the execution of temporal plans computed by the cognitive operator can be integrated into the reflective operator.

For the RailCab system introduced in Section 1.4, whose software is developed in MechatronicUML, executing actions of a plan means triggering specific transitions of RTSCs via asynchronous messages. Technically, this can be done by translating the plan into a RTSC that schedules when to send which message. To make this possible, the system model has to include information about which plan actions trigger which transitions in which RTSCs.

Instead of assigning exactly one transition to each action, it might be reasonable to trigger more than one transition (from different RTSCs) at the same time for certain actions. An example is the action `joinConvoy`, whose execution involves the instantiation of multiple RTCPs: the RTCP ConvoyCoordination for communicating with the coordinator of the convoy and the RTCP DistanceControl for communicating with a neighboring member. Similarly, it might be reasonable for certain actions to trigger no transition at all or to make this conditional on the action executed beforehand. For example, when an application of the action `moveRailCab` follows another application of the same action, no change in behavior is necessary.

If a centralized planning process is used for multiple agents, the task of assigning transitions to actions is yet a little more complicated. Since some actions might involve multiple agents but other actions involve only certain agents, the plan has to be assigned to different agents on a per-action basis. However, we do not know the acting agents at design time. To be able to perform this assignment task automatically at runtime, the acting agents need to be identifiable either from the plan or from additional information included in the system model.

Semi-Automatic Definition of Constraint Morphisms Concurrency and urgency rules enable the explicit specification of requirements regarding the concurrent or urgent execution of durative graph transformations, respectively. Specifying these

rules involves defining several constraint morphisms for the purpose of connecting demanding and satisfying rules at compatible matches. Instead of specifying each constraint morphism manually, it is also viable to search for them automatically. Since an automatic search for constraint morphisms is performed on the level of rules, i.e., in design time, this does not increase any planning time.

Unfortunately, a fully automatic specification of constraints in concurrency and urgency rules is less expressive than a manual one. Essentially, it causes all technically possible constraint morphisms to be included in the concurrency or urgency rule. This is the reason why we did not propose this idea as a general solution in Sections 5.5 and 5.6.

A better idea is to search for *potential* matches automatically and *choose* which of them to include as constraint morphism in the concurrency or urgency rule manually. Consider the concurrency rule `allowChangePublication` as an example. Its satisfier interface consists of a `RailCab` node and two `Track` nodes. Searching for satisfier constraint morphisms from this interface to LHSs of durative graph transformation rules leads to a lot of matches, some of them legitimate and some of them not, e.g., where images of `Track` nodes are interchanged compared to the legitimate matches. The latter kind of matches can simply be discarded by a developer when choosing which of the matches found to include into `allowChangePublication` as a satisfier constraint morphism. This approach retains the expressiveness of concurrency and urgency rules but still offers a more convenient means of specification.

To prevent certain illegitimate matches from being found in the first place, the search for matches could also consider both LHS and RHS of each durative rule instead of its LHS or RHS only. This comes down to a morphism based on graph transformation rules instead of graphs. Such inter-rule morphisms can be defined component-wise, i.e., mapping LHSs to LHSs and RHSs to RHSs, such that they preserve relations between LHSs and RHSs. In the example of concurrency rule `allowChangePublication`, we could thus specify that there has to be an `on` edge being deleted adjacent to the left `Track` node. In doing so, a significant amount of illegitimate matches can be avoided. Note that there are two ways NACs can be employed in this approach: first, as a component whose relation to an LHS is to be preserved by inter-rule morphisms; and second, as a NAC *for* inter-rule morphisms, i.e., forbidding the existence of elements in those rules acting as hosts for the inter-rule morphisms.

Mutex Rules There are situations where durative graph transformation rules are forbidden to be executed concurrently. In many cases, such a mutual exclusion of durative graph transformation rules is ensured by the locking mechanism implemented into their semantics. However, in some application examples, such an implicit prevention of forbidden concurrent execution might not be enough.

Consider for example a job shop scheduling problem [Gra66], i.e., n jobs have to be scheduled on m machines, where executing a job on a machine is implemented as a durative rule consuming the job and producing some output. Since the application of such a rule does not change the machine in any way, this machine gets a read

lock but no write lock. As a result, multiple jobs can be scheduled to be executed on the same machine at the same time. To prevent this from happening, we need some kind of *mutex rule*, i.e., a means to explicitly specify mutual exclusion. The idea of mutex rules is thus inverse to that of concurrency rules.

Both the DGTS formalism and the translation scheme compiling DGTS models into PDDL could be extended to support such mutex rules. Unlike concurrency and urgency rules, mutex rules are symmetric, i.e., there is no distinction between demanding and satisfying rules. Nevertheless, they can be specified similarly, i.e., by defining constraint morphisms for each participating durative rule, and realized via indicators similar to demand and satisfaction indicators.

Bibliography

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. "Model-checking in dense real-time". In: *Information and Computation* 104.1 (1993), pp. 2–34.

[AD94] Rajeev Alur and David L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235.

[Ahm12] Amir Shayan Ahmadian. "Exploiting Planning Graphs and Landmarks for Efficient GTS Planning". Master's thesis. University of Paderborn, Nov. 2012.

[AO95] Wil M. P. van der Aalst and Michiel A. Odijk. "Analysis of railway stations by means of interval timed coloured Petri nets". In: *Real-Time Systems* 9.3 (Nov. 1995), pp. 241–263.

[AUT14] AUTOSAR Release Management. *Release 4.2 Overview and Revision History – AUTOSAR Release 4.2.1*. Tech. rep. 000:AUTOSAR_Release4.2_Overview_RevHistory. AUTOSAR Development Partnership, Oct. 2014.

[Bal+08] Paolo Baldan, Andrea Corradini, Fernando Luís Dotti, Luciana Foss, Fabio Gadducci, and Leila Ribeiro. "Towards a notion of transaction in graph rewriting". In: *5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. Ed. by Roberto Bruni and Dániel Varró. Vol. 211. Electronic Notes in Theoretical Computer Science (ENTCS). Amsterdam, The Netherlands: Elsevier, Apr. 2008, pp. 39–50.

[BD91] Bernard Berthomieu and Michel Diaz. "Modeling and verification of time dependent systems using time Petri nets". In: *IEEE Transactions on Software Engineering* 17.3 (1991), pp. 259–273.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. "A tutorial on Uppaal". In: *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*. Ed. by Marco Bernardo and Flavio Corradini. Vol. 3185. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2004, pp. 200–236.

[Bec+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. "Symbolic invariant verification for systems with dynamic structural adaptation". In: *28th International Conference on Software Engineering (ICSE 2006)*. New York, NY, USA: ACM, May 2006, pp. 72–81.

[Bec+12] Steffen Becker, Christian Brenner, Christopher Brink, Stefan Dziwok, Renate Löffler, Christian Heinzemann, Uwe Pohlmann, Wilhelm Schäfer, Julian Suck, and Oliver Sudmann. *The MechatronicUML Design Method – Process, Syntax, and Semantics*. Tech. rep. tr-ri-12-326. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Aug. 2012.

[BF97] Avrim L. Blum and Merrick L. Furst. "Fast planning through planning graph analysis". In: *Artificial Intelligence* 90.1–2 (Feb. 1997), pp. 281–300.

[BH02] Luciano Baresi and Reiko Heckel. "Tutorial introduction to graph transformation: a software engineering perspective". In: *1st International Conference on Graph Transformation (ICGT 2002)*. Berlin Heidelberg: Springer-Verlag, 2002, pp. 402–429.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA ,USA: Addison-Wesley Publishing Company, 1987. ISBN: 978-0201107159.

[BLG97] Blai Bonet, Gábor Loerincs, and Héctor Geffner. "A robust and fast action selection mechanism for planning". In: *14rd National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI 1997)*. Menlo Park, CA, USA: AAAI Press, 1997, pp. 714–719.

[BÖ10] Artur Boronat and Peter Csaba Ölveczky. "Formal real-time model transformations in MOMENT2". In: *13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*. Ed. by David S. Rosenblum and Gabriele Taentzer. Vol. 6013. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2010, pp. 29–43.

[Bon+07] Iovka Boneva, Frank Hermann, Harmen Kastenberg, and Arend Rensink. "Simulating multigraph transformations using simple graphs". In: *6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Ed. by Juan de Lara and Dániel Varró. Vol. 6. Electronic Communications of the EASST (ECEASST). European Association of Software Science and Technology (EASST), 2007.

[Bra+04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. "A survey of self-management in dynamic software architecture specification". In: *1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*. New York, NY, USA: ACM, 2004, pp. 28–33.

[BST99] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. "Modeling urgency in timed systems". In: *Compositionality: The Significant Difference (COMPOS 1997)*. Ed. by Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli. Vol. 1536. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 1999, pp. 103–129.

[BT04] Roberto Barbuti and Luca Tesei. "Timed automata with urgent transitions". In: *Acta Informatica* 40.5 (Mar. 2004), pp. 317–347.

[BY04] Johan Bengtsson and Wang Yi. "Timed automata: semantics, algorithms and tools". In: *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2004, pp. 87–124.

[Byl94] Tom Bylander. "The computational complexity of propositional STRIPS planning". In: *Artificial Intelligence* 69.1–2 (1994), pp. 165–204.

[CFR09] Andrea Corradini, Luciana Foss, and Leila Ribeiro. "Graph transformation with dependencies for the specification of interactive systems". In: *Recent Trends in Algebraic Development Techniques: 19th International Workshop on Algebraic Development Techniques (WADT 2008)*. Ed. by Andrea Corradini and Ugo Montanari. Vol. 5486. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2009, pp. 102–118.

[Cla+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2007. ISBN: 978-3540719403.

[Col+08] Andrew Ian Coles, Maria Fox, Derek Long, and Amanda Jane Smith. "Planning with problems requiring temporal coordination". In: *23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*. Menlo Park, CA, USA: AAAI Press, 2008, pp. 892–897.

[Col+09a] Amanda Jane Coles, Andrew Ian Coles, Maria Fox, and Derek Long. "Extending the use of inference in temporal planning as forwards search". In: *19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. 2009.

[Col+09b] Andrew Ian Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Jane Smith. "Managing concurrency in temporal planning using planner-scheduler interaction". In: *Artificial Intelligence* 173.1 (Jan. 2009), pp. 1–44.

[Col+10] Amanda Jane Coles, Andrew Ian Coles, Maria Fox, and Derek Long. "Forward-chaining partial-order planning". In: *20th International Conference on Automated Planning and Scheduling (ICAPS 2010)*. May 2010.

[Col+11] Amanda Jane Coles, Andrew Ian Coles, Allan Clark, and Stephen T. Gilmore. "Cost-sensitive concurrent planning under duration uncertainty for service level agreements". In: *21th International Conference on Automated Planning and Scheduling (ICAPS 2011)*. June 2011.

[Cor+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. "Algebraic approaches to graph transformation – part I: basic concepts and double pushout approach". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 3, pp. 163–245.

[Cou90] Bruno Courcelle. "The monadic second-order logic of graphs. i. recognizable sets of finite graphs". In: *Information and Computation* 85.1 (1990), pp. 12–75.

[Cou97] Bruno Courcelle. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 5, pp. 313–400.

[CS07] Andrew Ian Coles and Amanda Jane Smith. "Marvin: a heuristic search planner with online macro-action learning". In: *Journal of Artificial Intelligence Research (JAIR)* 28.1 (Feb. 2007), pp. 119–156.

[Cus+07] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. "When is temporal planning really temporal?" In: *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2007, pp. 1852–1859.

[CWH06] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. "Temporal planning using subgoal partitioning and resolution in SGPlan". In: *Journal of Artificial Intelligence Research (JAIR)* 26.1 (May 2006), pp. 323–369.

[Det+12] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin, and Stephan Hildebrandt. *Story Diagrams – Syntax and Semantics*. Tech. rep. tr-ri-12-324. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012.

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. "Hyperedge replacement graph grammars". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 2, pp. 95–162.

[Eck+13] Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, and Wilhelm Schäfer. "Modeling and verifying dynamic communication structures based on graph transformations". In: *Computer Science – Research and Development (CSRD)* 28.1 (Feb. 2013), pp. 3–22.

[Ede03] Stefan Edelkamp. "Taming numbers and durations in the model checking integrated planning system". In: *Journal of Artificial Intelligence Research (JAIR)* 20.1 (2003), pp. 195–238.

[Ehr+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Berlin Heidelberg: Springer-Verlag, 2006. ISBN: 978-3540311874.

[Ehr+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. "Algebraic approaches to graph transformation – part II: single pushout approach and comparison with double pushout approach". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 4, pp. 247–312.

[EHR97] Andrzej Ehrenfeucht, Tero Harju, and Grzegorz Rozenberg. "2-structures – a framework for decomposition and transformation of graphs". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 6, pp. 401–478.

[EJL06] Stefan Edelkamp, Shahid Jabbar, and Alberto Lluch Lafuente. "Heuristic search for the analysis of graph transition systems". In: *3rd International Conference on Graph Transformation (ICGT 2006)*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Vol. 4178. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Sept. 2006, pp. 414–429.

[EKL91] Hartmut Ehrig, Martin Korff, and Michael Löwe. "Tutorial tntroduction to the algebraic approach of graph grammars based on double and single pushouts". In: *4th International Workshop on Graph Grammars and Their Application to Computer Science*. Ed. by Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 532. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 1991, pp. 24–37.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. "Graph-grammars: an algebraic approach". In: *14th Annual IEEE Symposium on Switching and Automata Theory (SWAT 1973)*. 1973, pp. 167–180.

[ER07] Stefan Edelkamp and Arend Rensink. "Graph transformation and ai planning". In: *2nd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2007)*. Sept. 2007.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. “Node replacement graph grammars”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 1, pp. 1–94.

[Est10] H.-Christian Estler. “Heuristic Search-Based Planning for Graph Transformation Systems”. Diploma thesis. University of Paderborn, Oct. 2010.

[EW11] H.-Christian Estler and Heike Wehrheim. “Heuristic search-based planning for graph transformation systems”. In: *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011)*. 2011, pp. 54–61.

[Fed71] Jerome Feder. “Plex languages”. In: *Information Sciences* 3.3 (July 1971), pp. 225–241.

[FL03] Maria Fox and Derek Long. “PDDL2.1: an extension to PDDL for expressing temporal planning domains”. In: *Journal of Artificial Intelligence Research (JAIR)* 20.1 (Dec. 2003), pp. 61–124.

[FN71] Richard E. Fikes and Nils J. Nilsson. “Strips: a new approach to the application of theorem proving to problem solving”. In: *2nd International Joint Conference on Artificial Intelligence (IJCAI 1971)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1971, pp. 608–620.

[Gau+14] Jürgen Gausemeier, Franz Josef Rammig, Wilhelm Schäfer, and Walter (Eds.) Sextro. *Dependability of Self-optimizing Mechatronic Systems*. Lecture Notes in Mechanical Engineering (LNME). Berlin Heidelberg: Springer-Verlag, 2014. ISBN: 978-3642537417.

[GHV02] Szilvia Gyapay, Reiko Heckel, and Dániel Varró. “Graph transformation with time: causality and logical clocks”. In: *1st International Conference on Graph Transformation (ICGT 2002)*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 2505. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Oct. 2002, pp. 120–134.

[GL05] Alfonso Gerevini and Derek Long. *Plan Constraints and Preferences in PDDL3 – The Language of the Fifth International Planning Competition*. Tech. rep. Department of Electronics for Automation, University of Brescia, Aug. 2005.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. “Self-organising software architectures for distributed systems”. In: *1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS 2002)*. New York, NY, USA: ACM, 2002, pp. 33–38.

[GN92] Naresh Gupta and Dana S. Nau. “On the complexity of blocks-world planning”. In: *Artificial Intelligence* 56.2–3 (Aug. 1992), pp. 223–254.

[Gra66] R. L. Graham. “Bounds for certain multiprocessing anomalies”. In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[GRS14] Jürgen Gausemeier, Franz Josef Rammig, and Wilhelm (Eds.) Schäfer. *Design Methodology for Intelligent Technical Systems: Develop Intelligent Technical Systems of the Future*. Lecture Notes in Mechanical Engineering (LNME). Berlin Heidelberg: Springer-Verlag, 2014. ISBN: 978-3642454349.

[Hau13] Thomas Hauck. "Multiagenten-Koordination in temporaler Planung". Master's thesis. University of Paderborn, Jan. 2013.

[HE10] Christian Heinzemann and Tobias Eckardt. "Reachability analysis on timed graph transformation systems". In: *4th International Workshop on Graph-Based Tools (GraBaTs 2010)*. Ed. by Juan de Lara and Dániel Varró. Vol. 32. Electronic Communications of the EASST (ECEASST). European Association of Software Science and Technology (EASST), 2010.

[Hec+99] Reiko Heckel, Gregor Engels, Hartmut Ehrig, and Gabriele Taentzer. "Classification and comparison of module concepts for graph transformation systems". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1999. Chap. 17, pp. 639–689.

[Hei14] Johannes Heil. "Ein hierarchischer Ansatz für temporale Planung bei erforderlicher Nebenläufigkeit". Bachelor thesis. University of Paderborn, Nov. 2014.

[Hel14] Malte Helmert. "On the complexity of planning in transportation domains". In: *6th European Conference on Planning (ECP 2001)*. Ed. by Amedeo Cesta and Daniel Borrajo. Palo Alto, CA, USA: AAAI Press, Aug. 2014, pp. 349–360.

[HH11] Christian Heinzemann and Stefan Henkler. *Timed Story Driven Modeling*. Tech. rep. tr-ri-11-326. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2011.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. "Graph grammars with negative application conditions". In: *Fundamenta Informaticae* 26.3–4 (Dec. 1996), pp. 287–313.

[HHV11] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. "Towards guided trajectory exploration of graph transformation systems". In: *4th International Workshop on Petri Nets and Graph Transformation (PNGT 2010)*. Ed. by Claudia Ermel and Kathrin Hoffmann. Vol. 40. Electronic Communications of the EASST (ECEASST). European Association of Software Science and Technology (EASST), 2011.

[Hir08] Martin Hirsch. "Modell-basierte Verifikation von vernetzten mechatronischen Systemen". PhD thesis. University of Paderborn, July 2008.

[HLF03] Keith Halsey, Derek Long, and Maria Fox. "Isolating where planning and scheduling interact". In: *22nd UK Planning and Scheduling Special Interest Group (PlanSIG 2003)*. 2003, pp. 104–114.

[HLF04] Keith Halsey, Derek Long, and Maria Fox. "Crikey – a planner looking at the integration of scheduling and planning". In: *ICAPS Workshop on Integrating Scheduling Into Planning (WIPIS 2004)*. 2004, pp. 46–52.

[HN01] Jörg Hoffmann and Bernhard Nebel. "The FF planning system: fast plan generation through heuristic search". In: *Journal of Artificial Intelligence Research (JAIR)* 14.1 (Jan. 2001), pp. 253–302.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107.

[Hoa78] Charles Antony Richard Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677.

[HOG04] Thorsten Hestermeyer, Oliver Oberschelp, and Holger Giese. "Structured information processing for self-optimizing mechatronic systems". In: *1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004)*. 2004.

[HPS04] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. "Ordered landmarks in planning". In: *Journal of Artificial Intelligence Research (JAIR)* 22.1 (2004), pp. 215–278.

[HW08] Chih-Wei Hsu and Benjamin W. Wah. "The SGPlan planning system in IPC-6". In: *6th International Planning Competition (IPC 2008)*. 2008.

[HWK12] Jörg Hoffmann, Ingo Weber, and Frank Michael Kraft. "SAP speaks PDDL: exploiting a software-engineering model for planning in business process management". In: *Journal of Artificial Intelligence Research (JAIR)* 44.1 (May 2012), pp. 587–632.

[ISO02] ISO/IEC JTC 1/SC 22. *Information technology – Z formal specification notation – Syntax, type system and semantics*. Tech. rep. 13568:2002. International Organization for Standardization (ISO) & International Electrotechnical Commission (IEC), July 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. 2nd ed. Cambridge, MA, USA: MIT Press, 2006. ISBN: 978-0262017152.

[JR80] Dirk Janssens and Grzegorz Rozenberg. "On the structure of node-label-controlled graph languages". In: *Information Science* 20.3 (Apr. 1980), pp. 191–216.

[Klö+10] Benjamin Klöpper, Shinichi Honiden, Jan Meyer, and Matthias Tichy. "Planning with utility and state trajectory constraints in self-healing automotive systems". In: *4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2010, pp. 74–83.

[KM07] Jeff Kramer and Jeff Magee. "Self-managed systems: an architectural challenge". In: *Workshop on the Future of Software Engineering (FOSE 2007)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2007, pp. 259–268.

[Kor90] Richard E. Korf. "Real-time heuristic search". In: *Artificial Intelligence* 42.2–3 (Mar. 1990), pp. 189–211.

[Kov12] Daniel L. Kovacs. "A multi-agent extension of pddl3.1". In: *3rd ICAPS Workshop on the International Planning Competition (WIPC 2012)*. 2012, pp. 19–27.

[KR06] Harmen Kastenberg and Arend Rensink. "Model checking dynamic states in GROOVE". In: *13th International Workshop on Software Model Checking (SPIN 2006)*. Ed. by Antti Valmari. Vol. 3925. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2006, pp. 299–305.

[KRH10] Emil Keyder, Silvia Richter, and Malte Helmert. "Sound and complete landmarks for and/or graphs". In: *19th European Conference on Artificial Intelligence (ECAI 2010)*. Amsterdam, The Netherlands: IOS Press, 2010, pp. 335–340.

[Lar+10] Juan de Lara, Esther Guerra, Artur Boronat, Reiko Heckel, and Paolo Torrini. "Graph transformation for domain-specific discrete event time simulation". In: *5th International Conference on Graph Transformation (ICGT 2010)*. Berlin Heidelberg: Springer-Verlag, 2010, pp. 266–281.

[Lar+14] Juan de Lara, Esther Guerra, Artur Boronat, Reiko Heckel, and Paolo Torrini. "Domain-specific discrete event modelling and simulation using graph transformation". In: *Software and Systems Modeling* 13.1 (Feb. 2014), pp. 209–238.

[Le 98] Daniel Le Métayer. "Describing software architecture styles using graph grammars". In: *IEEE Transactions on Software Engineering* 24.7 (July 1998), pp. 521–533.

[LE08] Leen Lambers and Hartmut Ehrig. "Efficient conflict detection in graph transformation systems by essential critical pairs". In: *5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. Ed. by Roberto Bruni and Dániel Varró. Vol. 211. Electronic Notes in Theoretical Computer Science (ENTCS). Amsterdam, The Netherlands: Elsevier, Apr. 2008, pp. 17–26.

[LEO06] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. "Conflict detection for graph transformation with negative application conditions". In: *3rd International Conference on Graph Transformation (ICGT 2006)*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Vol. 4178. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Sept. 2006, pp. 61–76.

[LF03] Derek Long and Maria Fox. "Exploiting a graphplan framework in temporal planning". In: *13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*. Menlo Park, CA, USA: AAAI Press, June 2003, pp. 52–61.

[LHL01] Joachim Lückel, Thorsten Hestermeyer, and Xiaobo Liu-Henke. "Generalization of the cascade principle in view of a structured form of mechatronic systems". In: *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2001)*. 2001, pp. 123–128.

[LO04] Marisa Llorens and Javier Oliver. "Structural and dynamic changes in concurrepetnt systems: reconfigurable Petri nets". In: *IEEE Transactions on Computers* 53.9 (Sept. 2004), pp. 1147–1158.

[Löw93] Michael Löwe. "Algebraic approach to single-pushout graph transformation". In: *Theoretical Computer Science* 109.1–2 (1993), pp. 181–224.

[MA98] Drew McDermott and AIPS-98 Planning Competition Committee. *PDDL – The Planning Domain Definition Language*. Tech. rep. CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control, Oct. 1998.

[McC94] Brenan J. McCarragher. "Petri net modelling for robotic assembly and trajectory planning". In: *IEEE Transactions on Industrial Electronics* 41.6 (Dec. 1994), pp. 631–640.

[Mei12] Ronald Meijer. "PDDL planning problems and GROOVE graph transformations: combining two worlds with a translator". In: *17th Twente Student Conference on IT*. University of Twente, June 2012.

[Mer74] Philip Meir Merlin. "A Study of the Recoverability of Computer Systems". PhD thesis. University of California, Irvine, CA, USA, 1974.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 1980. ISBN: 978-3540102359.

[MSO11] Eliseo Marzal, Laura Sebastia, and Eva Onaindia. "Full extraction of landmarks on propositional planning tasks". In: *12th International Conference on Artificial Intelligence Around Man and Beyond (AI*IA 2011)*. Ed. by Roberto Pirrone and Filippo Sorbello. Vol. 6934. Lecture Notes in Artificial Intelligence (LNAI). Berlin Heidelberg: Springer-Verlag, 2011, pp. 383–388.

[Neu07] Stefan Neumann. "Modellierung und Verifikation von zeitbehafteten Graphtransformationssystemen mittels GROOVE". Diploma thesis. University of Paderborn, Sept. 2007.

[ÖM07] Peter Csaba Ölveczky and José Meseguer. "Semantics and pragmatics of Real-Time Maude". In: *Higher-Order and Symbolic Computation* 20.1–2 (June 2007), pp. 161–196.

[OMG09] OMG. *Unified Modeling Language (UML) Superstructure Specification – Version 2.2*. Tech. rep. formal/2009-02-02. Object Management Group, Inc., Feb. 2009.

[OMG11] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.1*. Tech. rep. formal/2011-01-01. Object Management Group, Inc., Jan. 2011.

[Pav72] Theodosios Pavlidis. "Linear and context-free graph grammars". In: *Journal of the ACM (JACM)* 19.1 (Jan. 1972), pp. 11–22.

[PER95] Julia Padberg, Hartmut Ehrig, and Leila Ribeiro. "Algebraic high-level net transformation systems". In: *Mathematical Structures in Computer Science (MSCS)* 5.2 (1995), pp. 217–256.

[Pet62] Carl Adam Petri. "Kommunikation mit Automaten". PhD thesis. Darmstadt University of Technology, 1962.

[Plu05] Detlef Plump. "Confluence of graph transformation revisited". In: *Processes, Terms and Cycles*. Ed. by Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer. Vol. 3838. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2005, pp. 280–308.

[Plu93] Detlef Plump. "Hypergraph rewriting: critical pairs and undecidability of confluence". In: *Term Graph Rewriting: Theory and Practice*. Ed. by M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen. Chichester, UK: John Wiley & Sons Ltd., 1993. Chap. 15, pp. 201–213.

[PSH14] Julie Porteous, Laura Sebastia, and Jörg Hoffmann. "On the extraction, ordering, and usage of landmarks in planning". In: *6th European Conference on Planning (ECP 2001)*. Ed. by Amedeo Cesta and Daniel Borrajo. Palo Alto, CA, USA: AAAI Press, Aug. 2014, pp. 174–182.

[Ram73] Chander Ramchandani. "Analysis of Asynchronous Concurrent Systems by Petri Nets". PhD thesis. Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, 1973.

[RDV10] José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. "On the behavioral semantics of real-time domain specific visual languages". In: *8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010)*. Ed. by Peter Csaba Ölveczky. Vol. 6381. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2010, pp. 174–190.

[Ren04] Arend Rensink. "The GROOVE Simulator: a tool for state space generation". In: *2nd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2003)*. Vol. 3062. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2004, pp. 479–485.

[Ren08] Arend Rensink. "Explicit state model checking for graph grammars". In: *Concurrency, Graphs and Models*. Ed. by Pierpaolo Degano, Rocco De Nicola, and José Meseguer. Vol. 5065. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2008, pp. 114–132.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003. ISBN: 978-0137903955.

[Röh09] Malte Röhs. "Sichere Konfigurationsplanung adaptiver Systeme durch Model Checking". Diploma thesis. University of Paderborn, Sept. 2009.

[RR04] Carsten Rust and Franz Josef Rammig. "A Petri net approach for the design of dynamically modifiable embedded systems". In: *Design Methods and Applications for Distributed Embedded Systems: 18th IFIP World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2004)*. Ed. by Bernd Kleinjohann, Guang R. Gao, Hermann Kopetz, Lisa Kleinjohann, and Achim Rettberg. Vol. 150. IFIP International Federation for Information Processing. Berlin Heidelberg: Springer-Verlag, Aug. 2004, pp. 257–266.

[RVV09] José Eduardo Rivera, Christina Vicente-Chicote, and Antonio Vallecillo. "Extending visual modeling languages with timed behavior specifications". In: *Conferencia Iberoamericana de Software Engineering (CIbSE 2009)*. 2009, pp. 87–100.

[RW10a] Silvia Richter and Matthias Westphal. "The LAMA planner: guiding cost-based anytime planning with landmarks". In: *Journal of Artificial Intelligence Research (JAIR)* 39.1 (2010), pp. 127–177.

[RW10b] Malte Röhs and Heike Wehrheim. "Sichere Konfigurationsplanung selbst-adaptierender Systeme durch Model Checking". In: *9. Paderborner Workshop Entwurf mechatronischer Systeme*. Ed. by J. Gausemeier, F. Rammig, W. Schäfer, and A. Trächtler. Vol. 272. HNI-Verlagsschriftenreihe. Heinz Nixdorf Institut, 2010, pp. 253–265.

[RZ13] Christoph Rasche and Steffen Ziegert. "Multilevel planning for self-optimizing mechatronic systems". In: *5th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2013)*. (Valencia, Spain). Wilmington, DE, USA: Xpert Publishing Services, May 2013.

[Sac75] Earl D. Sacerdoti. "The nonlinear nature of plans". In: *4th International Joint Conference on Artificial Intelligence (IJCAI 1975)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1975, pp. 206–214.

[San10] Scott Sanner. "Relational Dynamic Influence Diagram Language (RDDL): Language Description". Unpublished. 2010.

[Sch97] Andy Schürr. "Programmed graph replacement systems". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Ed. by Grzegorz Rozenberg. River Edge, NJ, USA: World Scientific Publishing, 1997. Chap. 7, pp. 479–546.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. Chichester, West Sussex, England: Wiley, 1998. ISBN: 978-0471982326.

[SHS11] Julian Suck, Christian Heinzemann, and Wilhelm Schäfer. *Formalizing Model Checking on Timed Graph Transformation Systems*. Tech. rep. tr-ri-11-316. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Sept. 2011.

[Sil05] David Silver. "Cooperative pathfinding". In: *1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005)*. Menlo Park, CA, USA: AAAI Press, 2005, pp. 117–122.

[SKM07] Ron M. Simpson, Diane E. Kitchin, and T. L. McCluskey. "Planning domain definition using GIPO". In: *Knowledge Engineering Review* 22.2 (June 2007), pp. 117–134.

[Sni11] Erik Snippe. "Using heuristic search to solve planning problems in GROOVE". In: *14th Twente Student Conference on IT*. University of Twente, Jan. 2011.

[Spi92] John Michael Spivey. *The Z Notation: A Reference Manual*. 2nd ed. International Series in Computer Science. Upper Saddle River, NJ, USA: Prentice Hall, 1992. ISBN: 978-0139785290.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. 1st ed. Chichester, West Sussex, England: Wiley, 2006. ISBN: 978-0470025703.

[SV08] Eugene Syriani and Hans Vangheluwe. "Programmed graph rewriting with time for simulation-based design". In: *Theory and Practice of Model Transformations: 1st International Conference on Model Transformation (ICMT 2008)*. Ed. by Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio. Vol. 5063. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2008, pp. 91–106.

[SV11] Eugene Syriani and Hans Vangheluwe. "A modular timed graph transformation language for simulation-based design". In: *Software and Systems Modeling (SoSyM)* 12.2 (2011), pp. 387–414.

[TGM00] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. "Dynamic change management by distributed graph transformation: towards configurable distributed systems". In: *6th International Workshop on Theory and Application of Graph Transformations (TAGT 1998)*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 1764. Berlin Heidelberg: Springer-Verlag, 2000, pp. 179–193.

[TK11] Matthias Tichy and Benjamin Klöpper. "Planning self-adaptation with graph transformations". In: *International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2011)*. Ed. by Andy Schürr, Dániel Varró, and Gergely Varró. Vol. 7233. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2011.

[Vaq+09] Tiago Stegun Vaquero, José Reinaldo Silva, Marcelo Ferreira, Flavio Tonidandel, and J. Christopher Beck. "From requirements and analysis to PDDL in itSIMPLE3.0". In: *19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*. 2009.

[Vaq+12] Tiago Stegun Vaquero, Rosimarci Tonaco, Gustavo Costa, Flavio Tonidandel, José Reinalda Silva, and J. Christopher Beck. "itSIMPLE4.0: enhancing the modeling experience of planning problems". In: *ICAPS 2012 System Demonstrations and Exhibits*. (Atibaia, São Paulo, Brazil). 2012, pp. 11–14.

[Var+06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. "Termination analysis of model transformations by Petri nets". In: *3rd International Conference on Graph Transformation (ICGT 2006)*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Vol. 4178. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Sept. 2006, pp. 260–274.

[Var12] Szilvia Varró-Gyapay. "Optimization in graph transformation systems with time using Petri net based techniques". In: *5th International Workshop on Petri Nets, Graph Transformation and other Concurrency Formalisms (PNGT 2012)*. Ed. by Julia Padberg and Kathrin Hoffmann. Vol. 51. Electronic Communications of the EASST (ECEASST). European Association of Software Science and Technology (EASST), 2012.

[VFC95] Valentín Valero Ruiz, David de Frutos Escrig, and Fernando Cuartero Gómez. "Timed processes of timed Petri nets". In: *16th International Conference on Application and Theory of Petri Nets*. Ed. by Giorgio De Michelis and Michel Diaz. Vol. 935. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 1995, pp. 490–509.

[Vid11] Vincent Vidal. "YAHSP2: keep it simple, stupid". In: *7th International Planning Competition (IPC 2011)*. 2011, pp. 83–90.

[VV06] Szilvia Varró-Gyapay and Dániel Varró. "Optimization in graph transformation systems using Petri net based techniques". In: *2nd International Workshop on Petri Nets and Graph Transformation (PNGT 2006)*. Ed. by Paolo Baldan, Hartmut Ehrig, Julia Padberg, and Grzegorz Rozenberg. Vol. 2. Electronic Communications of the EASST (ECEASST). European Association of Software Science and Technology (EASST), 2006.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. "A graph transformation approach to software architecture reconfiguration". In: *Science of Computer Programming* 44.2 (Aug. 2002), pp. 133–155.

[WF99] Michel Wermelinger and José Luiz Fiadeiro. "Algebraic software architecture reconfiguration". In: *7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1999)*. Ed. by Oscar Nierstrasz

and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Sept. 1999, pp. 393–409.

[Wim+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauderand Andy Schürr, and Dennis Wagelaar. "A comparison of rule inheritance in model-to-model transformation languages". In: *Theory and Practice of Model Transformations: 4th International Conference on Model Transformation (ICMT 2011)*. Ed. by Jordi Cabot and Eelco Visser. Vol. 6707. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, 2011, pp. 31–64.

[YL04] Håkan L. S. Younes and Michael L. Littman. *PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects*. Tech. rep. CMU-CS-04-167. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2004.

[YS03] Håkan L. S. Younes and Reid G. Simmons. "VHPOP: versatile heuristic partial order planner". In: *Journal of Artificial Intelligence Research (JAIR)* 20.1 (2003), pp. 405–430.

[Zei84] Bernard Phillip Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Waltham, MA, USA: Academic Press, 1984. ISBN: 978-0127784502.

[ZG03] Lin Zhu and Robert Givan. "Landmark extraction via planning graph propagation". In: *ICAPS 2003 Doctorial Consortium*. 2003.

[ZH13a] Steffen Ziegert and Christian Heinzemann. *Durative Graph Transformation Rules*. Tech. rep. tr-ri-13-329. Heinz Nixdorf Institute, University of Paderborn, Mar. 2013.

[ZH13b] Steffen Ziegert and Christian Heinzemann. "Durative graph transformation rules for modelling real-time reconfiguration". In: *10th International Colloquium on Theoretical Aspects of Computing (ICTAC 2013)*. (Shanghai, China). Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8049. Lecture Notes in Computer Science (LNCS). Berlin Heidelberg: Springer-Verlag, Sept. 2013, pp. 427–444.

[Zha89] Weixiong Zhang. "Representation of assembly and automatic robot planning by Petri net". In: *IEEE Transactions on Systems, Man, and Cybernetics* 19.2 (1989), pp. 418–422.

[Zie14] Steffen Ziegert. "Graph transformation planning via abstraction". In: *3rd ETAPS Workshop on Graph Inspection and Traversal Engineering (GRAPHITE 2014)*. (Grenoble, France). Ed. by Dragan Bošnački, Stefan Edelkamp, Alberto Lluch Lafuente, and Anton Wijs. Vol. 159. Electronic Proceedings in Theoretical Computer Science (EPTCS). July 2014, pp. 71–83.

[ZW13] Steffen Ziegert and Heike Wehrheim. "Temporal reconfiguration plans for self-adaptive systems". In: *Software Engineering (SE 2013)*. (Aachen, Germany). Ed. by S. Kowalewski and B. Rumpe. Lecture Notes in Informatics (LNI). Bonn: Gesellschaft für Informatik e.V. (GI), Feb. 2013.

[ZW15] Steffen Ziegert and Heike Wehrheim. “Temporal plans for software architecture reconfiguration”. In: *Computer Science – Research and Development (CSRD)* 30.3–4 (Aug. 2015), pp. 303–320.