Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems

vorgelegt von
Diplom-Informatiker
Manuel Koch
aus Berlin-Spandau

Vom Fachbereich 13 - Informatik -
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften
- Dr. Ing. -

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr. K. Obermeyer
Berichter:	Prof. Dr. H. Ehrig
Berichter:	Prof. Dr. H.-J. Kreowski
Berichter:	Prof. Dr. F. Parisi-Presicce

Tag der wissenschaftlichen Aussprache: 30. September 1999

Berlin 2000
D83

To my grandfather

Zusammenfassung

Die vorliegende Arbeit stellt eine formale Spezifikationsmethode für verteilte Systeme vor. Ziele der Arbeit sind die komponentenweise Modellierung des Verhaltens verteilter Systeme, die Spezifikation von Eigenschaften verteilter Systeme und die Unterstützung zum (automatischen) Überprüfen dieser Eigenschaften.

Um diese Ziele zu erreichen, schlage ich einen integrierten Ansatz zweier formaler Methoden vor: *verteilte Graphtransformation* und *temporale Logik*. Verteilte Graphtransformation wird zur Modellierung des Verhaltens verteilter Systeme eingesetzt, Eigenschaften werden mit temporaler Logik spezifiziert. Der kombinierte Ansatz profitiert von den Stärken der einzelnen Ansätze und stellt eine geeignete Methode zur Verfügung, Aspekte verteilter Systeme formal und graphisch zu modellieren.

Ich habe einen Single-Pushout (SPO) Ansatz für verteilte Graphtransformation entwickelt, der sowohl die Topologie als auch die lokalen Datenzustände spezifiziert. Dieser Ansatz läßt eine Vielzahl von Spezifikationstechniken für die lokalen Zustände zu. Ein Hauptbeitrag meines Ansatzes ist die kompositionale operationale Semantik, welche auf den von Martin Große-Rhode entwickelten *Transformationssystemen* basiert.

Ich integriere den SPO Ansatz für verteilte Graphtransformation und eine aussagenlogische temporale Logik. Die Integration vollzieht sich über eine Pre-Ordnung (eine reflexive und transitive Relation) über verteilten Graphen und resultiert in graphinterpretierten temporalen Formeln und temporalen Graphmodellen. Graphinterpretierte Formeln erlauben die graphische Spezifikation temporaler Eigenschaften, so daß die Spezifikation des verteilten Systems und die Spezifikation der Systemeigenschaften im selben Formalismus geschieht.

Dank der Integration verteilter Graphtransformation und temporaler Logik können vorhandene Konzepte im Bereich der temporalen Logik zum Nachweis graphinterpretierter Formeln bzgl. eines mit verteilter Graphtransformation modellierten Systems genutzt werden. Insbesondere das Konzept des Model-Checkens wird in dieser Arbeit zur automatischen Verifikation von graphinterpretierten Formeln betrachtet. Ich stelle die Konstruktion eines *typischen Modells* vor, welche ein möglicherweise unendliches temporales Graphmodell (d.h. ein Modell mit unendlich vielen Zuständen) zu einem endlichen Graphmodell zusammenfaßt. Das typische Modell hilft, von lokaler Erfüllbarkeit lokaler Formeln in einem Teilsystem des verteilten Systems auf die globale Erfüllbarkeit globaler Formeln im gesamten verteilten System zu schließen.

Abstract

This work provides a methodology for the specification of distributed systems based on formal methods. Major aims are the compositional modeling of the behavior of distributed systems, the specification of distributed system properties and the support of their (automatic) verification.

To reach this goal, I propose an integrated approach of two formal methods, namely *distributed graph transformation* and *temporal logic*. Distributed graph transformation is intended for modeling the behavior of distributed systems, whereas distributed system properties are specified by temporal logic. The combined approach benefits from the strength of each single approach and provides a suitable methodology to model formally the aspects of distributed systems.

I develop a single-pushout approach to distributed graph transformation that specifies the topology of the distributed system as well as its local data states. My approach permits a variety of specification techniques for local data states, where I investigate the requirements for the specification techniques that can be used for the local data. A main contribution of my approach to distributed graph transformation is the compositional operational semantics. This semantic is based on transformation systems introduced by Große-Rhode.

I integrate the single-pushout approach to distributed graph transformation and a propositional temporal logic. The integration takes place over a pre-order relation over distributed graphs and results in graph-interpreted temporal formulas and temporal graph models. Graph-interpreted formulas provide a methodology to specify temporal properties graphically, in the same way as the distributed system itself is specified.

The integration of distributed graph transformation and temporal logic allows to make use of the variety of verification concepts in the area of temporal logic to check graph-interpreted temporal formulas with respect to a temporal graph model. I focus in this work especially on the concept of model-checking for the automatic verification of graph-interpreted temporal formulas. I provide the construction of a typical graph model that collapses a possibly infinite temporal model to a finite one and that supports the reasoning from the local satisfaction of local formulas in a sub-system of the distributed system to global satisfaction of global formulas in the complete distributed system.

Preface

Writing a PhD Thesis let you fight face to face for several years against an (apparent) unreachable, uncatchable and untangible ghost. That the ghost materialized finally and that I am standing at the end one step higher on the potestal than the ghost was possible only due to the motivating following wind blown by many people.
First of all, and most important, my family, that gave me the best imaginable environment and all the conditions to concentrate myself on writing the Thesis. Without them, this work would have been never possible (also from a non biological point of view).
On the way of writing the Thesis, from the start to the end, Reiko Heckel was a patient and motivating companion that gave me constructive answers and help to many questions I was confronted with on this way. Thanks go also to Silke and Max that had to bear me on all the visits around the world. Beside Reiko, main victims of my questions were Fabio Gadducci and Martin Große-Rhode. I am greatful to have met all of them and that I got the chance to learn of them (even if I still not understand why dry cakes should be more tasty). Moreover, all of them motivated me for the trip to Italy confronting me with many interesting experiences. At this point, I thank Ivano Salvo for his patient teaching in the Italian language and culture.
Thanks go to all members of the research groups at the TU Berlin and the Friedrich Alexander Universität Erlangen-Nürnberg where I spent my working time. I am greatful to my advisor Prof. Hartmut Ehrig and the graduate colleague "Kommunikationsbasierte Systeme" leaded by Prof. Hommel, that gave me the chance to write this Thesis. I thank also Prof. Hans-Jörg Kreowski and Prof. Francesco Parisi-Presicce that were willing to report on this thesis. Special thanks to members of this two groups go to Reiko and Martin (I know that I have mentioned them already above, but it is worth to mention them twice), Ingrid Fischer (for teaching me the usefulness of an email communication and for occasional food support) and Michael Berthold (not a member of these groups, but an important member of my research activities), Rosi Bardohl (and Bernie for introducing me to interesting topics beside the thesis), Michael Rudolf (for interesting discussions around Italian cinemas and a more innovating way of using an office), Mercè Llabrés (who gave me surprising ethnological hints to small islands) and Alfio Martini (one minute of his life can tell incredible stories) .
To take my mind occasionally off the PhD and to refresh my power I have to thank many people, summarized in the following groups: the old friends from school, my Judo team, the Stenzel brothers, Paule and Benni.

Contents

Chapter 1

Introduction

In the beginnings of modern computers in 1945, computers were huge and expensive. Institutions had only few computers, working independent of each other. In the middle of the 80s this situation changed. Computers became smaller and cheaper due to two developments: on the one hand powerful microprocessors, on the other hand the start of local area networks (LAN). These technologies enable relatively simple to construct computer systems consisting of a high number of processors connected via high-speed nets: the birth of *distributed systems*. Compared with central systems, distributed systems have a number of advantages: they are more economical, faster, are more suitable for inherent distributed applications, are more reliable, and are easy expandable. These advantages made sure that our daily life is confronted with a growing number of distributed systems.

But the more the society depends on the new technology, the more it feels its weaknesses. Again and again failures happen, that can have a variety of reasons: faulty software, overloaded communication nets as well as overloaded or failed servers. Something of that kind will occur more and more frequently, if the nets for banks, schools, companies, etc. will become bigger and bigger. With luck a failure costs only time and the nerves of the user. If a cash dispenser is defect, mostly one can find one near that operates. The crash of the computer system of a stock market causes already more consequences. The American computer stock market NASDAQ in 1994 or the stock market in Frankfurt in 1998 are only two examples. In 1990 a failure of the telephone net of AT&T resulted in 60000 telephone connections that could not be used anymore and long distance calls were not possible at all. At the latest, if the health or even the life of people is concerned by a faulty system, for instance in a hospital or a plane crash, failures cannot be tolerated. So, it is no wonder, that there is a growing economical and security interest in developing protection against the threat of faulty distributed systems.

To improve the correctness of distributed systems, an ongoing effort is done to develop methods that support the specification and verification of distributed systems. Examples of modeling and specification techniques used for distributed systems are process algebras like CCS [Mil89] or the π-calculus [MPW92], Petri nets [Pet62,Rei87], temporal logics [Sti92,Eme90], graph transformation [EKMR99,Roz97,Tae96], object-oriented modeling techniques like OMT [RBP$^+$91], OOD [Boo94] and UML [Rat97] and may more. The specification techniques address the specification of a distributed system from different points of view: Some of them like process algebras or temporal logics have a more formal orientation, providing mathematical models for the analysis and verification of systems. Others like UML focus more on the design aspects of a distributed system. And some techniques like graph transformations try to provide both a theoretical basis suitable for the analysis and the verification as well as an intuitive way to design a system. I discuss now briefly aspects that have to be covered by a specification technique for distributed systems.

1.1 Requirements to a Specification Technique for Distributed Systems

The following list shows the requirements to a specification technique for distributed systems. These requirements help also to motivate the integration of graph transformation and temporal logic done in this work and sketches the target group of my approach.

Structural Aspects: These aspects are concerned with the description of a distributed system's state. In particular, this includes the current topology of the distributed system, i.e. its sites, processes, components, and their connections. This aspect is expressed preferably graphically. Local states of sites, processes, and components together with their relations via shared data form the second part of a distributed system state.

Transformational Aspects: A distributed system is an evolving, dynamic structure. Evolution takes place in each part of the system. Topological reconfiguration, i.e. the creation and removal of sites, processes, and components as well as connection changes are a crucial property of systems incorporated in an evolving environment. Besides the necessary state transformations of the processes resp. local components themselves, communication and interconnection aspects, as replication, migration, and remote interactions, have to be described as well.

Compositional Aspects: A distributed system consists of several interacting components. Therefore, a notion of a component and their composition has to be defined in the specification technique in order to support the composition of distributed systems by their components.

Analysis Aspects: Whereas analysis techniques for non-distributed systems, preferably tool-supported, are already a highly desirable feature and focus of research in the last decades, the natural increased complexity of distributed systems make such feature even more desirable in the area of distributed systems.

None of the specification techniques for distributed systems mentioned above satisfies all of the aspects completely. For instance, process algebras are well suited to model compositional and analysis aspects, but are less suitable for structural aspects, especially the local data aspects of a process. In contrast, UML supports nicely the development of static system structures and the description of sequential and concurrent behavior. However, verification and analysis techniques are rarely due to a less formal basis. I will discuss in more detail in Chapter 6 how far existing specification techniques for distributed systems satisfy the aspects when I relate my integrated approach of graph transformation and temporal logic with existing specification techniques for distributed systems. I have chosen a combination of graph transformation and temporal logic for the specification of a distributed system, since graph transformation supports the statical as well as the transformational aspects in an intuitive, graphical and formal way and the research in temporal logic provides us with various analysis techniques and tools to specify and check properties of a system.

1.1.1 Distributed Graph Transformation

The algebraic approach of graph transformation has been developed in the early seventies by Ehrig, Pfender and Schneider as a generalization of Chomsky grammars and term rewriting

[EPS73,Roz97]. Graph grammars provide an intuitive description for the manipulation of graphs and graphical structures as they occur in programming language semantics, data bases, operating systems, rule based systems and various kinds of software and, last but not least, distributed systems. Mainly, the graph transformation models for distributed systems concentrate on topological aspects of distributed systems [DM87,JR91,KLG91,MPR99]. In [Tae96], Taentzer combined structured graph transformation on two levels. In this way, allocation of objects and tasks, object replication and migration, remote interactions, multiple threads of control and dynamic network topologies can be specified in a rule-based and graphical way.

1.1.2 Temporal Logic

Temporal Logic is a special type of *modal logic* [Sti92,Eme90]. Modal logic was originally developed by philosophers to study different models of truth. Whereas an assertion can be true in one "world" , it could be false in another one. In temporal logic the truth values of assertions change over time. Various temporal operators are provided to describe and reason about how truth values of assertions vary with time. It was Pnueli in the late seventies [Pnu77] who argued that temporal logic could be a useful formalism for specifying and verifying correctness of computer programs. Temporal logic is especially suitable for *reactive systems*, i.e. systems, which do not just compute a function from a given input, but which continuously and in general nonterminating interact with an environment. Starting from this basic idea, a variety of temporal logics have been developed: Linear time, branching time, real time, logics with and without past operators, logics referring to intervals rather than moments of time etc. Verification techniques for checking temporal formulas with respect to a temporal model are provided, often supported by a tool.

1.2 Main Results

I present now the main results of this work, that are divided into *conceptual* and *theoretical* results.

1.2.1 Conceptual results

1.2.1.1 Modeling Distributed Systems by Distributed Graph Transformation

It suggests itself to describe the topology of a distributed system graphically. This approach occurs in many existing approaches to graph transformations for modeling distributed systems [MPR99,Tae96,Sch93,DM87] and also my approach follows the idea of a graphical representation of the topology. I follow in particular [MPR99] where a distributed system consists of concurrent processes communicating over shared ports. The different process and port types that can occur in an actual distributed system are modeled in a *type graph* [CELP96].

To go along with [Tae96] both the topological as well as the local level of a distributed system are integrated in one structure. In this structure the allocation of data and tasks to processes is explicitly modeled. Whereas [Tae96] treats only plain graphs for the local level, my approach is more abstract in the sense that a specification technique most suitable for modeling local states of processes can be chosen. However, the designated specification technique has to fit into the framework of category theory [BW95] and has to be cocomplete. To treat the specification technique used for the specification of local states more abstractly allows to reuse existing specification techniques. The reuse of specification techniques lies in practical interest, in particular when existing specification tools and new tools have to be integrated into an existing environment [Löw98]. All actions in the distributed system are described rule-based and include, for instance, the evolution of the topology, the transformation of local states, the replication of data and their migration or the communication over shared data.

In my approach, the distributed system is specified componentwise by modeling the local behavior for processes separately for each process type that may occur in the distributed system. The local behavior of a process of a certain type is specified by a *process grammar* for this type containing a set of *process productions* that describe intuitively and graphically the actions a process of this type may perform. The interaction between the processes is specified by a *synchronization relation*, that determines which actions of a process have to be synchronized with actions of other processes. This componentwise local specification of a distributed system is different to the global specification of a distributed system introduced in [Tae96].

The difference between a local specification of processes and the global specification is also reflected in the operational semantics of the distributed system. I provide a compositional operational distributed semantics, that is composed from the operational semantics of the single processes according to the synchronization relation. The operational semantics of a single process describes the local behavior of a process where also all possible effects of the environment are considered that are visible for the process. The operational semantics of a process describes only the local states of the process; there does not exist a global state. The composed distributed semantics contains the global states constructed from the local states of the processes.

The compositional operational semantics can be simulated by a global semantics operating on global states by using amalgamation of process productions. This allows to consider each distributed computation composed from the local computations from a global point of view. Vice versa, a global step can be split into local computations if the global computation is compatible with the distribution of the processes.

1.2.1.2 Graph-interpreted Temporal Logic for the Specification and Verification of Distributed System Properties

To reason about system properties, considerable effort has been invested in the area of *temporal logic*. Starting with Pnueli a number of concepts are developed to check temporal logic specifications. One of the most studied concept is *model checking* [Eme90,Sti92]. To support the designer of a distributed system using distributed graph transformation also with the possibilities offered in the field of temporal logic, I introduce *graph-interpreted temporal logic* as a specification tool accompanying rule-based distributed graph transformation.

Properties of a distributed system are specified graphically by *graph-interpreted temporal formulas*. These formulas are temporal formulas where the variables are interpreted by *graphical constraints*: Such constraint is just a distributed graph equipped with variable assignments which is satisfied by a second distributed graph if this provides at least the same structure of the first.

For checking temporal formulas in a distributed system two main problems occur: First, most of the automated techniques for verifying temporal properties of systems assume finite systems. This condition is often neither satisfied in distributed systems nor in non-distributed systems. Second, there is no global state where formulas can be checked. This is especially a problem, when the checking of a global formula is of interest. I provide in this work a construction of a finite model build up from a set of graph-interpreted temporal formulas, that supports the checking of temporal formulas in a possibly infinite distributed system. This construction helps to solve the two problems stated above. The construction collapses the possibly infinite model for a distributed system to a finite model by identifying states which are logically indistinguishable with respect to a given set of graph-interpreted formulas. For checking a formula in the collapsed finite model any kind of model-checker can be used. If a formula is satisfied in the collapsed finite model it is satisfied also in the infinite one. If the formula is not satisfied in the finite collapsed one, however, the formula may be true in the infinite one. In this way, the construction of the finite model supports the checking of temporal formulas in infinite systems, but does not provide a complete solution.

Furthermore, the collapsed finite model supports the second point of checking global formulas in a distributed system as well. For that, the global formula is divided into local formulas. Local formulas are temporal formulas that can be checked with respect to one local process. The finite model constructed with respect to the global and the local formulas is used to reason from local satisfaction of the local formulas to the global satisfaction of the global formula by means of a model checker. Again, if the model checker gives a positive answer, the global formula is satisfied, if it answers negative, the global formulas can still be valid. For the local satisfaction of the local formulas, I can use again the construction of the finite model or other existing verification techniques.

1.2.1.3 Summary of Main Conceptual Results

- A componentwise specification of distributed systems by a type graph for the process and port types that may occur in the distributed system, a process grammar for each process type, that shows how processes of this type behave and a synchronization relation, that shows which process actions have to be synchronized.

- A compositional operational semantics composed from the local operational semantics of the processes according to the synchronization relation. The compositional operational semantics can be considered also from a global point of view by a global semantics using amalgamated process productions.

- Graphical specification of distributed system properties by graph-interpreted temporal formulas.

- Construction of a finite temporal graph model for a set of graph-interpreted temporal formulas for supporting the automatic verification of graph-interpreted formulas by means of a model-checker.

- A case study of a distributed configuration management system for testing and validating the conceptual results of the thesis.

1.2.2 Main Theoretical Results

1.2.2.1 Distributed Graph Transformation

The topological structure and its evolution can be described by graph transformations. The use of graph transformation for the specification was tackled in the 80's by Degano and Montanari [DM87]. They model the topological and temporal aspects of a distributed system, but abstract from the actual states of processes. The ideas have been extended by Montanari and Rossi in [MPR99] to a version being more applicable to distributed systems. In [Tae96], Taentzer takes up the ideas of a graphical description and transformation of both the topology and local states and combines both levels to an approach called *distributed graph transformation*. Her approach is more suitable for an early stage of system development, where a global view on the whole system is desirable to get an overview.

I get my inspiration mainly form these two approaches to develop a new approach called *distributed graph transformation*, too, even if my approach goes beyond the approach of Taentzer. However, I keep the notion of distributed graph transformation because of historical reasons.

Just as [MPR99], I consider a distributed system consisting of concurrent processes communicating over shared ports. Just as [TFKV99,Tae96], I integrate the local states in this topological description. However, in contrast to [TFKV99,Tae96], local states can be specified by objects of an arbitrary cocomplete category.

The specification of a distributed system contains a *network type graph*, that is, a type graph [CELP96] representing the process and port types that may occur in the distributed system and the connections allowed. The topology of the distributed system is modeled by typed graphs with respect to the network type graph. For each process type in the network type graph, a *process grammar* is given. A process grammar for a process type provides a set of *process productions* and a *start production*. Process productions specify the possible moves of the process and the start production starts the process and initializes it. A process move may affect the transformation of the topology or the transformation of its local state. For the application of a process production only the local state of one process is necessary, i.e. I do not need a global view on the distributed system for its application. From the viewpoint of object-oriented modeling, a process grammar can be seen as a class. Each instance of a process grammar gets the "program" specified in the process grammar. I provide a name space for the process instances for each process type. The synchronization of process instances is given by a *synchronization system*, that determines which process productions of which process instances have to be synchronized and over which data the synchronization takes place. Altogether, a distributed system is specified by a *distributed system specification* consisting of a network type graph, a process grammar and a name space for each process type in the network type graph and a synchronization relation.

I define the semantics of a distributed system specification by *transformation systems*, introduced in [GR98,GR99]. Transformation systems are a common semantical framework in which specification techniques can be interpreted, in particular also distributed graph transformation. This fact enables to use the results for the composition of transformation systems to specify a compositional semantics for a distributed system. The distributed semantics for a distributed system specification is composed from the transformation systems for the single process instances, called *open process semantics*. The open process semantics of a process instance includes the behavior of a process instance generated by the process

productions of the corresponding process grammar and all possible effects of the environment on ports. The composition of the transformation systems for the process instances is induced by the synchronization relation. This relation determines how the transformation systems are connected.

In previous works to distributed graph transformation [TFKV99,Tae96], the operational semantics of the distributed system is given by the application of global productions on global distributed system states. I relate this approach of a global operational semantics to the compositional operational semantics, where all applications of productions are local to local states. The global operational semantics for a distributed system specification is a transformation system generated by amalgamated process production, where the synchronization relation induces which process productions of which process instances have to be amalgamated. The transformation systems for the global operational semantics and the compositional operational semantics are isomorphic, if the global operational semantics is restricted to those computations that can be split in local computations for the process instances. Therefore, each distributed computation composed from the local computations can be regarded from a global point of view as a global application of an amalgamated production as well. Vice versa, a global step can be split into local computations if the global computation is compatible with the distribution of the process instances.

1.2.2.2 Integration of Graph Transformation and Temporal Logic

Graph-interpreted temporal logic specializes propositional temporal logic by interpretation on *graph transition systems* and their specification by temporal logic formulas. A graph transition system is a transition system where the states are distributed graphs equipped with variable assignments. Graph transition systems have the same discriminating power like general transition systems.

For the sake of a uniform presentation, the variables form a graph as well and the assignments are represented by distributed graph morphisms. The same idea is used for specifying properties of states by *graphical constraints*: Such constraint is just a pattern state (i.e., a distributed morphism) which is satisfied by a second state if this provides at least the same structure of the first one. In this way, a relation on graphical constraints can be defined, that turns out as a pre-order on graphical constraints. This concept has originally been developed in [HW95] where graphical constraints were used in order to express static consistency properties. In [Koc96,HEWC97] it was combined with propositional temporal logic, able to express also dynamic properties of systems.

Most of the automated techniques for verifying temporal properties of systems assume transition systems with finite sets of states. Hence, for applying such techniques it is necessary to collapse infinite transition systems to finite ones by identifying states which are logically indistinguishable. A transition system is called *fully abstract* if it is completely collapsed in this sense, i.e., if any two indistinguishable states are identified in the system. On the other hand, a transition system may represent a whole class of systems if it is *typical* for that class in the sense that it satisfies a formula if and only if this formula is satisfied by *all* transition systems of that class. I present a construction that, for a given interpretation of the propositional variables as graphical constraints, produces a transition system which is both typical and fully abstract. More exactly, I provide the construction of the transition system for any pre-order that has least upper bounds and I show that the pre-order on

graphical constraints has least upper bounds. I argue that these two properties, typical and fully abstract, correspond to the existence and uniqueness of transition system morphisms from all transition systems of the class to the typical, fully abstract one. In other words, the constructed system is a final object in a suitable category of transition systems. Moreover, the system is finite provided that I only use finitely many different constraints in the temporal specification. Then, the typical system can be used to examine the validity (entailment, tautology, etc.) of formulas for the whole class of system by checking the corresponding property in the typical transition system only.
I show how global formulas can be checked by means of the typical model. This is done by dividing the global formula in local formulas and the global distributed system to sub-systems, where the local formulas can be checked. The sub-systems are constructed in such a way, that the satisfaction of a local formula in the sub-system implies satisfaction in the global system. Under the assumption of the satisfaction of the local formulas in the corresponding sub-systems, the global formula can then be checked by a model checker in the typical model.

1.2.2.3 Summary of Main Theoretical Results

The main original contributions to the formalization are:

- The development of a single pushout approach to conditional distributed graph transformation where local states are specified by objects of a cocomplete category **C**. (cf. Section 4.3).
- The integration of the single pushout approach to distributed graph transformation in the framework of transformation systems of Martin Große-Rhode (cf. Definition 4.3.22).
- A compositional operational semantics for a distributed system specification based on the composition of transformation systems (cf. Subsection 4.3.4) and the relation to the global distributed semantics based on amalgamated productions (cf. Theorem 4.3.36).
- Integration of temporal logic with pre-orders and the construction of a finite typical and fully abstract transition system (cf. Section 5.2).
- Definition of a pre-order over distributed graphs, where local states are attributed graphs, that has least upper bounds (cf. Section 5.3).
- Compositional verification for reasoning from local properties to global properties by means of the typical and fully abstract transition system for the pre-order over distributed graphs (cf. Theorem 5.3.19).

1.3 Organization of the Thesis

The remainder of this thesis is divided into two main parts. Chapter 2 and 3 explain the concepts informally, whereas Chapter 4 and Chapter 5 provide the formal treatment of my approach. The two parts can be read independent of each other. A reader, whose intention is simply to use distributed graph transformation for the specification of a distributed system should read Chapter 2 and 3. The reader interested in the formal background is

referred to Chapter 4 and 5. However, also for her/him the informal chapters may be useful, since it provides additional examples. The informal as well as the formal part is divided again in a part concerning the single pushout approach to distributed graph transformation (Chapter 2 informal, Chapter 4 formal) and a part concerning the graphical specification of distributed system properties by graph-interpreted temporal logic (Chapter 3 informal, Chapter 5 formal).
As mentioned above, the introduction in Chapter 2 and 3 is intended for a reader who is not yet familiar with graph transformation or temporal logic, however, who would like to use the approach for the specification of a distributed system. Therefore, the primary aim of these two chapters is to explain the main concepts of my approach in such a way that it enables a designer to use the concepts. Due to this motivation, I do not go in full detail in all concepts possible, but I focus on the most needed ones. I provide a case-study of a distributed configuration management system throughout the two informal chapters to support the explanations by concrete examples.
Chapter 2 firstly gives a brief introduction to the case study. By means of this case study I introduce the specification of a distributed system by distributed graph transformation in four steps. Each step provides the concepts necessary to perform and understand the step. Step 1 deals with the topology of a distributed system and introduces typed graphs for its specification. Step 2 is concerned with the local data states of the distributed system. In this part local data states are specified by attributed graphs, since they fit best for the case study. Step 3 is concerned with distributed graphs and distributed productions and the componentwise specification of the behavioral aspects in a distributed system. The last step deals with the synchronization of processes and both the global operational and the compositional operational semantics.
Chapter 3 is concerned with the specification of dynamic system properties for a distributed system specified by distributed graph transformation. I introduce graph-interpreted temporal formulas that are based on distributed graphs and distributed morphisms introduced in Chapter 2. In this way, the specification technique for the distributed system and the requirements to its behavior coincide. I explain, how graph-interpreted temporal formulas are specified and introduce their semantics, i.e. the question, when a graph-interpreted formula is satisfied by a model. This includes a brief introduction in propositional temporal logic. The verification of graph-interpreted temporal formulas is addresses as well. The main focus lies on the construction of the typical model, that is introduced by an example taken from my case study. This example will be also used for checking a graph-interpreted formula by means of a model checker for propositional temporal logic.
As mentioned above, Chapter 4 and 5 provide the formal background of my approach. Chapter 4 defines the single pushout approach to distributed graph transformation with local **C**-states. I show how a distributed system is specified componentwise by a distributed system specification consisting of a network type graph, a process grammar and a name space for each process type in the network type graph and a synchronization relation. I introduce transformation systems and their composition to define the compositional operational semantics for distributed graph transformation. I show how distributed graph transformation fits into the framework of transformation systems such that compositional results with respect to transformation systems can be used in my case as well. I compare the compositional operational semantics with the global operational semantics generated by amalgamated productions and conclude this chapter by a concrete category **C** used in the

case study of Chapter 2, that is, the category of attributed graphs.
Chapter 5 considers the integration of distributed graph transformation and a propositional temporal logic. Therefore, I start this chapter by a brief introduction in temporal logic. I define a slightly different notion of a temporal model based on a pre-order and provide the construction of the typical and fully abstract transition system with respect to a pre-order. Then, I consider distributed graphs together with a variable assignment and define a pre-oder over them and show that this pre-order has least upper bounds. I conclude with aspects of compositional verification and show how local properties are preserved by the composition of distributed system components and how the typical model can be used to reason form local satisfaction to global satisfaction.
Chapter 6 discusses the related work and I close by presenting the conclusions and intentions for further work.

Chapter 2

Single Pushout Approach to Distributed Graph Transformation *Application to a Distributed Configuration Management System*

This chapter presents the concepts of *distributed graph transformation* in the single pushout approach for the specification of distributed systems. The introduction provides the concepts informally, supported by a concrete distributed system and is guided by a designer, who does not know graph transformation yet and is not necessarily interested in the formal background. The formal description of our approach can be looked up in Chapter 4 if a closer look in the concepts is desired.
We consider in this work distributed systems consisting of a number of concurrent processes together with their local states, where communication between processes takes place over common ports. The topology of the distributed system may change dynamically throughout its life-time, that is, new processes can be added or removed, new ports established or existing ports may be cut. This is similar to the dynamic reconfiguration possibilities in the π-calculus [MPW92]. We call this kind of a distributed system *open* since an interaction with the environment during run-time is possible. Local state transformations of the processes and the data sent over ports are in the viewpoint of our specification technique as well. The communication may take place asynchronously or synchronously.
To specify a distributed system, the designer has to answer a list of questions.

How does the topology of the system look like? The question deals with the types of processes that participate in the distributed system and the possible ports indicating communication paths. It has to be determined if new processes can join the distributed system and, if it is possible, how this shall happen. As well the deletion of processes has to be tackled and the creation and deletion of ports. Altogether, the designer has to determine the statical as well as the dynamical aspects that occur with respect to the topology of the distributed system.

Which kind of data are used? Processes have local data states on which they operate. It has to be specified how this data look like and which specification technique seems to be most suitable. Data may be specified by graphs, partial algebras or other specification techniques. The data types in ports have to be specified as well in order to provide a communication interface between processes.

How does a single process behave? It has to be worked out which actions a process can perform on local states or the topology and under which conditions the process can perform these actions.

How does communication take place? A process action may be not independent from its environment, but it has to perform this action together with other processes. Therefore, synchronization is necessary with possibly several other processes. The synchronization of process actions has to be specified as well as the way how communication shall take place in the synchronization case, i.e. over which structures the synchronization is performed.

Distributed graph transformation supports the designer of a distributed system with concepts to answer these questions. Topological aspects can be specified by *typed graph transformation*, where the statical aspect is covered by a *network type graph* and the dynamical aspect by *network productions*. The specification of local data and their manipulation is supported by *attributed graph transformation* that includes the specification of the data used in processes and ports as well as their rule-based transformation.[a] Topological as well as local state aspects are integrated in *distributed graphs* which specify the topology and the local states of distributed system states in one common structure. Their transformation is defined by *distributed productions* including topology as well as local state transformations. The behavior of a process is given by a *process grammar* that contains the distributed productions for its actions. A process grammar can be seen as a pattern, that describes how instances of the process behave. Synchronization constraints between process actions are given in a *synchronization relation* restricting the behavior of processes.
I introduce the concepts mentioned above by means of a case study of a *distributed configuration management system* used in distributed software development. Since I would like to focus in this work rather on the concepts of distributed graph transformation than on the detailed features of the case study and in order to achieve a more legible introduction I restrict myself to the main features of the case study. A more detailed explanation of the distributed configuration management system can be looked up in [TFKV99,Vol97]. However, even though the case study is restricted in this work, it provides all the features of a distributed system that we would like to specify.

The following list shows on the one hand the organization of the remainder of this chapter, on the other hand it provides a check-list of working steps a designer has to do if she/he specifies a distributed system by distributed graph transformation. The steps are:

1. Specify the topology of the distributed system in a so-called *network type graph*. This graph provides the process and port types of the distributed system as well as their admissible connections.
2. Specify the data used for the local states of processes and ports.
3. Specify the behavior for each process type x given in the network type graph by means of a *process grammar* of type x providing a set of *process productions* of type x and the initial state of a process of type x. From the object-oriented viewpoint, the process grammar can be seen as a class for processes of type x. The name space for the instances of type x that may occur in the distributed system is given in this step as well.
4. Specify the synchronization constraints between the process instances, i.e. which process moves have to be performed synchronously?

[a] I have chosen attributed graph transformation in this chapter since it supports the case study of a distributed configuration management system best. However, other specification techniques as graphs or partial algebras can be chosen as well. The admissible class of specification techniques for local states is discussed in Section 4.2.

The result of these four working steps is a *distributed system specification* consisting of a network type graph, a distributed grammar consisting of a process grammar and a name space for process instances for each process type in the network type graph and a synchronization relation.
Before I breathe life into these points, I introduce the case study of a distributed configuration management system.

2.1 Distributed Configuration Management

The necessity of distributed configuration management is motivated by the structure of today's software projects. Not only that they are become bigger nowadays but also stricter quality requirements make it even more difficult to plan and to coordinate a project. The result of a development process often has not much in common with the original concept. Additionally, two third of the complete amount of time and money are spent for maintenance and further developments after the initial project was completed [Boe87,VW83]. Furthermore, the knowledge to complete a big project is hardly available in a single company. Hence, external project partners must contribute. More often it is also economically sensible to pass work to external subcontractors. With an ever growing demand on distributed software development the problems of communication, coordination and quality management grow as well. For instance, all project sites must have access to a consistent up to date set of project documents. When one project site changes a document, it must become known to all other project sites, too. This is especially a problem when there is no central archive available for all project partners. To cope with these problems, several concepts of *distributed configuration management* are proposed in the literature [Tic94,Est95,Som96,Con97,vdHHW96,CW97,Vol97]. I follow in this work the distributed configuration management system presented in [Vol97].

Documents, Revisions Distributed software projects are carried out by several project partners, distributed over different sites. Each project partner works on a number of *documents* forming a part of the software product that is going to be developed in the distributed software project. A document may be a text file, an executable file, a code file etc. manipulated by a project partner. Documents evolve during the development process and each state of an evolving document yields a new version, called a *revision*. Revisions of the same document have the same *document identification number* and are related by a *revision number* that shows the temporal order on the versions of a document and their dependencies. Revisions may have several additional attributes: the *contents* of a revision shows the state of the document version, a *revision name* helps to distinguish revisions not only by the revision number, the *project partner* shows who created this version, the *creation time* and *date* show when the version was created etc. For the sake of simplicity, I restrict myself in this work to the document identification number, the revision number and the contents of a revision.

Revision Archives and Replication The documents of one project partner have to be made accessible for remote project partners if they need the documents to construct their own software component. In order to present the documents developed in one site to remote project partners, each project partner has a *revision archive* that contains the revisions of the documents of the project partner. The project partners of the distributed project have access to the revision archives of remote project partners, such that a connection between revision archives of two partners can be established if a data replication is intended between them. The connection is usually directed in the sense that one project partner exports its documents and the other one imports the documents. To replicate documents in the other direction, an own connection has to be established. By means of a connection, revisions can be replicated from one project partner to the other one, performed into two phases: first, one site exports its documents, second, the other site imports these documents into its own revision archive. These two steps are usually performed asynchronously. If the replication is finished, the connection may be cut again. In this way, the structure of the distributed software project changes dynamically, since also new project partners can join or old one may leave the project. However, even if a project partner leaves the project, the documents have to be preserved within the project. This is based on one of the most important properties of a revision archive: *no revision is allowed to be deleted ever from the revision archive.* This rule is motivated by the practice, where client requirements can make the need of old revisions necessary.

Workspaces, Check-in and Check-out of Revisions A project partner itself is in generally represented by an entire developer team consisting of several developers working at the same site. When a document of the site is changed or a new document is created by one of the developers, this is not directly done in the revision archive but by means of a *workspace*. A workspace is designated to a developer, in which she/he works, that is, where documents are manipulated. If a developer would like to change an existing revision of a document, she/he firstly connects her/his workspace with the revision archive and copies the corresponding revision from the revision archive into her/his workspace. To put a copy of a revision from the archive into the workspace is called the *check-out* of the revision. Since revisions are not allowed to be deleted from the revision archive, the revision checked out is a copy of the revision in the archive. The actual change of the document takes place within the workspace or a new document may be created. When the work is finished, the changed or the new documents have to be put back into the revision archive. This is called the *check-in* of the documents. The check-in of a document into the revision archive deals also with the calculation of the revision number for the version of the document checked in. If a new document is checked into the revision archive, this can be done without problems, since there does not exist a predecessor version. The check in of a changed revision is more complicated, since the changed revision must be inserted as a successor of the revision from which it was checked out. Therefore, the revision number of the revision from that the changed revision was checked out influences the calculation of the revision number of the changed revision.

Developers can connect workspaces to revision archives in arbitrary number and can disconnect them, whenever a developer decides to cut a connection.

2.2 Step 1: The Topology of the Distributed System

In the beginning of the development process for a distributed system it is reasonable to consider its topological aspects. The idea of using graphs for describing the topology of a distributed system was already put in practice (e.g. in [MPR99,Tae96]). Also this work models the topology of the distributed system by a graph, more exactly a *typed graph* [CELP96], where nodes and edges of a graph are equipped additionally by type informations.[b]
A typed graph (Def. 4.1.2) consists of a set of nodes and a set of edges. Edges of a graph are directed, i.e. they run from one node (the *source* of the edge) to a node (the *target* of the edge). An edge has always a unique source and target node, not allowed are edges without source resp. without target node, as well as edges between edges. Each node and each edge has a *type*. Several nodes resp. edges of the same type may occur in one graph.

Example 2.1 (typed graph)
The typed graph in Figure 2.1 models a possible network of a distributed configuration management system. I model revision archives and workspaces by black square nodes, that

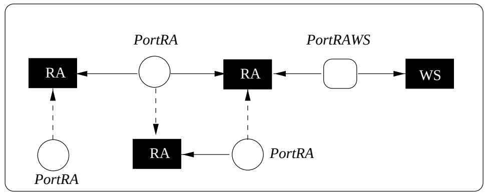


Figure 2.1: *The network of a distributed configuration management system modeled by a typed graph.*

have the type RA for revision archives and WS for workspaces, respectively. The replication of documents between revision archives takes place over a common port. The port is modeled by a node of type $PortRA$, that is connected to a revision archive by a dashed or a solid edge. The connection with a dashed edge means, that the port belongs to the revision archive. Each revision archive has its own unique port of type $PortRA$. If the revision archive is connected with a port $PortRA$ by a solid arrow, the revision archive uses the remote port. Several revision archives may use a port of a remote revision archive at the same time. It is also possible, that no revision archive uses a port of an archive. The dashed and solid arrows show additionally the replication direction between archives: documents are replicated from the archive, that is the target of the dashed arrow to the archive, that is the target of the solid arrow. The port of type $PortRAWS$ is used for the communication between a revision archive and a workspace. Over these ports, the documents are checked in resp. checked out. Since there does not exist a designated direction for the check out and the check in of documents, I have only solid arrows. △

As mentioned in the introduction of this chapter, I consider a distributed system to be a system consisting of concurrent processes communicating via common ports. For a given distributed system, the types of processes and the types of ports that can occur in the

[b] A general introduction to graph transformation is given in [Roz97].

distributed system are modeled in a so-called *network type graph* (Def. 4.3.1). The network type graph is a graph that contains for each process and port type one node and for each edge type an edge. I require that processes can be only connected via ports. A direct connection between processes is not possible. The source and the target node of an edge in the network type graph specify the node types, between that instances of the edge may exist. The network type graph can be seen as a pattern for a whole class of graphs. A graph satisfies the pattern given by the network type graph, if it has only types provided in the type graph and connects nodes with edges of the correct type only. I call the graph in this case a *network graph* with respect to the network type graph (Def. 4.3.2).

Example 2.2 (network type graph for the case study)
The network type graph of the case study of a distributed configuration management system is shown in Figure 2.2. The example in 2.1 introduced already the types necessary

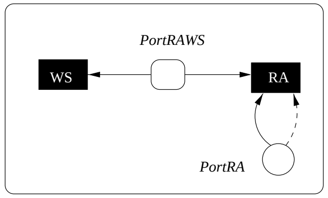


Figure 2.2: *The network type graph of the case study.*

for modeling a distributed configuration management system. There is a process type for revision archives, that is type RA, a process type for workspaces, that is WS, and the node types for the two different ports, that are $PortRA$ for ports between revision archives and $PortRAWS$ for ports between revision archives and workspaces. The types RA and WS are the process types and the $PortRA$ and $PortRAWS$ are the port types. The network type graph reflects our view of a distributed system, since the processes are connected via ports over which the communication takes place. Since I would like to specify, which port of type $PortRA$ belongs to which revision archive and which ports are only used by an archive, I introduce two types of edges between ports of type $PortRA$ and revision archives, namely a dashed and a solid one. A dashed arrow shows that the port belongs to the archive, a solid one that the archive uses the port.
The graph in Figure 2.1 is a network graph with respect to the network type graph in Figure 2.2, since it uses only types given in the network type graph and the connections obey the network type graph as well. △

Network graphs with respect to a network type graph represent the possible topologies of the distributed system. To specify the reconfiguration of a given topology, i.e. the transition form one topology to another one, I introduce *network productions* given by *partial graph morphisms* $r : L \rightharpoonup R$ between graphs L and R (Def. 4.1.2). A partial graph morphism $r : L \rightharpoonup R$ consists of a pair of partial injective mappings between the set of nodes and the set of edges of L and R, that are compatible with the graph structure and the types. Compatible with the graph structure means that whenever the mapping for edges is defined for an edge, then the mapping for nodes is defined for the source and the target node of the

edge. Moreover, the source node (target node) for this edge has to be mapped on the the source node (target node) of the image of the edge. Compatible with the type means that nodes and edges are mapped only on nodes and edges of the same type. If the mappings for nodes and edges are total, I call the graph morphism *total*.
The graph L of the network production is called *left-hand side* and describes which objects a network graph must contain such that the production can be applied. The objects in L, where the partial graph morphism is defined, are intended to be preserved by the application of the production, the other one are designated for deletion. The graph R of the network production is called *right-hand side*. All objects of the right-hand side that are outside of the image of $r(L)$ are added to the preserved objects.

Example 2.3 (network production)
Figure 2.3 shows an example of a network production for the reconfiguration of the topology. The left-hand side consists of a revision archive, that uses a port $PortRA$ of another revision archive (indicated by the solid arrow). The topology shall be changed in this way, that the connection to the port is cut and a new workspace is connected to the revision archive. This

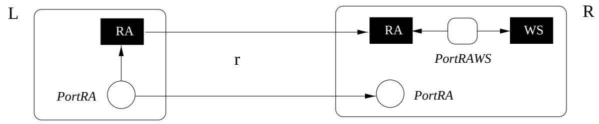


Figure 2.3: *Example of a network production.*

intended meaning is modeled by the partial graph morphism for the production in Figure 2.3. The mapping for nodes is total, since the revision archive as well as the port in L have a counterpart in R (indicated by the arrows from L to R). Therefore, the archive and the port are preserved by the production. The mapping for the edge is partial, since the mapping is not defined for the edge in L. Therefore, the production deletes this edge and cut the connection between the port and the archive. The mappings are type compatible, since the nodes of the left-hand side are mapped on nodes of the right-hand side that have the same type. The compatibility with respect to the graph structure follows immediately from the fact, that the mapping for edges is undefined. Since the workspace and the port $PortRAWS$ of the right-hand side are not in the image of the morphism, they are newly created. △

A production $r : L \rightharpoonup R$ can be applied to a network graph G if the objects of the left-hand side L of the production occur in G. This occurrence is characterized by the existence of a total graph morphism from L into G, that we call *match* in the following. The application of the production is performed by adding the new elements of the right-hand side of the production (those, that are not in the image of the production morphism) and delete the objects that are designated for deletion by the production and the match (those objects of G that have a pre-image under m that is outside of the domain of r). If the result of the deletion is not a graph anymore, i.e. there are some edges without target or source node, then delete also the edges. We call the application of a production to a graph a *direct derivation*.

Example 2.4 (direct derivation)
The example in Figure 2.4 shows the production in Figure 2.3 together with a match in a network graph G. The network graph G shows a possible topology of a distributed config-

uration management system consisting of three revision archives connected via ports. The match specifies, that the production shall be applied to the archive in the upper right corner and the upper port $PortRA$. For building the direct derivation, the new workspace and the

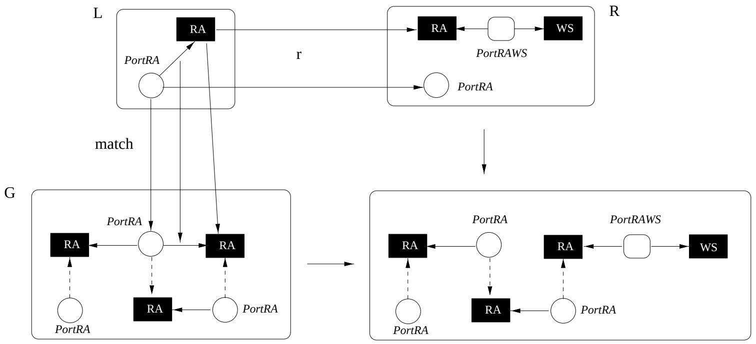


Figure 2.4: *Example of a direct derivation.*

new port of type $PortRAWS$ are added to the revision archive in G given by the match and delete the edge between the revision archive and the port. △

2.3 Step 2: The Local Data Aspects of a Distributed System

After having specified the topological aspects of the distributed system by a network type graph, the local data aspects of processes and ports are considered and their possible local states are determined. The local data of a process can be specified in different specification techniques as graphs or partial algebras, hierarchical graphs, etc. The choice for one specification technique is driven by the application. As mentioned above, I use *attributed graphs* for representing local states. There are several approaches for attributed graphs in the literature [Wag97,CL95,Löw93,Sch92]. I use the approach of [CL95], since this approach fits best to our case study.

An attributed graph is a graph as explained in the previous section where nodes and edges can be additionally equipped by attributes. Attributes may be natural numbers, strings, booleans or elements of more complex algebraic structures. One node resp. one edge may carry several attributes at the same time. The attribute information, i.e. the possible attributes for a node resp. an edge, is given formally by an *attributed graph signature* (Def. 4.4.1). Since the reader, especially a reader who is not familiar with the formal background, will appreciate a less formal introduction, I provide the attribute information in a so called *attribute graph*. The attribute graph is similar to a type graph and shows the attribute names and the attribute types a node or an edge may carry. Since not all nodes and edges have to carry the same attributes, the attribute graph has for each different node resp. edge attribution a single node and a single edge, respectively. An attributed graph is an attributed graph with respect to an attribute graph, if its nodes and edges carry only attributes defined in one of the nodes

resp. edges in the attribute graph, where the attributes have concrete values taken from the domain of the corresponding attribute type.

Example 2.5 (The attribute graph of the case study)
In the case study of a distributed configuration management system, I have to specify the local data in revision archives and workspaces. The local state of a revision archive as well as the local state of a workspace is given by a set of revisions. In the case of a revision archive, that are the revisions developed in the site, in the case of a workspace, that are the revisions the developer works on. I introduced the attributes of revisions in the description of the case study in Section 2.1. I consider in this work the attributes *document identification number*, *revision number* and the *contents* of the revision. The document identification number, short $DocID$, indicates to which document the revision belongs. The revision number, short $revnr$, indicates which version of the document the revision is. The contents shows the current state of the document in the version of the revision. Figure 2.5 shows the corresponding attribute graph for revisions on the left-hand side. Since all revisions of documents in revision archives

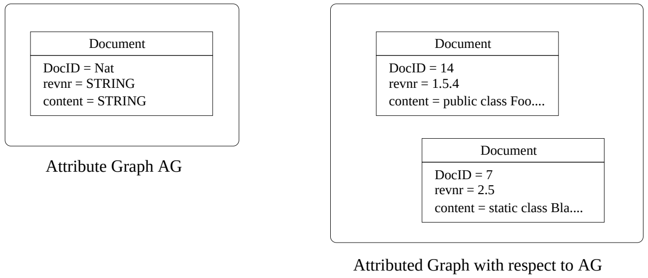


Figure 2.5: *The attribute graph for documents and an attributed graph.*

as well as in in workspaces have the same attributes, I have only one node in the attribute graph. Moreover, I have no edges in the local states of a revision archive or a workspace. The attribute type of the document identification number $DocID$ is given by natural numbers. The attribute type for the revision number is a string, since a revision number is represented by a sequence of numbers separated by a point. Consider as example the revision numbers of the documents in the right hand side of the figure. The upper document has the revision number 1.5.4, the lower document has the number 2.5. The meaning of this syntax is as follows: The first number of the sequence indicates the number of the document and the first version of a document has only a single number as revision number, for example the number 1. If this first revision is changed, a new revision is generated having the number 1.1. the next version has the number 1.2 and so on. If the revision 1.2 is changed, we get a revision 1.2.1. If we change the revision 1.2 again, we get a new revision 1.2.2 and so on. The contents of a revision is a string as well. The attributed graph signature for this attribute graph is given in Example 4.19.
The local state of a revision archive or a workspace contains several nodes, that have the attributes given in the attribute graph, where the attributes have concrete values. On the right-hand side of the Figure 2.5 two revisions of documents with respect to the attribute graph on the left-hand side are shown. △

For the specification of the case study in this work, the attributes for revisions of documents coincide in revision archives, workspaces and the ports. Therefore, the specification of the local data is completely given by the attribute graph in Figure 2.5. If the attributes for revisions shall differ in revision archives, workspaces and ports, the attributes have to be specified separately for the archive, the workspace and the ports. This is meaningful, especially if additional attributes for revisions are considered. Since the focus of this work are the concepts of distribution and not the development of a sophisticated attribute structure and for the sake of space, I have chosen this simpler case. The case, where the attributes of revisions in revision archives and workspaces differ is possible in our framework as well. This more detailed specification of the attributes for the local data is presented in [TFKV99].

In contrast to types, that are fixed throughout the existence of the nodes and edges, attributes can change their value. This enables, for example, to model the change of the contents of a revision. How an attribute shall be changed is specified by an *attributed production* (Def. 4.4.5). In a first step, one can consider an attributed production between two attributed graphs as a partial graph morphism between graphs as explained in step 1. However, since we do not want to provide for each possible contents of a revision one or possibly more productions, that model the change of a concrete contents, we introduce *variables* for the attributes in the attributed graphs of the production. The attributes of preserved nodes in the left-hand side of an attributed production may be changed by the production, where the new value can be calculated by a certain operation possibly based on the variables of the left-hand side. In such a way, the variables allow to specify the change of attribute values more abstractly. I have to mention, however, that the change of attributes is not as easy as it is explained here. To get a more detailed explanation of the change of attributes, the reader may have a look in the formal part in Section 4.4.1 or in the literature mentioned there.
An attributed production can be applied to an attributed graph by finding a match (Def. 4.4.6) for the left-hand side of the production as explained in the previous section for typed graph transformation. But in addition, the match substitutes the variables of the left-hand side with concrete values given in the attributed graph, where the attributed production is applied to.
The application of an attributed production at a given match is given by the direct derivation as explained in step 1 for typed graphs (Def. 4.4.6). In the case of attributed graphs, the values of the attributes of the derived graph are additionally calculated based on the assignment of variables given by the match.

Example 2.6 (attributed production and direct derivation)
At the top of Figure 2.6 the attributed production for the change of the contents of a revision in the workspace is shown. The attributed production has one document in its left-hand side, where the attributes does not have concrete values but variables $x1$, $s1$ and $s2$. The attributed production preserves the revision of the document, but changes the attribute *contents*. The attribute value of the attribute contents in the document of the right-hand side is calculated by an operation $make - private$. The intended meaning of this operation from the viewpoint of object-oriented programming languages is, that it declares a class to be a private class. The operation get the old contents, that is a class in the example, as input. The revision number and the document identification number remain unchanged

by the production. This is indicated by the same variable for $DocID$ and $revnr$ on the left-hand side and the right-hand side of the production. This production is applied to a

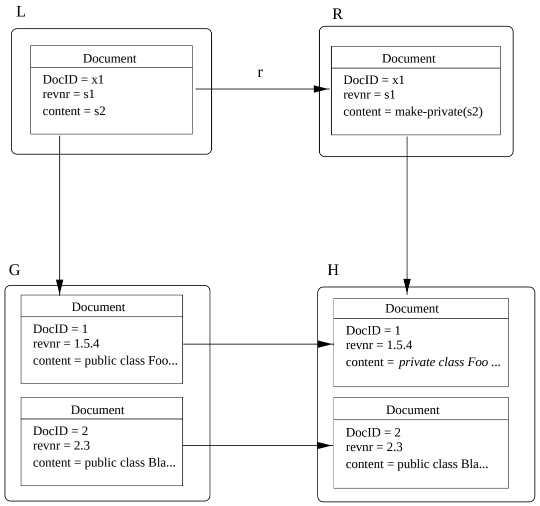


Figure 2.6: *Changing the contents of a document.*

graph G consisting of two documents that may represent the set of revisions in a workspace. The match for the production chooses one of the documents to change its contents. In the example, the match chooses the upper document with $DocID = 1$. Because of this match, the variable $x1$ is substituted with the value 1, the variable $s1$ with 1.5.4 and the variable $s2$ with *public class Foo...*.
The direct derivation of the attributed production at this match does not delete or create any node, since the attributed production does not change the graphical structure. However, whereas the graphical structure is preserved, the attribute *content* of the document 1 is changed from *public class Foo...* to *private class Foo...*. △

2.4 Step 3: Componentwise Specification of Distributed Systems

In step 1, the topological aspects of a distributed system were specified by a network type graph that provides the types of processes and ports that may occur in a distributed system. In step 2, the local data states are specified by attributed graphs representing the local states of processes and ports.
In the first part of this step, I introduce *distributed graphs* that integrate the topological and the local data aspects in one common structure. I show how distributed graphs can be manipulated by *distributed productions* and introduce *distributed graph transformation*. We will see, that this first approach does not reflect the locality aspects in a distributed system very well. In order to use distributed graph transformation for the specification of distributed systems, I introduce in the second part of this step concepts for distributed

graph transformation reflecting locality aspects. These concepts are mainly based on *process productions*, that are restricted distributed productions modeling an action of one single process without knowledge of other processes. For each process type in the network type graph, we provide a *process grammar*. A process grammar of a certain type specifies the behavior of a process of this type and contains a set of process production of the type and a *start production* to create the process. A process grammar can be seen as a class, that describes how instances behave. A *distributed grammar* contains a process grammar and a name space for process instances for each process type given in the network type graph. The name space represents the names available for process instances of a certain type.

2.4.1 Distributed Graph Transformation – Integrating Topological and Local Data Aspects

Until now we are able to describe the topology of a distributed system by typed graphs and the local states by attributed graphs. I am going to integrate now both concepts in one common structure, called a distributed graph (Def. 4.3.3), to describe an entire distributed system's state. The integration is provided by a mapping that assigns to each node of the network graph an attributed graph, called *local state*, and to each edge an attributed graph morphism, that is compatible with the node assignments.

Example 2.7 (distributed graph)
Consider the Figure 2.7 for an example of a distributed graph. The network graph consists of a revision archive and its port *PortRA*. The local state of the revision archive shall contain two revisions of documents, where one revision is put into the port. This revision shall be the only document in the port *PortRA*. The local states for the revision archive and the port are modeled by the attributed graphs at the bottom of the figure. The assignment

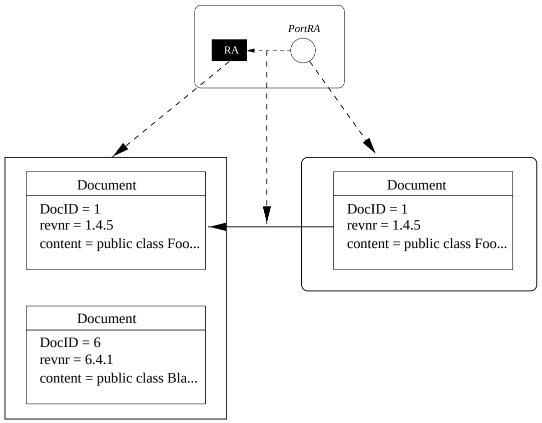


Figure 2.7: *A distributed graph integrates the topology and local states.*

of the local states to the revision archive and the port is shown by the dashed arrows. To indicate graphically that the document in the port comes from the revision archive, the edge

of the network graph is refined to an attributed graph morphism, that maps the document of the port to the document in the revision archive. △

Notation of distributed graph: I introduce now the notation of distributed graphs used in the following parts of this work, where the network graph and the assignment of the attributed graphs and attributed graph morphisms to the network graph is not explicitly shown. Consider the example in Figure 2.8, that shows the distributed graph of Figure 2.7 in the notation used in the following. There is a box for each process and port of the network

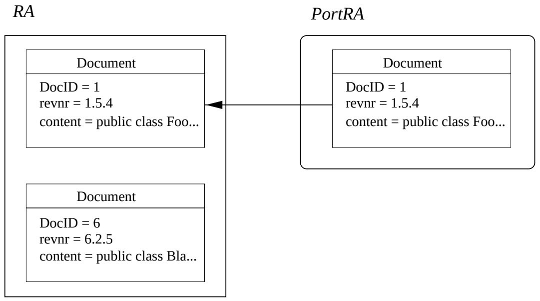


Figure 2.8: *Notation of distributed graphs.*

graph, where the local state of the process resp. the port is included. The box is marked by the name of the corresponding network node. The attributed graph morphisms for network edges are explicitly drawn in this notation, but not the network edge itself. However, there is a network edge in the network graph if there is an attributed graph morphism. If the attributed graph morphism for a network edge is totally undefined, I draw the network edge.

The manipulation of distributed graphs is modeled by *distributed productions* (the general definition is given in 4.3.7, for the case study in 4.4.11), that are *distributed morphisms* (Def. 4.3.4) with some special properties. A distributed morphism consists of a partial graph morphism between the network graphs, called *network morphism* in the following, and an attributed graph morphism for each node in the domain of the network morphism, called *local morphism*, between the local state for this node and the local state of the image of this node under the network morphism. The local morphisms have to be compatible with the assignment of attributed graph morphisms for network edges in the following sense: For each network edge, that is in the domain of the network morphism, the composition of the attributed morphism for the network edge and the local morphism for its target node has to be equal to the composition of the local morphism for its source node and the attributed graph morphism for the image network edge.
A distributed production is a distributed morphism, where all local morphisms are attributed productions as explained in Section 2.3. Since the network morphism of a distributed production is a network production as explained in Section 2.2, a distributed production specifies the transformation of the topology together with the transformation of local states at once.

Example 2.8 (distributed production)
The example of a distributed production in Figure 2.9 models the replication of a document of one revision archive to a connected remote revision archive. The left-hand side shows a distributed graph with two revision archives connected via a port $PortRA$. One of the revision archives contains a document. The right-hand side of the distributed production shows that the document is replicated into the lower revision archive, where the connection between the original and the replicated document is given by the document in the port. The

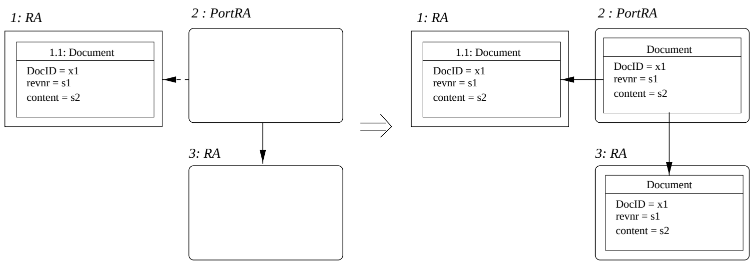


Figure 2.9: *A distributed production for the replication of documents.*

network morphism of the distributed morphism is indicated by the numbers at the network names. The nodes of the left-hand side are mapped to nodes of the right-hand side that have the same number. The network morphism is total in this case, i.e. the revision archives, the port and their connection are preserved by the production. Since the network nodes are preserved, we have three local morphisms: The local morphism for the revision archive $1 : RA$ is an attributed production that does not change the local state. The attributed production for the port as well as the second revision archive $3 : RA$ creates a new document, where the attribution is taken from the document in $1 : RA$. The local morphisms are compatible with the assignment of attributed graph morphisms to the network edges, since for both network edges in the left-hand side of the production holds: It is the same whether I go from the local state of the port in the left-hand side to the local state of the revision archive in the left-hand side and then via the local morphism to the local state of the revision archive in the right-hand side or if I go to the local state of the port in the right-hand side via the local morphism and then to the local state of the revision archive of the right-hand side. $\triangle$

A *distributed match* (general Def. 4.3.8 and case study 4.4.11) for a distributed production is given by a distributed morphism, where the network graph morphism and all local morphisms are total. This ensures that the network morphism of the distributed match is a match for the network production of the distributed production and we have a match for the attributed productions of the distributed production given by the total local morphisms of the match. The application of a distributed production at a distributed match can be constructed pointwise by the direct derivation of the network production and the direct derivation of each local attributed production. The morphisms between local states in the derived graph are then uniquely given by the derived graphs for the local states (general Def. 4.3.9, case study Def. 4.4.13).

Example 2.9 (distributed direct derivation)
In the example of a distributed direct derivation in Figure 2.10, the distributed production in Figure 2.9 is applied to a current distributed system state. The total distributed morphism on the left-hand side of the diagram is the distributed match for the production. The current distributed system state consists of two revision archives connected by a port. One revision

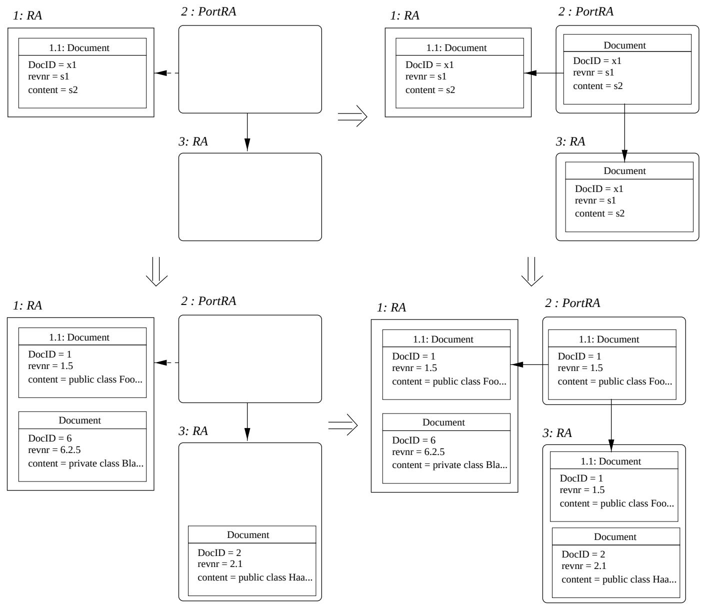


Figure 2.10: *A direct distributed derivation.*

archive contains two revisions, the other one contains one revision. Until now, there does not take place a replication of documents between these two archives, since the port is empty. The distributed match indicates (by the same number 1.1 at the document) that the revision with the revision number 1.5 shall be replicated to the remote revision archive $3 : RA$. The application of the distributed production firstly applies the network production that does not change any topological structure in our example. Then, the three local attributed productions are applied: The attributed production for the revision archive $1 : RA$ preserves the revision 1.5 in the archive $1 : RA$. Due to the local match for this attributed production, the variables $x1$, $s1$ and $s2$ get the values 1, 1.5 and *public class Foo...*, respectively. This variable assignment is used in the attributed productions for the port and the revision archive $3 : RA$. Both attributed productions create a revision with the values of the revision 1.5. The replicated document and the original are connected by the revision in the port. $\triangle$

This pointwise construction of the distributed direct derivation is not possible in general. Since we use attributed graphs as local states, where in particular attributed morphisms are partial, we can apply a distributed production at a given distributed match without any conditions (cf. Theorem 4.4.13). To ensure a pointwise construction of the direct derivation for arbitrary specification techniques for the local states (e.g. graphs with total graph morphisms), we provide a *gluing condition* that has to be satisfied by a given distributed production and a distributed match (cf. Definition 4.3.8). These conditions were already introduced by Taentzer in [Tae96] under the notion of *pushout conditions* and ensure that the direct derivation can be performed pointwise.

2.4.1.1 Negative Application Conditions

Distributed productions describe intuitively how given distributed graphs shall be transformed into derived distributed graphs. A distributed production is applicable if there is a distributed match in a given distributed graph. I introduce now*negative application conditions* for a distributed production that restrict the application of distributed productions to a certain context. I consider here only negative application conditions that forbid certain structures in the context of a match. The more general case is given in Section 4.3.2. For more informations about application conditions see also [Fis98,HHT96,HMTW95]. I start by an example to motivate the use of negative application conditions.

Example 2.10 (negative application condition)
Consider the distributed production in Figure 2.11 that I already introduced in Figure 2.9 and that replicates a revision from one archive to another one. The replication of a revision shall take place only if the revision is not already replicated to the other revision archive over the connected port. This negative condition is specified by a distributed graph that

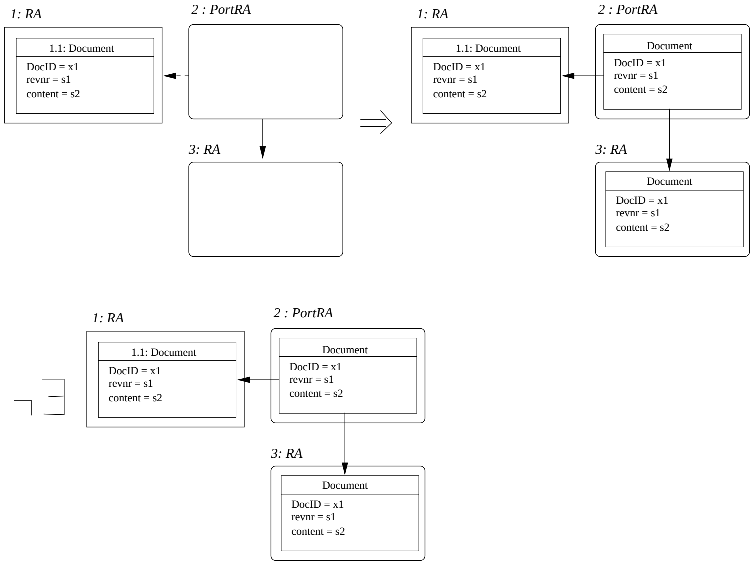


Figure 2.11: *A negative application condition.*

contains the left-hand side of the distributed production and the forbidden structure. I indicate the distributed graph of the negative application condition by $\neg\exists$. The negative application condition of the example is shown in Figure 2.11. The forbidden context consists of the documents in the revision archive and the port. We can apply a production with an application condition at a given match only if the match satisfies the negative application condition in the sense that the situation specified in the application condition does not occur in the context of the match. The match given in Figure 2.10 satisfies the application condition, since there does not exist a document with $DocID = 1$, $revnr = 1.5$ and *content= public class Foo...* in the revision archive $3 : RA$ to that we apply the production. △

Formally, a negative application condition is given by a distributed morphism from the left-hand side of the production to the distributed graph that contains the forbidden structure. Also the satisfaction of an application condition with respect to a match is characterized by a distributed morphism. These points are explained detailed in the formal part of application conditions in Section 4.3.2.

2.4.2 Distributed Graph Transformation for Distributed Systems

A distributed system consists of distributed processes communicating over common ports. For each process its local data state is visible for itself, but it is invisible for remote processes. If a process wants to make visible a part of its local state in order to communicate, the local data are written into a port. The local data of ports are visible for all processes. The local data can then be used for the communication between two processes.
If we consider this principle of visibility in a distributed system, the definition of a distributed production and its application to a distributed graph does not reflect this principle yet. For instance, the left-hand side of the distributed production in Figure 2.10 consists of two revision archives $1 : RA$ and $3 : RA$. In order to apply this production to a distributed graph, we have to find for both revision archives a revision archive in the distributed graph such that the local states of the revision archives in the production occur as local states in the distributed graph. To find this occurrence for both archives at the same time requires a global view on the distributed system. For the application condition in Figure 2.11 we need a global view on the distributed system as well, since it forbids the existence of a revision in the local state of a remote revision archive. To check this negative application condition requires the knowledge of the local state of two revision archives that is only possible with a global view on the distributed system.
Since such a global view does not exist in a distributed system, I restrict distributed productions to so-called *process productions* (Def. 4.3.13). The left-hand side of a process production contains one and only one process node possibly together with some ports. Since the local states of ports are visible for all processes, the process production may refer also to the local states of ports. The right-hand side of the process production contains at most one process node, i.e. a process can be deleted as well. If the right-hand side contains a process node, then it is that one of the left-hand side. The type of the process in the left-hand side determines the type of the process production. Due to this restriction, the local information of one process is sufficient to find a match for a process production. To ensure a local application of a process production, its application condition has to be restricted as well. Therefore, the structures checked in a process application condition refer only to the local state of the process itself and to ports, but not to other processes. Altogether, a process

production ensures that the boundaries of visibility of a process in a distributed system are not violated by finding an occurrence for its application.
Examples of process productions will be given in Figure 2.13- Figure 2.25 when I introduce the process productions for the case study.

One process production of type x specifies one possible action of a process of type x. All possible actions for a process of type x are specified in a *process grammar* of type x consisting of a set of process productions of this type and a *start production* for starting an instance of this process type (Def. 4.3.15). A start production (Def. 4.3.14) is a distributed production, where the left hand side may contain some ports and the right hand side contains possibly some ports and one process node of type x. The right-hand side represents the initial state of the process, the left-hand side the required environment for starting a process of this type. Examples of process grammars are given in the next two subsections, where I present the process grammars of the case study. Since the network type graph of the case study in Figure 2.2 contains two process types, namely RA for revision archives and WS for workspaces, I specify a process grammar for revision archives and one for workspaces.

2.4.2.1 Process Grammar for Revision Archives

The start production for the process grammar of type RA is shown in Fig. 2.12. Its left-hand side is the empty distributed graph, i.e. the creation of a revision archive does not require existing ports. At the same time as the revision archive RA is created, a port of type $PortRA$ is generated connected by a dashed arrow since each archive has to have its own port of type $PortRA$. The revision archive uses this port to replicate its documents. I introduce an initial document in the newly created revision archive, since it simplifies and reduces the following set of process productions for the two process grammars. The document identification number of the document is calculated by the operation *newID* that creates a new unique number for the new document. The operation *newRevNr* creates the corresponding revision number.

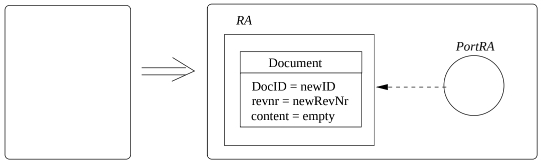


Figure 2.12: *The start production* `start-RA` *for revision archives.*

If a new project partner would like to join the project, a new revision archive has to be created. This can be done by the start production `start-RA`. In this case, the introduction of the new project partner is triggered by the environment of the project. If a new project partner is introduced by existing members of the project, the process production `create-RA` for creating a new revision archive in Figure 2.13 is used. This process production creates a new port of type $PortRA$ and connects itself immediately to this newly created port in order to build a connection for the replication of project documents. The new port is connected

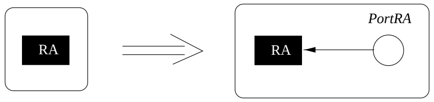


Figure 2.13: `create-RA`: *Create a new revision archive.*

by a solid arrow to indicate that the new port is a port of a remote revision archive. Since process productions contain only the view of one process, the production does not include an explicit creation of the remote revision archive. In order to specify that the creation of the $PortRA$-port implies a new revision archive node as well, the process production in Figure 2.13 has to be synchronized with the start production for revision archives. This will be explained in more detail later on, if I treat the synchronization of processes. I do not provide process productions for the deletion of revision archives. This ensures that the project partners have always access to the documents provided by revision archives.

When a programmer is starting to work on a project at a certain site, a workspace belonging to this programmer must be added to the corresponding revision archive. That means, the creation of new workspaces is triggered by the revision archive. This can be done with the help of the process production `create-WS` in Figure 2.14 that creates a new port of type $PortRAWS$. Just as the process production for creating a new revision archive, the production does not create a new workspace node due to locality reasons. The process production in Figure 2.14 has to be synchronized with the start production for workspaces in Figure 2.21.

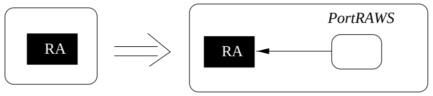


Figure 2.14: `create-WS`: *Create a new workspace.*

When a project partner wants to connect to another revision archive in order to establish a new replication edge, he connects itself to the corresponding port. This has to be done by a solid arrow since the port belongs to a remote revision archive. The negative application conditions of the process production `new-replication` ensure that the process production can be applied only if the port does not belong to the revision archive and there does not exist a replication edge to this port already. This prevents revision archives from importing its own documents.
Revision archives can remove replication edges at any time. The corresponding process production `rem-replication` is shown in Figure 2.16.

If a project partner replicates a document, the document is put into the port of the corresponding revision archive. Connected revision archives are then able to get access to this document. The process production `replicate` for the replication is shown in the Figure 2.17. The attributes of the replicated document are not changed by the replication. Therefore, I depict the document in a more abstract representation. I will use this representation when-

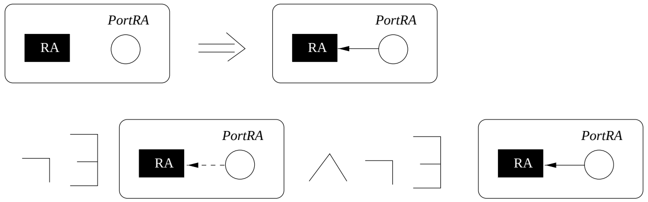


Figure 2.15: `new-replication`: *A new replication is established.*

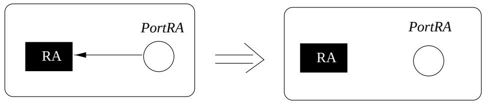


Figure 2.16: `rem-replication`: *Deletion of a replication edge.*

ever the attributes are not changed by the application of the process production. If a change is performed on attributes, I denote the attributes explicitly. The application condition of the production `replicate` ensures that a document is not replicated if this already happened. The dashed arrow between the revision archive and the port ensures that the port belongs to the archive.

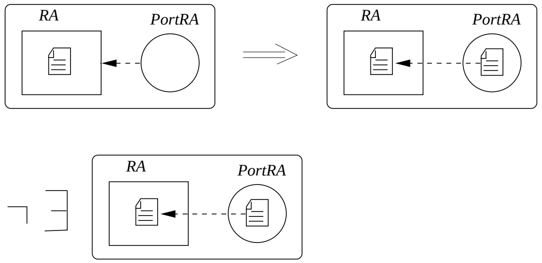


Figure 2.17: `replicate`: *Replicate a document over its port.*

When a revision archive puts a document in its port, remote processes can work with these documents by importing them. This is possible only if there is a replication edge between the revision archive and the port. The edge has to be solid, since only documents form remote revision archives shall be imported and not the own one. The production `import` in Figure 2.18 specifies this move. For the sake of simplicity, I allow that documents are imported several times.

Documents are neither allowed to be deleted nor to be changed in revision archives. Changes of documents have to be done in workspaces and if a workspace is connected to the revision archive by a $PortRAWS$-port, a document can be checked out in the workspace for changing

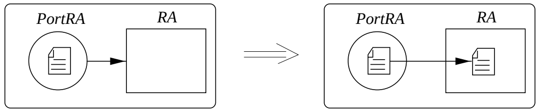


Figure 2.18: `import`: *Import a document from a port.*

it. The process production `check-out-RA` in Figure 2.19 writes the document from the revision archive in the port $PortRAWS$. This takes place only if the document is not already in the port, ensured by the negative application condition. From this port, the document can be put into the workspace.

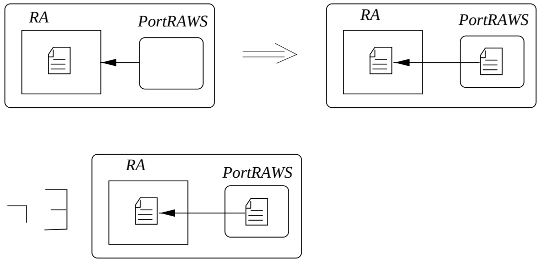


Figure 2.19: `check-out-RA`: *Checking out a document.*

If the programmer has changed the document, she/he checks in a new revision of the document in the revision archive. The process production `check-in-WS` in Figure 2.24 describes this action from the viewpoint of the workspace, the process production `check-in-RA` in Figure 2.20 from the viewpoint of the revision archive. The production creates a new doc-

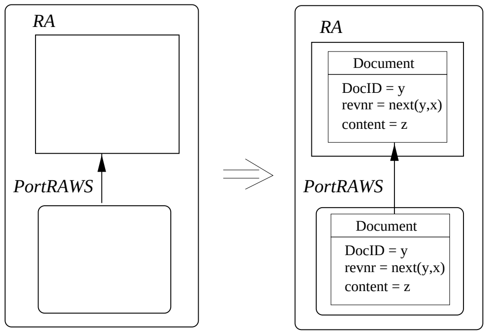


Figure 2.20: `check-in-RA`: *Checking in a document into the revision archive.*

ument in the revision archive and the port $PortRAWS$ that coincide in their attributes.

Table 2.1: *The process grammar for revision archives.*

production	page	figure
`create-RA`	29	2.13
`create-WS`	29	2.14
`new-replication`	30	2.15
`rem-replication`	30	2.16
`replicate`	30	2.17
`import`	31	2.18
`check-out-RA`	31	2.19
`check-in-RA`	31	2.20

The revision numbers of the newly created documents is calculated by an operation *next* that determines the next revision number with respect to a given document and a revision number.
The production may confuse at this point, since the production creates new documents by itself. At a first glance, it would be more reasonable if there is a document in the port of the left-hand side that is imported into the revision archive. However, to ensure that the check in of changed documents from the workspace into the revision archive ensures a correct calculation of the revision numbers, the process production `check-in-RA` has to be synchronized with the process production `check-in-WS` for workspaces. Synchronization requirements will ensure that the document is checked in from the workspace into the port and from the port into the revision archive at the same time, such that the documents in both views coincide. I explain the synchronization of process productions in full detail in step 4 of this chapter.

I conclude with the Table 2.1 for the process grammar $\mathcal{GG}(RA)$ for revision archives.

2.4.2.2 Process Grammar for Workspaces

The start production for the process grammar for workspaces is shown in Figure 2.21. The creation of a workspace does not require ports. Together with an empty workspace, a connected $PortRAWS$-port is created.

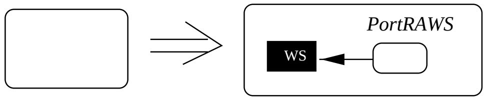


Figure 2.21: `start-WS`*: The start production of a workspace.*

If a programmer would like to change a document, she/he checks out the document from the revision archive. The revision archive has to be put the document in the port that connects the workspace with the revision archive before, what can be done by the process production `check-out-RA` for revision archives shown in Figure 2.19. Therefore, the process production

`check-out-WS` in Figure 2.22 requires a document in the port of its left-hand side. If the

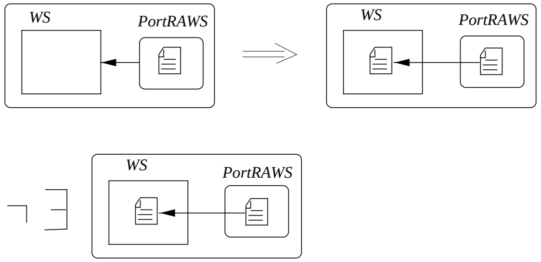


Figure 2.22: `check-out-WS`: *Checking out a document in the workspace.*

document is not already imported from the port, indicated by the negative application condition, the document is put into the workspace and can be changed now from the programmer.

The process production `change-document` for changing a document is given in Figure 2.23. The change of a document implies the generation of a new document and refers only to its

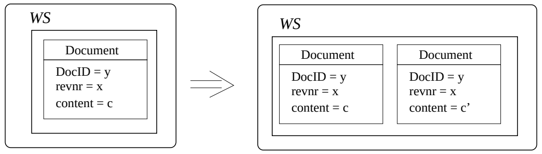


Figure 2.23: `change-document`: *Changing the content of a document.*

contents. The attributes $DocID$ and $revnr$ remain unchanged. Whereas the $DocID$ remains unchanged always, the revision number is changed by a possible check in of the document into the revision archive.

If a programmer has changed a document in the workspace, she/he can decide to check in the document into the revision archive. Not all of the documents in the workspace have to be checked in again, since the workspace may contain also intermediate versions. For checking in a document, the programmer writes the document in the port using the production `check-in-WS` of Figure 2.24. The newly created document in the port has a new revision number calculated from the operation *next* applied to the document identification number y and the old revision number x.

When a programmer finishes, the workspace is deleted by the process production `delete-WS` in Figure 2.25, where the deletion of the workspace also includes the deletion of the port. Please notice, that also all documents in the workspace and the port are deleted as well.
I conclude with the Table 2.2 for the process grammar $\mathcal{GG}(WS)$ for workspaces.

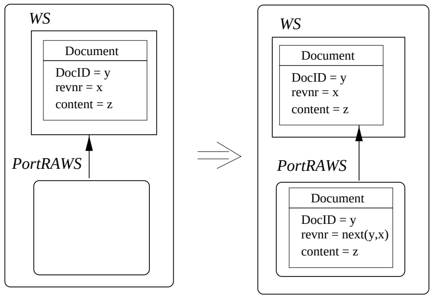


Figure 2.24: `check-in-WS`: *Checking in a document into the revision archive.*

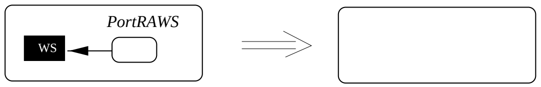


Figure 2.25: `delete-WS`: *Deleting a workspace.*

2.4.2.3 Distributed Grammar

As shown above, the behavior of a process of a certain type is specified by a process grammar containing a set of process productions and a start production of this type. A process grammar of type x can be seen as a pattern for process instances of type x. Each process instance has the "program" given in the process grammar, i.e. its process productions and the start production. To have access to the different process instances of one type and to distinguish them, we provide a *name space* for the process instances of each process type. The name space contains the names for process instances that may occur in the distributed system. The name space is fixed, what is not a restriction, since we can choose the name space arbitrarily. From the viewpoint of object-oriented programming, a process grammar can be seen as a class and the name space contains the names for possible objects of this class.

We combine the process grammar and the name space of each process type of the network type graph in a *distributed grammar* (Def. 4.3.15). The process productions and the start production of a process instance of type x with name i taken from the name space for processes of type x are indexed by the name i as well in order to distinguish the productions of the instances.

Example 2.11 (distributed grammar)
The distributed grammar of the case study of a distributed configuration management system contains the two process grammars $\mathcal{GG}(RA)$ and $\mathcal{GG}(WS)$ in Table 2.1 and Table 2.2, respectively. The name space for revision archives is given by $Ind(RA) = \{1, 2, 3\}$, the name space for workspaces by $Ind(WS) = \{1\}$. Therefore, our distributed system may contain at most three revision archives and at most one workspace. We have chosen for the sake of simplicity two small finite sets. However, any other (possibly infinite) set is possible as well. $\triangle$

Table 2.2: *The process grammar for workspaces.*

production	page	figure
`start-WS`	32	2.21
`check-out-WS`	33	2.22
`change-document`	33	2.23
`check-in-WS`	34	2.24
`delete-WS`	34	2.25

2.5 Step 4: Synchronization of Processes and Distributed Operational Semantics

Until now, the specification of the distributed system does not consider the interaction between process instances. However, certain process instance productions have to be synchronized to ensure a correct system behavior. For instance, the start production `start-WS1` of the workspace instance $WS1$ of the case study has to be synchronized with a process instance production `create-WSi` of one of the revision archive instances Ri ($i = 1, .., 3$), since workspaces are allowed to be created only from an existing revision archive. Another example is the synchronization constraint for the workspace instance production `check-in-WS1` with a revision archive instance production `check-in-RAi` ($i = 1...3$) to ensure a correct revision number of the revision checked in. I continue this section with the synchronization of process instance productions, explain when process productions are synchronizable and introduce the *synchronization relation* on the process instance productions of a distributed grammar, that shows which process instance productions have to be synchronized with each other. The effect of the synchronization relation on the *distributed operational semantics* is discussed afterwards. I introduce a *global semantics* that operates on a global state and uses global distributed productions and a *compositional semantics* that is composed from the local semantics of each process instance using process productions without the need of global states.

2.5.1 The Synchronization Relation

Process instance productions are synchronizable if they have the same *port production*. The port production (Def. 4.3.16) of a process production is constructed by deleting the process node from all distributed graphs of the process production, i.e. also from the graphs of a possible application condition. Therefore, in the distributed graphs of port productions occur only port nodes, but no process node. Synchronizable process instance productions perform the same actions on the ports. They differ only in the transformation of the local process state.

Example 2.12 (port production)
The port production at the top of the Figure 2.26 is the port production of the process instance productions `create-WS1`, `create-WS2` and `create-WS3` of the revision archive instances $RA1$, $RA2$ resp. $RA3$ and the start production `start-WS1` of the workspace instance $WS1$. Therefore, these productions are synchronizable.

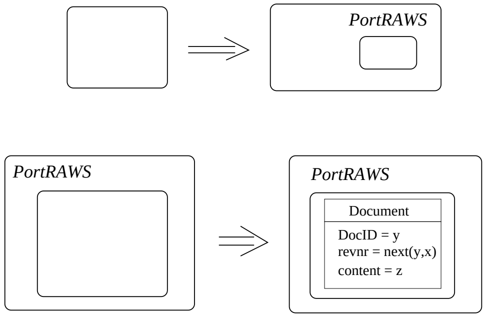


Figure 2.26: *Examples of port productions.*

The port production at the bottom of the figure is a port production for the process instance productions `check-in-RAi` for $i = 1, 2, 3$ in Figure 2.20 and the process instance production `check-in-WS1` in Figure 2.24. That is, the check in of a revision archive is synchronizable with the check in of a workspace. △

A synchronization relation (Def. 4.3.17) for a given distributed grammar determines which productions of process instances have to be synchronized. Only synchronizable process instance productions are related. Please notice, that the synchronization relation relates productions on the instance level, but not on the level of the process grammars. Therefore, the same process production may be synchronized in one instance and may be not in another one. Synchronized productions have to be applied at the same time and concern the same ports and perform the same port transformation.

Example 2.13 (synchronization relation)
The synchronization relation for our case study is defined with respect to the distributed grammar in Example 2.11 and is given by

$$\begin{aligned}\sim_S = \{&(\mathtt{create-RA1}, \mathtt{start-RA2}), (\mathtt{create-RA2}, \mathtt{start-RA3}),\\ &(\mathtt{create-WS3}, \mathtt{start-WS1}), (\mathtt{check-in-RA3}, \mathtt{check-in-WS1})\}\end{aligned}$$

The synchronization of the process productions `create-RA1` of instance $RA1$ and `start-RA2` of instance $RA2$ requires, that the creation of the port $PortRA$ by $RA1$ has to be take place at the same time as the creation of the new instance $RA2$ and the port $PortRA$. The synchronization relation requires that the two process productions perform the same port transformation to the same ports. Hence, the port $PortRA$ created by $RA1$ is the same port as the port $PortRA$ created by the start production of $RA2$. Therefore, the instance $RA1$ creates the instance $RA2$. I explain the effects induced by the synchronization relation in more detail in the next section, when I introduce the operational semantics of a distributed system specification.
Moreover, the synchronization relation requires that the process instance $RA2$ creates the process instance $RA3$. The creation of the workspace instance $WS1$ is triggered by the process instance $RA3$ by using the process production `create-WS3`. Therefore, the workspace

instance is connected to the revision archive instance $RA3$. The productions `check-in-RA3` and `check-in-WS1` have to be synchronized in order to ensure a correct revision number of the revisions checked in. All other process instance productions do not need a synchronization. For instance, the start production `start-RA1` for starting the instance $RA1$ can be performed whenever the production is applicable. △

2.5.2 Operational Distributed Semantics

I consider in this section the operational semantics of a distributed system specification consisting of a distributed grammar and a synchronization relation. I will explain, how the synchronization relation influences the connections between process instances and their interaction. At first, I consider a *global operational semantics* as introduced in [TFKV99,FKT98], where I consider distributed systems with global states. The formal introduction of the global semantics is given in Section 4.3.5. Afterwards, I introduce a *compositional operational semantics* that is equivalent to the global operational semantics (cf. Theorem 4.3.36), but does not operate on a global state and does not use global operations. Its formal introduction can be found in Section 4.3.4.2.

2.5.2.1 A Global Semantics by Amalgamated Productions

The global operational semantic for a distributed system specification is based on the *amalgamation* of process instance productions (Def. 4.3.30). The concept of amalgamated productions is introduced in [CMR+97,Tae97] and specifies the gluing of elementary productions w.r.t. a common subproduction. By definition of the synchronization relation, related process instance productions have a common port production, that is a subproduction of all participating process instance productions. The amalgamation of process instance productions over a common port production glues the process instance productions along the port production. The amalgamated production specifies then the synchronized application of the single process instance productions.

Example 2.14 (amalgamated productions)
Figure 2.27 shows the amalgamated productions constructed form the process instance productions `create-RA1` and `start RA2` (at the top) and `create-WS3` and `start-WS1` (at the bottom). The productions are glued along their common port production. The upper amalgamated production shows that the instance $RA1$ creates the instance $RA2$ and the lower production shows that the instance $RA3$ creates the workspace instance $WS1$.
The amalgamated production for `check-in-RA3` and `check-in-WS1` is depicted in Figure 2.27. The amalgamated production shows the synchronized check-in of a revision from the workspace into the revision archive. △

For the global operational semantics of a distributed system specification consisting of a distributed grammar and a synchronization relation, we consider the set of all process instance productions factorized by the equivalence relation induced by the synchronization relation (Def. 4.3.31). For each equivalence class, we construct the amalgamated production by amalgamating all the members of the equivalence class along their common port production. The global operational semantics is then generated by the derivation sequences starting at the empty distributed graph using the amalgamated productions for the equivalence classes. However, since the start production creates a new instance and we do not

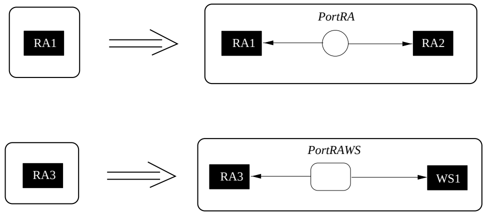


Figure 2.27: *Amalgamated productions for* `create-RA1` *and* `start-RA2` *resp.* `create-WS3` *and* `start-WS1`.

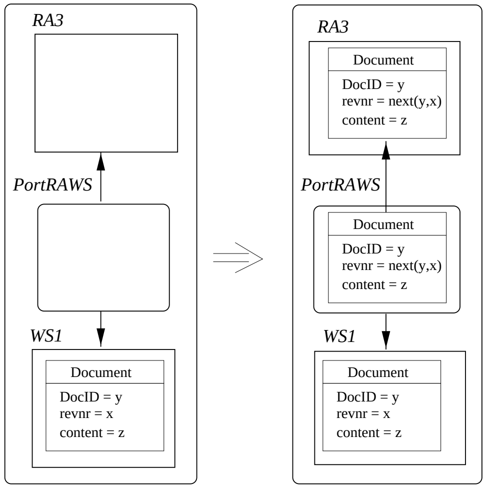


Figure 2.28: *Amalgamated production for* `check-in-RA3` *and* `check-in-WS1`.

allow two or more instances with the same name, we require that the start productions of instances resp. amalgamated productions containing a start production of an instance are applied in one derivation sequence at most once. Therefore, also all process instance productions synchronized with a start production can be applied at most once.
Amalgamated productions are in general global distributed productions, specifying the transformation of several process instances. The amalgamated productions are applied to a distributed graph representing the global state of the distributed system containing several process instances. I require for the match of an amalgamated production in the global distributed system state, that the match is compatible with the instances (Def. 4.3.34). Compatible with the instances means, that the match for an amalgamated production, that is constructed from the process instance productions for instance 1 to n, can be applied only to the process instances 1 to n in the global system graph such that the process instance node i of the amalgamated production is mapped to the process instance node i in the global distributed system graph. This ensures, that we can split the amalgamated match

into local matches for the instances. Then, the global distributed semantics is equivalent to the compositional operational semantics that I will introduce in the next section (cf. Theorem 4.3.36).

Example 2.15 (global operational semantics)
The Figure 2.29 shows a part of the global operational semantics of the case study. The derivation sequence starts at the empty distributed graph, i.e. in the beginning there are no processes in the distributed system. Then, the start production for the instance $RA1$

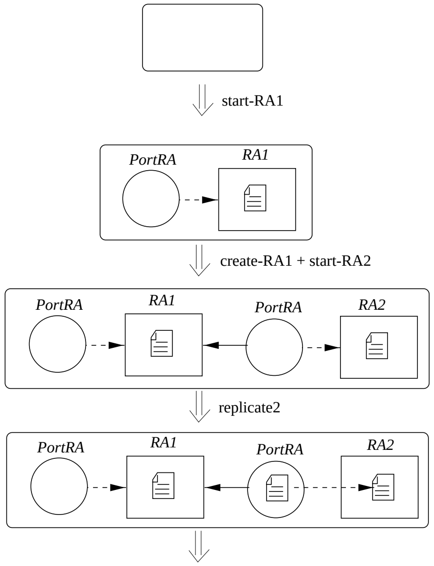


Figure 2.29: *Part of the global distributed semantics.*

is applied. This process instance production can be applied without any synchronization constraints. Since the revision archive instance $RA1$ shall occur only once in the distributed system, the start production for $RA1$ can not be applied anymore. Then, the amalgamated global production `create-RA1+start-RA2` is applied on the distributed system graph. The production starts the instance $RA2$ and connects it with instance $RA1$. Now, one can see that the creation of the port by means of the production `create-RA1` and the synchronization constraint on `create-RA1` induces the creation of the new instance $RA2$. The amalgamated production `create-RA1+start-RA2` can not be applied again, since it would create a second process instance $RA2$. The global distributed system state consists now of two revision archives with their ports $PortRA$. We apply now the process instance production *replicate2* to the instance $RA2$ to put a document into the port. Even if the production can also be applied to the instance $RA1$, I forbid this application, since the amalgamated matches shall be compatible with the instances. Further productions can now be applied to the new global distributed system state. △

2.5.2.2 A Compositional Operational Semantics

The global operational semantic uses global productions got by the amalgamation of process instance productions, that are applied on a global distributed system graph. Since a global view on a distributed system does not exist, I restricted global distributed productions to process productions, that can be applied locally. In order to prevent also the global view for the generation of the operational semantics of a distributed system specification, I introduce a compositional operational semantics that is constructed from the local semantics of the process instances participating in the distributed system. This compositional operational semantics is equivalent to the global operational semantics. In the sequel, I informally introduce the main ideas of the compositional operational semantics for a distributed system specification. A detailed formal explanation can be looked up in Section 4.3.4. The main features of my approach of an operational semantics are

1. the description of the local operational semantics for each process instance in the distributed grammar by its *open process semantics* (Def. 4.3.25) and

2. the composition of the open process semantics according to the synchronization relation (Def. 4.3.27).

Open Process Semantics If we have given a distributed system grammar containing a process grammar and a name space for each process type in the network type graph, I provide for each possible process instance its local operational semantics. This local operational semantics for a process instance describes only the local state of the process instance and do not contain any global state. It is called *open process semantics*, since it considers not only the effects generated by the process productions of the process instance, but also all effects on ports that can be caused by port productions of other process instances of the distributed grammar.

The open process semantics for a process instance is generated by the derivation sequences starting at the empty distributed graph using the process productions of the process instance and the port productions of the process productions of the other instances in the distributed grammar. I start at the empty distributed graph, since an instance can be created at any time during the "life-time" of a distributed system. In particular, in any possible environment, that satisfies the conditions for the creation of the instance. The process instances are created by the application of their start production. Just as in the case of the global operational semantics, I require that the start production of each process instance is applied at most once, since I want to describe the local behavior of a single process instance only. The distributed graphs of the open process semantics for a process instance contain at most one process node, namely that one for the corresponding instance. Altogether, the open process semantics for a process instance describes the distributed system from the local viewpoint of this single process instance, where a maximal environment is assumed in the sense that all possible effects on ports are considered.

Example 2.16 (open process semantics)
In the distributed system grammar of our case study, we have three revision archive instances $RA1$, $RA2$ and $RA3$ as well as a workspace instance $WS1$. The left-hand side of the Figure 2.30 shows a possible sequence of transformations in the open process semantics of the

revision archive instance $RA1$, on the right-hand side there is a possible sequence of transformations in the open process semantics of the revision archive instance $RA2$. Both sequences

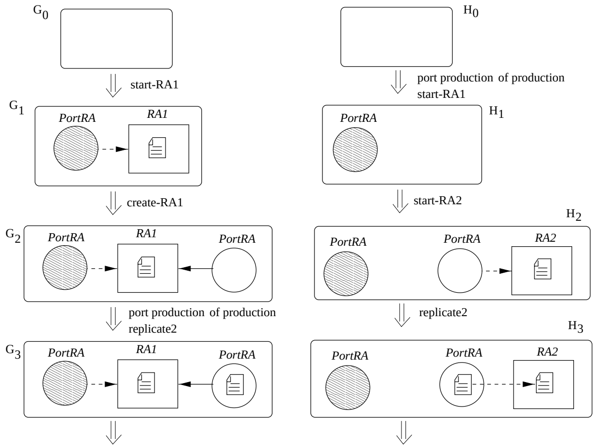


Figure 2.30: *Parts of the open process semantics of the revision archive instances* $RA1$ *(left-hand side) and* $RA2$ *(right-hand side).*

start at the empty distributed graph. In the first step of the transformation sequence of the revision archive $RA1$, the instance is created by the start production `start-RA1`. From this point, the start production cannot be used anymore for further transformations to ensure a local view of instance $RA1$. In a second step, the instance $RA1$ creates a new port by the production `create-RA1`. The third step of $RA1$ applies a port production of another instance, namely the port production of *replicate*2. This port production creates a new document in the port on the right-hand side. This application of port productions of other instances motivates the name *open*, since the creation of a document in the port is based on the environment and not of the instance $RA1$ itself. The instance $RA1$ can then apply further process productions and port productions.

The transformation sequence for the instance $RA2$ applies in the first step the port production of `start-RA1` and creates a port. This shows, that a process instance does not have to be active from the beginning of the distributed system. The process instance $RA2$ is activated in the second step. In the third step, the initial document is replicated to the port by the production `replicate2`.

The distributed graphs in the open process semantics for $RA1$ show only the local state of the process instance $RA1$ and the port environment, that is visible for all process instances in the distributed system. The distributed graphs of the open process semantics for $RA2$ show only the local state of $RA2$ and the port environment. $\triangle$

I show next, how the open process semantics for each instance in the distributed grammar can be used to construct the compositional distributed semantics.

Distributed Semantics: Composition of the Open Process Semantics The compositional operational semantics of a distributed system, specified by a distributed grammar and a synchronization relation, is composed from the open process semantics of its process instances.
The composition of two open process semantics with respect to process instances $P1$ and $P2$ composes the distributed graphs and the transformations between distributed graphs. The composition does not necessarily reflect all distributed graphs resp. transformations of an instance and is performed as follows (cf. Def. 4.3.27): A distributed graph $\hat{G}$ of $P1$ can be composed with a distributed graph $\hat{H}$ of $P2$ if the distributed graphs restricted to the ports are equal. The actual composition of the two distributed graphs is their disjoint union, where the ports are identified afterwards. The synchronization relation influences the composition of the transformation steps. A transformation step between two distributed graphs $\hat{G}_1$ and $\hat{G}_2$ with the production p_1 at match $\hat{m}_1$ in $P1$ can be composed with a transformation step between two distributed $\hat{H}_1$ and $\hat{H}_2$ with the production p_2 at match $\hat{m}_2$ in $P2$ if

- the distributed graphs $\hat{G}_1$ and $\hat{H}_1$ as well as $\hat{G}_2$ and $\hat{H}_2$ can be composed,
- if p_1 and p_2 are process instance productions then they are in the synchronization relation, if p_1 is a port production then p_1 is the port production of p_2, if p_2 is a port production then p_2 is the port production of p_1, and
- the matches $\hat{m}_1$ and $\hat{m}_2$ restricted to the ports are equal.

Process instance productions can be related by the synchronization relation if they have the same port production. If the matches of related process instance productions additionally coincide on the ports and the port graphs of the transformations are equal, the two transformation steps in $P1$ and $P2$ coincide in the port transformation, i.e. they transform the same elements in the same ports in the same way. The composition of the two transformation steps composes the distributed graphs $\hat{G}_1$ and $\hat{H}_1$ as well as $\hat{G}_2$ and $\hat{H}_2$. The composition of the transformations is again their disjoint union, where the ports are identified. The composed transformation step on the ports is equal to the transformation in $P1$ and $P2$ and the transformation on the processes coincide in $P1$ resp. $P2$.

Example 2.17 (composition)
I consider the composition of the two transformation sequences in Figure 2.30. The distributed graphs $\hat{G}_i$ and $\hat{H}_i$ for $i = 0, .., 3$ can be composed, since they are equal on the ports. I have indicated the ports equal in both distributed graphs $\hat{G}_i$ and $\hat{H}_i$ by the common pattern (grey and white, respectively). In addition, the single transformation steps of $RA1$ and $RA2$ can be composed, since the transformations on the ports are equal as well. The synchronization relation is satisfied in the transformation steps 1 and 3 since one instance uses the port production of the process instance production used in the other instance. In the transformation step 2, the instance $RA1$ uses the production `create-RA1` and the instance $RA2$ applies the start production `start-RA`. This pair is in the synchronization relation of the case study, such that the transformation steps can be composed. The matches restricted to the ports coincide in all transformation steps. The composition of the sequences is isomorphic to the global distributed semantics in the Figure 2.29. △

Chapter 3

Specifying and Checking Graph-Interpreted Temporal Formulas in Distributed Systems

This chapter is concerned with the graphical specification of properties of a distributed system and the possibilities for their checking. The properties considered in this work include safety as well as liveness properties. Safety properties ensure that nothing bad happens in the distributed system. This means for our case, that the distributed graphs representing the states of the system have always a certain structure. Liveness properties ensure that the system behaves well. This means in our case, that a certain distributed graph is eventually constructed. Safety and liveness properties are expressed by *temporal formulas*. Temporal logic is a well established concept to specify those properties and to reason about the dynamical behavior of systems. There exists a vast amount of work in the area of temporal logic that can be used for checking formulas. To make use of these results I combine distributed graph transformation with propositional temporal logic. This combination enables to graphically specify the temporal formulas for the distributed graph model. The main idea of the combination is to make use of the results for checking temporal formulas and not to develop new techniques especially customized to distributed graph transformation. The concept for checking formulas on which I will in particular focus in this work is model checking. Tools, so called model checkers, are provided to examine if a certain temporal formula is valid in a given temporal model. Most of the model checker require a finite temporal model. Therefore, I provide a construction to collapse a possibly infinite temporal model to a finite one, called *typical model*. Formulas, that are true in the typical model are true in the original infinite model as well. When a formula is false in the typical model, however, the formula may be true in the original model. In this case, the checking of the formula has to be done by the designer.
Checking of properties in a distributed system is in particular a problem since no global system state can be assumed in that the property can be checked. However, global properties may be of interest to ensure the desired behavior of the distributed system. I will show, how the typical model can be used to reason from local satisfaction of local formulas to global satisfaction of global formulas. Local formulas are temporal formulas, that can be checked locally for one process.

The remainder of this chapter is organized as follows: The case study of the distributed configuration management as introduced in the previous chapter will be used throughout the chapter to illustrate the concepts. Therefore, I start with the case study and introduce the properties that I am going to check. Afterwards, the concepts for specifying and checking these properties are introduced. The graphical specification of temporal formulas requires knowledge in propositional temporal logic, such that I introduce it before. After I have explained how temporal properties can be graphically specified by graph-interpreted temporal formulas, I provide the construction of the typical model. I conclude by using a model checker for checking a global temporal formula of our case study.

3.1 The Main Properties of the Case Study

One of the most important rules in distributed configuration management says that no information gets ever lost. This is based on practical requirements where customers may need old versions of programs that they bought years ago or to detect errors in the development process. To ensure the availability of documents, they are not allowed to be deleted in revision archives. Moreover, to provide a consistent and current set of project documents for each project partner, documents have to be replicated to the revision archives of the project partners as well. Therefore, each project partner must eventually sent its documents over a port to a remote project partner that receives the documents.
The properties that I am going to prove for the distributed graph model of Chapter 2 are therefore:

Main properties:

1. Each document is preserved in the distributed system.

2. Each document in a revision archive is replicated to a remote revision archive

Please notice, that these properties are global, since they refer to the whole distributed system respectively to several local components.

In the sequel I show how the main properties are expressed by graph-interpreted temporal formulas and what has to be done to check them. A graph-interpreted formula is a temporal formula, where the propositional variables are interpreted by distributed graphs equipped by variable assignments. The basic idea of checking a (global) formula can be summarized as follows:

1. Reducing the global property to local properties.

2. Reducing the distributed system to sub-systems in which the local formulas can be checked locally. The satisfaction of a local formula in the sub-system has to imply satisfaction of the local formula in the distributed system.

3. Checking the local formulas in the corresponding sub-systems.

4. Reason from local satisfaction to global satisfaction.

Whenever the underlying model of the distributed system is finite, i.e. the number of states in the distributed system is finite, each graph-interpreted formula can be checked by means of a model checker (see e.g. in [Eme90]). I present later on a model checker, called $STeP$, introduced in [Man94], as a concrete model-checker used for checking a graph-interpreted temporal formula of the case study.
If the underlying model is infinite, most of the checking techniques are not applicable anymore. Research in this area was and is done to reduce the complexity of the model or

to restrict infinite models to finite ones [Bra92,BS90,BCMD90]. I provide in this work a construction for collapsing an infinite model to a finite one with respect to a given set of temporal formulas. This finite model is typical for the set of formulas in the sense that whenever a formula is satisfied in the typical model then it is satisfied in the infinite original model. If the formula is not satisfied in the typical model, the formula could be satisfied in the infinite model. In this way, the typical model supports the checking of temporal formulas in an infinite model, but does not provide an exhaustive checking method. However, the typical model can be used for checking local temporal formulas in the reduced sub-system as well as for the reasoning from local satisfaction to global satisfaction. The latter point will be the example for explaining the typical model, since it helps to solve the problem of the non-existence of a global state.

3.2 Graph-interpreted Temporal Formulas

I introduce now graph-interpreted temporal formulas, that are a mixture of temporal formulas in *linear-time propositional temporal logic* and distributed graphs equipped by a variable assignment. In linear-time propositional temporal logic the nontemporal part of the logic is just classical propositional logic over a set of propositional variables and the course of time is linear in the sense that for each moment there is only one possible future moment. Various other kinds of temporal logics are introduced in the literature. A survey is given in [Eme90,Sti92].
Graph-interpreted temporal formulas are propositional temporal formulas, where the propositional variables of the formula are substituted by distributed graphs equipped by a variable assignment, that is, a distributed morphism with source graph $\hat{X}$. The distributed graph $\hat{X}$ represents the *graphical variables* of the graph-interpreted temporal formula and the distributed morphism from the graph $\hat{X}$ in a distributed graph is called *assignment for* $\hat{X}$. In the following we explain the meaning of graph-interpreted temporal formulas informally. The formal definition of the temporal logic used in this part and the graph-interpretation is given in Section 5.1. This informal part shall be used to enable a software developer to specify graphically the properties of her/his distributed system by graph-interpreted formulas. In particular, I answer the following two questions:

1. How can I write a graph-interpreted temporal formula?

2. What does a graph-interpreted temporal formula mean, i.e. when is it satisfied?

3.2.1 How to Specify Graph-interpreted Temporal Formulas

This section deals with the syntax of graph-interpreted formulas that are build up from temporal formulas in propositional temporal logic and assignments for a given distributed graph $\hat{X}$ of graphical variables. I first consider assignments and give then a brief introduction into the syntax of temporal formulas and conclude with the combination of assignments and temporal formulas resulting in graph-interpreted temporal formulas.

3.2.1.1 Assignments

An *assignment* (Def. 5.3.1) for a graph $\hat{X}$ is a distributed graph morphism as introduced in the previous chapter with source graph $\hat{X}$. Such a distributed morphism can be seen as a pattern for a whole class of distributed morphisms. This class contains all distributed morphisms that have the same source graph $\hat{X}$ and the target graph provides at least the structure that is given in the target graph of the assignment.

Example 3.1 (assignment)
In Figure 3.1 an example of an assignment $\hat{a}$ for a distributed graph $\hat{X}$ is shown. The

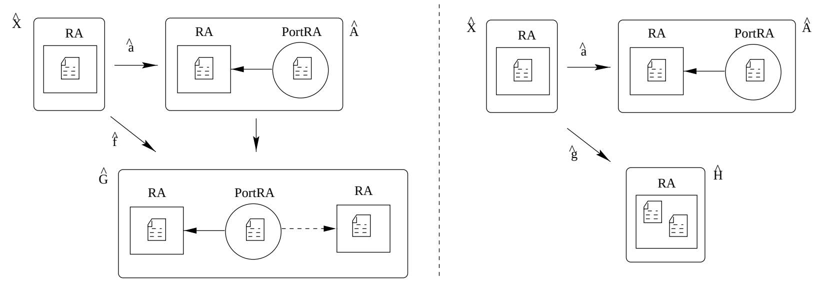


Figure 3.1: *An assignment is a pattern for distributed graph morphisms: The morphism $\hat{f}$ on the left-hand side is in the class, that one on the right-hand side is not.*

distributed graph $\hat{X}$ contains a revision archive together with one document. The target graph $\hat{A}$ of $\hat{a}$ describes the pattern for the target graphs of the class. It requires a connected port together with a document. On the left-hand side of the figure we have a distributed graph morphism $\hat{f}$ that satisfies the pattern of the assignment. On the right-hand side we have a distributed graph morphism that is not in the class of distributed graph morphisms defined by the assignment, since the target graph does not satisfy the pattern of a connected port. △

As one can see, for the membership of a distributed morphism to the class generated by an assignment it is necessary to find an embedding from the assignment in the distributed graph morphism. In this way I define a relation on distributed morphisms, called *embedding relation* (Def. 5.3.5), in the sense that a distributed morphism $\hat{a} : \hat{X} \to \hat{A}$ is related to a distributed morphism $\hat{b} : \hat{X} \to \hat{B}$ if $\hat{a}$ can be embedded in $\hat{b}$. Embedding of $\hat{a}$ in $\hat{b}$ means to embed the domain of $\hat{a}$ in the domain of $\hat{b}$, i.e. $\hat{b}$ is defined at least for the elements where $\hat{a}$ is defined, and to find a total distributed morphism $\hat{f} : \hat{A} \to \hat{B}$ commuting the resulting diagram, i.e. $\hat{b}$ provides at least the elements occurring in the target graph of $\hat{a}$ and possibly some more. The embedding relation on distributed graph morphisms will be used for modeling satisfaction of temporal formulas later on.

Example 3.2 (embedding relation)
Examples for the embedding relation are already shown in Figure 3.1. Since all distributed morphisms $\hat{a}$, $\hat{f}$ and $\hat{g}$ are total, their domains coincide. The assignment $\hat{a}$ is related to $\hat{f}$, since we can find an embedding of $\hat{A}$ in $\hat{G}$, but $\hat{a}$ is not related to $\hat{g}$ since there is no embedding of $\hat{A}$ in $\hat{H}$. This example shows the special case of a total assignment and total distributed morphisms $\hat{f}$ and $\hat{g}$. The Figure 3.2 shows an example with partial assignments for a distributed graph $\hat{X}$ that contains a revision archive and a port with a document in each

of the network nodes. The distributed graph morphisms are shown as spans $\hat{X} \hookleftarrow \hat{X}_a \to \hat{A}$ and $\hat{X} \hookleftarrow \hat{X}_b \to \hat{B}$, where the domain of the distributed morphism $\hat{a}$ is given by the distributed graph $\hat{X}_a$ and the domain of $\hat{b}$ by the distributed graph $\hat{X}_b$. The assignment $\hat{a}$

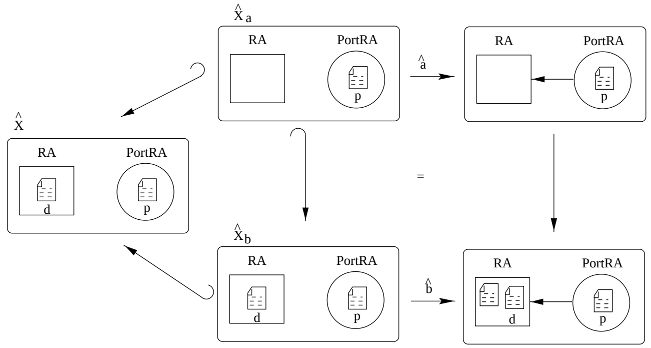


Figure 3.2: *The upper distributed morphism can be embedded in the lower one, but not vice versa.*

can be embedded in $\hat{b}$ since $\hat{X}_a$ is a subgraph of $\hat{X}_b$ and there is a total distributed graph morphism from the target graph of $\hat{a}$ in that one of $\hat{b}$ such that the diagrams commutes. An embedding of $\hat{b}$ into $\hat{a}$ is not possible, since neither $\hat{X}_b$ is a subgraph of $\hat{X}_a$ nor there is a total distributed graph morphism from the target graph of $\hat{b}$ to that one of $\hat{a}$.

3.2.1.2 Combination of Graph Transformation and Temporal Logic

Temporal formulas specify by means of special *temporal operators* how truth values of assertions change over time. The temporal operators used in this work (cf. Def. 5.1.1) include

- $\Diamond\Phi$, read as *sometimes* Φ, which is true now if there is a future moment at which the temporal formula Φ becomes true,
- $\Box\Phi$, read as *always* Φ, which is true now if Φ is true at all future moments and
- $\bigcirc\Phi$, read as *next* Φ, which is true now if Φ is true in the next moment.

The following figure shows the intuition of the temporal operators again in a graphically way. It shows the course of time and the meaning of the temporal operators, where a black circle shows that the formula Φ is true.

The set of *graph-interpreted temporal formulas* over $\hat{X}$, short formulas, is the least set of formulas generated by the following rules (cf. Def. 5.1.1 and Subsection 5.3.9):

- each assignment for $\hat{X}$ is a formula,
- if Φ is a formula then $\neg\Phi$ is a formula,
- if Φ_1 and Φ_2 are formulas then $\Phi_1 \wedge \Phi_2$ is a formula and

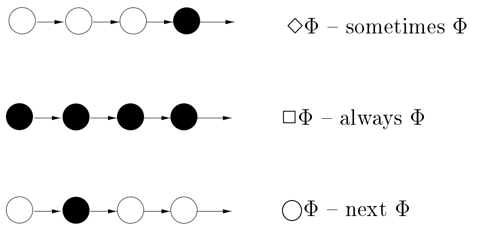


Figure 3.3: *Intuition for linear-time operators.*

- if Φ is a formula then $\Diamond\Phi$ and $\bigcirc\Phi$ are formulas.

The operators $\neg$ and $\wedge$ are the usual propositional operators for negation and conjunction and $\Phi_1 \vee \Phi_2 := \neg(\neg\Phi_1 \wedge \neg\Phi_2)$ and $\Phi_1 \rightarrow \Phi_2 := \neg\Phi_1 \vee \Phi_2$ are the usual classical abbreviations for disjunction and implication, respectively. Furthermore, $\Box\Phi := \neg\Diamond\neg\Phi$ defines the always operator. The distributed graph $\hat{X}$ is considered as a set of graphical variables.

Example 3.3 (graph-interpreted temporal formula)
I specify now the main properties of our case study by graph-interpreted formulas. The first property requires that a document gets never lost in the distributed system. This is ensured by the graph-interpreted formula in Figure 3.4. The source graph $\hat{X}$ consists of two revision archive nodes and one *PortRA*-node. Each of the network nodes contains one document. The atomic assignments for $\hat{X}$ from that the graph-interpreted formula is build up, are q_1 and q_2. The assignment q_1 is defined for the left revision archive of $\hat{X}$ and its document, undefined otherwise. The assignment q_2 is defined only for the network node of the revision archive, but not the document and the other network nodes. The intended

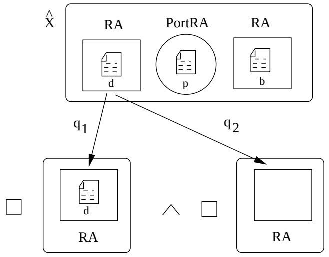


Figure 3.4: *A revision archive is always preserved and all documents in revision archive are preserved as well.*

meaning of the formula is, that a document in the revision archive is always preserved and the archive is preserved as well. The formula is shown again at the top of Figure 3.5 in the notation used up to now for graph-interpreted formulas. In the following, I assume that all graphical constraints are graphical constraints over the distributed graph $\hat{X}$ in Figure 3.4. Therefore, I omit the explicit representation of the graph $\hat{X}$ and notate the definition of the assignments by the indices at the objects in the target graphs of the assignments.

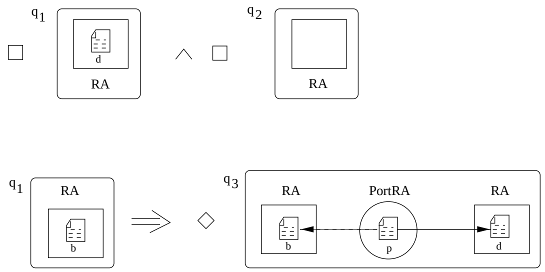


Figure 3.5: *A revision archive is always preserved and all documents in revision archive are preserved as well. All documents in a revision archive are eventually replicated to a remote archive*

The second main property requires that all documents are eventually replicated. This property is specified by the graph-interpreted formula at the bottom of Figure 3.5. Its intended meaning is as follows: Whenever there is a document in a revision archive then the document is eventually sent over the port of the revision archive to a remote revision archive. To express that the left revision archive (with index d) and the right revision archive (with index b) are different, I inserted two revision archives into the distributed graph $\hat{X}$. $\triangle$

3.2.1.3 Global and Local Temporal Formulas

The checking of temporal formulas in a distributed system is especially a problem because of the lack of a global state. Therefore, before I am going to treat the satisfaction of graph-interpreted temporal formulas and the possibilities to check them, I distinguish between *local* and *global* graph-interpreted temporal formulas (Def. 5.3.17). Local temporal formulas do not require a global state for their checking. The properties specified by a local formula refer only to one process and its visible environment. A global formula is a formula, where the information of one local system is not sufficient to check the formula.
I clarify these two terms by the graph-interpreted formulas of the case study in Figure 3.5. The formula at the top of the figure is a local formula, since it requires a property for only one revision archive. It is possible to decide locally for each instance of a revision archive in the distributed system if a document is always in the archive and if an archive always exist. The local check of satisfaction is not possible for the temporal formula at the bottom of the Figure 3.5. This formula requires the existence of a remote revision archive and a document in its local state. Since a revision archive is only able to get access to other revision archives over ports and a revision archive is not allowed to see local states of remote revision archives, the local information of one revision archive is not sufficient any more for checking this formula.
However, to check global properties is desired to ensure a correct behavior of a distributed system. Therefore, the global formula is divided into several local formulas. This reduction to local formulas has to be done by the designer. However, I provide a construction with respect to a set of graph-interpreted formulas that enables to check automatically whether the global formula follows from the satisfaction of the local formulas. We will come back to

this construction later on in Section 3.3. Here, I present the reduction of the global formula of our case study to local formulas.

Example 3.4 (Local Formulas)
The global formula at the bottom of Figure 3.5 is reduced to the two local formulas in Figure 3.6. The upper local formula requires that each document in a revision archive is

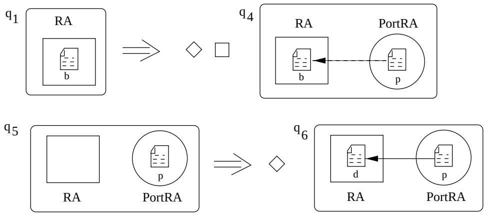


Figure 3.6: *Each document is replicated eventually and each presented document is imported eventually.*

eventually replicated in the port belonging to the archive (indicated by the dashed edge) in the distributed graph next to q_4. Moreover, if this took place, the document remains always in the port to ensure that all remote archives have access to this document including those one that are possibly created in the future.
The lower local formula requires that a revision archive imports a document if it is possible. This ensures that revision archives get all documents that are replicated. $\triangle$

3.2.2 Satisfaction of Graph-interpreted Formulas

We have seen in the previous section, that the atomic building blocks of a graph-interpreted formula are assignments for a distributed graph $\hat{X}$. The satisfaction of graph-interpreted formulas is now defined with respect to *runs*, that are infinite sequences of assignments for $\hat{X}$ (Def. 5.2.2 and Def. 5.3.5). These sequence of assignments can be generated by a derivation sequence (see Subsection 3.2.2.1), but it has not to be necessarily. The satisfaction of a graph interpreted formula by a run is mainly based on the embedding relation introduced in the previous section (see also the Example 3.2).

Let $\sigma = (\hat{a}_0\hat{a}_1\hat{a}_2....)$ be a run of assignments for $\hat{X}$, where $a_i : \hat{X} \to \hat{A}_i$ $(i \in \mathbb{N})$. Then, σ satisfies a graph interpreted temporal formula Φ, denoted by $\sigma \models \Phi$, if one of the following cases is verified:

- if Φ is a single assignment $\hat{a}$ for $\hat{X}$ then $\sigma \models \hat{a}$ if and only if $\hat{a}$ can be embedded into $\hat{a}_0$ according to the embedding relation,
- $\sigma \models \neg\Phi$ if and only if $\sigma \not\models \Phi$,
- $\sigma \models \Phi_1 \wedge \Phi_2$ if and only if $\sigma \models \Phi_1$ and $\sigma \models \Phi_2$,
- $\sigma \models \bigcirc\Phi$ if and only if the suffix $\sigma|_1 = (\hat{a}_1\hat{a}_2...)$ satisfies Φ and

- $\sigma \models \Diamond\Phi$ if and only if there is an $i \in \mathbb{N}$ such that the suffix $\sigma|_i = (\hat{a}_i\hat{a}_{i+1}...)$ satisfies Φ.

Example 3.5 (satisfaction of graph-interpreted formulas)
As an example we consider the run $\sigma = (\hat{a}_0\hat{a}_1\hat{a}_2\hat{a}_2...)$ given in Figure 3.7 and the formula at the bottom of Figure 3.5 build up from the assignments q_1 and q_3. The assignment $\hat{a}_0$ is defined only for the revision archive with document b, $\hat{a}_1$ is total, i.e. it is defined for the revision archives with document b and d as well as the port with document p, and the assignment $\hat{a}_2$ is undefined for all nodes in $\hat{X}$. The premise of the implication of the graph-

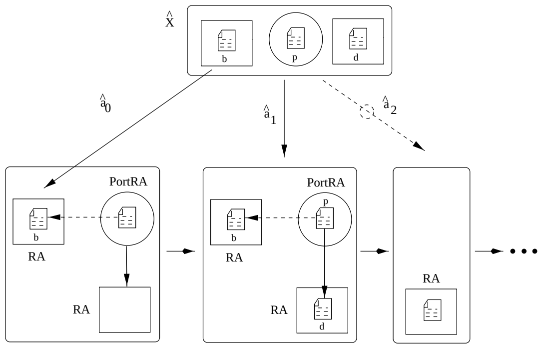


Figure 3.7: *A run of assignments.*

interpreted formula at the bottom of Figure 3.5 is satisfied, since the assignment q_1 can be embedded into the first assignment $\hat{a}_0$ of the run σ. The domains of a_0 and q_1 are equal and the assignment a_0 provides the revision archive with document b required by q_1. Therefore, we have to check the implication of the formula for its satisfaction, i.e. wether we can find an assignment in the run, where the assignment q_3 can be embedded. The search is successful, since q_3 can be embedded into $\hat{a}_1$. The domain of q_3 is equal to the domain of a_1 and a_1 provides the structure required by q_3. Therefore, the run σ satisfies the graph-interpreted temporal formula at the bottom of Figure 3.5. $\triangle$

3.2.2.1 Graph Transition Systems and their Generation by Graph Grammars

In general, runs do not grow on trees, but are given in a *temporal graph transition system* over $\hat{X}$ (Def. 5.3.9). A temporal graph transition system over $\hat{X}$ consists of a set of assignments for $\hat{X}$, called *states*, a transition relation on the states and a set of runs. The set of runs is build up from the states of the temporal graph transition system and the transition relation. Formally, a temporal graph transition system is a transition system with a special set of states (cf. Chapter 5). A graph transition system satisfies a graph-interpreted temporal formula if each of its runs satisfies the formula.
Each graph grammar, consisting of a set of productions and an initial graph, generates a temporal graph transition system over $\hat{X}$ with respect to an assignment for $\hat{X}$ into the initial distributed graph. The set of derivation sequences of the grammar generates the graph transition system as follows (Def. 5.3.12): If we have given a derivation sequence

$\rho = (\hat{G}_0 \Rightarrow \hat{G}_1 \Rightarrow \hat{G}_2 \Rightarrow ...)$ starting at the initial distributed graph $\hat{G}_0$, we continue the assignment in $\hat{G}_0$ to an assignment in $\hat{G}_1$ along the direct derivation $\hat{G}_0 \Rightarrow \hat{G}_1$. We continue the assignment in $\hat{G}_1$ to an assignment in $\hat{G}_2$ along $\hat{G}_2 \Rightarrow \hat{G}_3$ etc.

An assignment $\hat{b} : \hat{X} \to \hat{H}$ is a continuation of an assignment $\hat{a} : \hat{X} \to \hat{G}$ along a direct derivation $\hat{G} \Rightarrow \hat{H}$ if $\hat{b}$ is compatible with deletion, preservation and creation of items in the following sense (Def. 5.3.11): $\hat{a}$ and $\hat{b}$ have to agree on all elements that are preserved from $\hat{G}$ to $\hat{H}$, only elements of $\hat{G}$ that are deleted by the step may be forgotten from $\hat{a}$ to $\hat{b}$, and only those elements that are new in $\hat{H}$ may be used to extend the assignment $\hat{b}$ with respect to $\hat{a}$. In this way, the distributed graph $\hat{X}$ keeps track of the objects during the direct derivation.

Example 3.6 (continuation)
The example in Figure 3.8 shows the continuation of the assignment $\hat{a}$ along the direct derivation $\hat{G} \Rightarrow \hat{H}$. The direct derivation preserves the revision archive and its document as

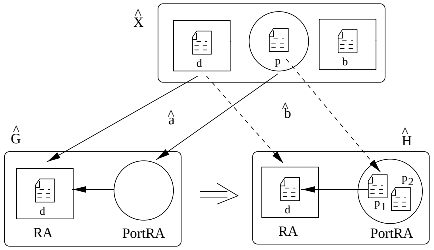


Figure 3.8: *A continuation of an assignment along a direct derivation.*

well as the *PortRA*-node, where two new documents are added. Since $\hat{a}$ is defined on the revision archive and its document, the continuation $\hat{b}$ has to be defined on these objects in the same way. For the same reason, $\hat{b}$ has to be defined on the *PortRA*-node. The two new documents in the *PortRA*-node may be used to define the assignment $\hat{b}$ on the document p in the graph $\hat{X}$. In the example, the document p in $\hat{X}$ is mapped to p_1. However, mapping the document p in $\hat{X}$ to p_2 is also a valid continuation. The assignment may be also undefined for p. Therefore, one assignment can generate a set of assignments for its continuation along one direct derivation. $\triangle$

The graph transition system generated by a graph grammar and an assignment $\hat{a}_0$ in the initial distributed graph of the grammar has as states all assignments got by the continuation of $\hat{a}_0$ on the derivation sequences of the grammar. There is a transition between two assignments, if one assignment is a continuation of the other one along a direct derivation contained in a derivation sequence. The set of runs of the generated graph transition system is build up from the continuations of $\hat{a}_0$ on the derivation sequences. Since runs have to be infinite and a derivation sequence may be finite, we extend the sequence of assignments got by a finite derivation sequence to an infinite one by repeating the last assignment infinitely often.

3.3 Checking Satisfaction of Graph-Interpreted Temporal Formulas

This section is concerned with the verification of distributed systems specified by distributed graph transformation. In this work, I am mainly interested in the *model-checking* approach that was begun by Clarke and Emerson [CE81,CES86] as well as Quielle and Sifakis [QS81]. The idea model-checking is based on is to consider a system as a model for the temporal logic and check whether the model satisfies a given formula. The emphasize of model-checking lies on the checking aspect: rather than performing proofs, one has an algorithm which takes the model and the formula as input and returns a yes/no answer. First model-checkers are given in [CE81,CES86], I consider a model checker called *STeP* (Stanford Temporal Prover) developed by the group of Manna introduced in [Man94]. The question, if a model satisfies a given formula is decidable for propositional temporal logic and models with a finite set of states, since one can do an exhaustive search through the paths of the finite model. And this is mainly the idea how model-checkers work together with some sophisticated optimization strategies to reduce the state space. However, the model checkers can not be used any more for infinite models.

Moreover, the model checkers expect the model as input. Unfortunately, in the context of distributed systems we do not have a global model for the whole distributed system.

Therefore, there are two problems. That are infinite models and the non-existence of a global model. I propose the construction of a so-called *typical graph transition system* induced by a set of graph-interpreted temporal formulas, that collapses a possibly infinite system to a finite one and that allows to reason form local satisfaction to global satisfaction. The typical graph transition system supports the checking process of formulas in distributed systems, but do not solve the two problems completely. Before I am going in more detail with respect to this statement, I firstly introduce the construction of the typical graph transition system.

3.3.1 The Typical Graph Transition System

We have seen in a previous section, that assignments for a distributed graph $\hat{X}$ are the atomic building blocks of a graph-interpreted temporal formula. If we have given a set of graph-interpreted formulas, we have also a set of assignments for $\hat{X}$, namely those one used for building the formulas. For this set of assignments I construct a temporal graph transition system that is *typical* for the set of assignments in the sense that it satisfies exactly those graph-interpreted formulas that are build up from assignments of this set and that are satisfied by all temporal graph transition systems (cf. Corollary 5.2.6).

Construction of Least Upper Bounds Given a set of assignments, the idea of the construction of the typical model for this set is to build for every possible subset of assignments a distinguished assignment which implements this subset, i.e., the distinguished assignment can be embedded in exactly those assignments, where all the assignments of the subset can be embedded. The distinguished assignment for a set of assignments is called the *least upper bound* for this set. The name is motivated by the fact, that the embedding relation introduced in Section 3.2.1.1 forms a pre-order (cf. Prop. 5.3.6) and the distinguished assignment is in fact the least upper bound for a set of assignments with respect to this pre-order (cf. Prop. 5.3.8). The construction of the least upper bound for a set of assignments is given by the union of the domains of the assignments and the union of the total distributed morphisms from the domains. The construction of the least upper bound is given formally in Proposition 5.3.8. At this point, I present only an example of the construction sufficient for understanding the following explanations and for applying the construction to our case study.

Example 3.7 (least upper bound of assignments)
Consider the two assignments q_4 and q_6 in Figure 3.9 taken from the graph-interpreted formulas in 3.6. The figure shows the construction of its least upper bound, denoted by $q_4 \cup q_6$. The domain of the assignment q_4 is given by the distributed graph $\hat{X}_{q_4}$ that contains

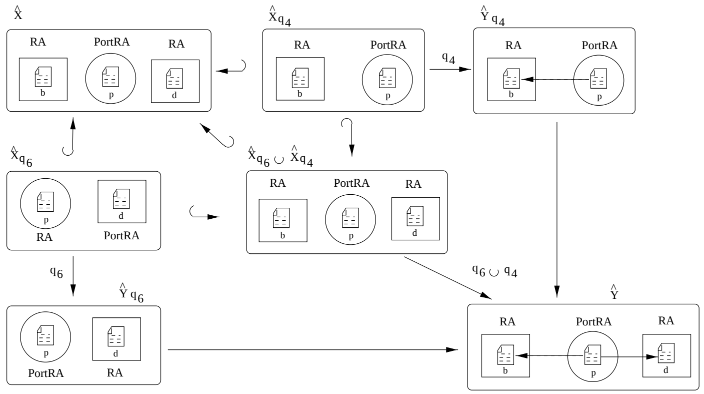


Figure 3.9: *The least upper bound of the assignments q_4 and q_6.*

the *RA*-node with document b and the *PortRA*-node with document p. The domain of q_5 contains the other *RA*-node with the document d and also the *PortRA*-node. The assignment $q_4 \cup q_6$ is constructed in three steps:

1. The union of the domains of q_4 and q_6 is constructed. The domain $\hat{X}_{q_4} \cup \hat{X}_{q_6}$ is equal to $\hat{X}$ since each element of $\hat{X}$ occurs either in $\hat{X}_{q_4}$ or $\hat{X}_{q_6}$.

2. The distributed graph $\hat{Y}$ is constructed by the union of the distributed graphs $\hat{Y}_{q_4}$ and $\hat{Y}_{q_6}$, where the *PortRA*-node is identified, since it is the same in both graphs $\hat{Y}_{q_4}$ and $\hat{Y}_{q_6}$. The two revision archives in $\hat{Y}$ are connected over this port.

3. The total distributed graph morphism from the domain $\hat{X}_{q_4} \cup \hat{X}_{q_6}$ to the graph $\hat{Y}$ is the union of the total distributed morphism from $\hat{X}_{q_4}$ to $\hat{Y}_{q_4}$ and the total distributed morphism from $\hat{X}_{q_6}$ to $\hat{Y}_{q_6}$.

It is easy to see, that whenever q_4 and q_6 can be embedded into an assignment, also the least upper bound $q_4 \cup q_6$ can be embedded. $\triangle$

Construction of the Typical Model The construction of the typical graph transition system with respect to a set of assignments A is based on the construction of the least upper bounds of subsets of A (Constr. 5.2.3). The subsets are all subsets B of A that are closed under entailment, i.e. whenever an assignment of A can be embedded into the least upper bound of B then the assignment is already in the set B. For an example, consider the assignments q_4 and q_1 taken from the graph-interpreted formulas in Figure 3.6. The least upper bound of the set containing only q_4 is q_4 itself. The assignment q_1 can be embedded into this least upper bound, since the domain of q_1 is a subgraph of the domain of q_4 and the target graph of q_1 is a subgraph of the target graph of q_4. Therefore, the subset containing only q_4 is not closed under entailment, since whenever q_4 is in the set also q_1 has to be in the set.
Given a set of assignments A, the typical graph transition system with respect to A has the least upper bounds for all subsets B of A as states that are closed under entailment. The transition relation is full in the sense that there is a transition form each state to each state and the set of runs contains all infinite paths through the transition system. The construction of the typical graph transition system is formally given in the Construction 5.2.3.
The typical transition system is finite, whenever the set of assignments is finite (cf. Prop. 5.2.4). In a practical specification, the set of assignments is induced by the assignments of the graph-interpreted temporal formulas of the specification. Since this set of formulas of a practical specification is finite, the graph transition system is finite for practical specifications.
As mentioned above, the typical graph transition system for a set of assignments A satisfies exactly those graph-interpreted formulas that are build up from assignments of A and that are satisfied by all temporal graph transition systems (cf. Corollary 5.2.6). That means, if we have given a temporal graph model generated by a distributed system specification and the typical graph model with respect to the graph-interpreted temporal formulas of this specification, then a graph-interpreted temporal formula is true in the temporal model for the grammar if the formula is true in the typical graph model. If the formula is not satisfied in the typical graph model, however, it can be true in the temporal graph model for the grammar. In this case, other checking techniques have to be considered. In this way, the typical graph transition system supports only the checking process, but does not solve the checking problem in infinite models completely.

Example 3.8 (typical graph model)
I construct the typical graph transition system for the graph-interpreted temporal formulas of the case study given in Figure 3.5 and Figure 3.6. The set of assignments is given by the constraints $q_1, ..., q_6$ occurring in these formulas. The states of the typical graph transition system are constructed as least upper bounds for all subsets of $\{q_1, ..., q_6\}$, that are closed under entailment (see Figure 3.10). The transition system has 12 states, where I have equipped each state by the set of assignments from that the least upper bound was constructed. I omitted the transition relation, since it is full. $\triangle$

3.3.1.1 From local to global properties

We have seen that the typical model collapses a possibly infinite model to a finite one. In this section I would like to mention another possibility given by the typical model, that is, to reason from local satisfaction to global satisfaction. As mentioned in the previous chapter, I provide for each process instance the open process semantics for a process instance,

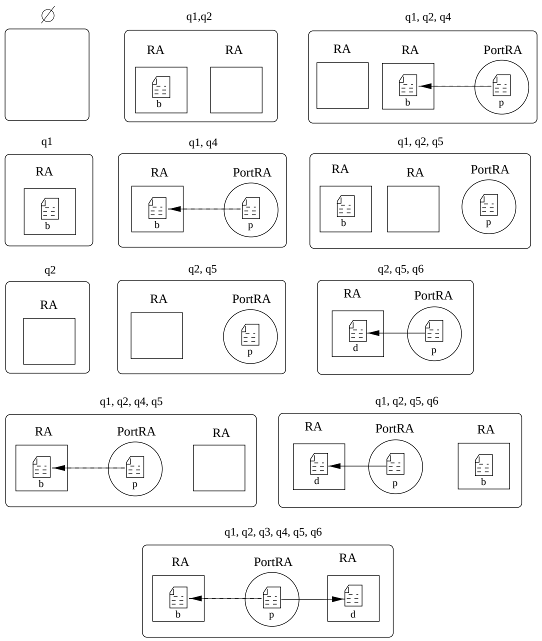


Figure 3.10: *The typical model for the graph-interpreted formulas in 3.5 and 3.6.*

that describes the behavior of a process instance in a complete environment in the sense that also all possible effects on ports are considered. The open process semantic induces a graph transition system, where local formulas can be checked. In Theorem 5.3.18 is shown that the satisfaction of local formulas in the open process semantics can be used to show the satisfaction of the local formulas in the complete distributed system. In this way, the checking of local formulas can be reduced to a local system state and we do not need a global state. Under the assumption of local satisfaction in the open process semantics, the typical model gives the possibility to check global formulas without constructing a global state (see Theorem 5.3.19). More details can be found in Section 5.3.3. I would like to clarify this property of the typical model in the next section, where I check the main properties of the case study.

3.3.2 Checking the main properties of the case study

I check now the two main properties of the case study, that are expressed by the two graph-interpreted formulas in Figure 3.5. The first formula specifies that a revision archive and a document in a revision archive are always preserved. The lower formula specifies that each document is eventually replicated to a remote archive. Whereas the first formula is a local formula, that can be checked locally in the open process semantics for revision archives, the lower formula is global. Therefore, I divided this global formula into the two local formulas

given in Figure 3.6.
The graph-interpreted formulas in Figure 3.5 of the case study are checked in two steps:

1. All local formulas are checked w.r.t. the open process semantics of revision archives. In this work I check their satisfaction informally. One may apply the typical model also for the local case, however, I omitted this step, since the local properties can be easily checked by regarding the process productions. Moreover, the typical model and the automatic verification of a graph-interpreted formula by a model-checker is explained in the next step.

2. I prove that the satisfaction of the local formulas in Figure 3.6 implies the satisfaction of the the global formula in Figure 3.5. This is done by means of the typical graph transition system given in Figure 3.10. This step shows as well, how to use a model checker to automatically check graph-interpreted temporal formulas.

3.3.2.1 Checking the local formulas

The satisfaction of the local formulas can be checked in the open process semantics for revision archives by regarding the process productions introduced in Chapter 2, Table 2.1.

- The local formula at the bottom of Figure 3.5 is satisfied since there does not exist a process production for revision archives that deletes a revision archive or a document in an archive. Therefore, revision archives as well as documents in archives are always preserved.

- The upper local formula in Figure 3.6 requires that a document is eventually put in the port of a revision archive and remains there. To ensure these property for our distributed system, we need the notion of *fairness* in the sense that a process production that can be applied is not infinitely often delayed. This can be attained by taking only those runs into the graph transition system that satisfy the fairness constraint. Then the local formula is satisfied since the process production `replicate` in Figure 2.17 can be used to put each document in the port. Since I do not have process productions for the deletion of documents in ports of type *PortRA*, the document remains there always.

- To ensure the satisfaction of the local formula at the bottom of the Figure 3.6 I need again fairness. Under this assumption, however, it is easy to see that the process production `import` in Figure 2.18 is always applicable.

3.3.2.2 Checking the global formula

I check now the global formula in Fig. 3.5 without constructing a global state. I need only the satisfaction of the local formulas in Fig. 3.6 and the typical model in Fig. 3.10 (cf. Theorem 5.3.19). I check the global formula by means of the model checker *STeP* (see [Man94]) that gets a transition system and a temporal formula as input. It answers "yes" if the transition system satisfies the temporal formula, gives a counterexample otherwise. Since the transition system in Fig. 3.10 is finite, the model-checker terminates. In Fig. 3.11 I have shown the main window of the model-checker. Figure 3.12 shows (a part of) the

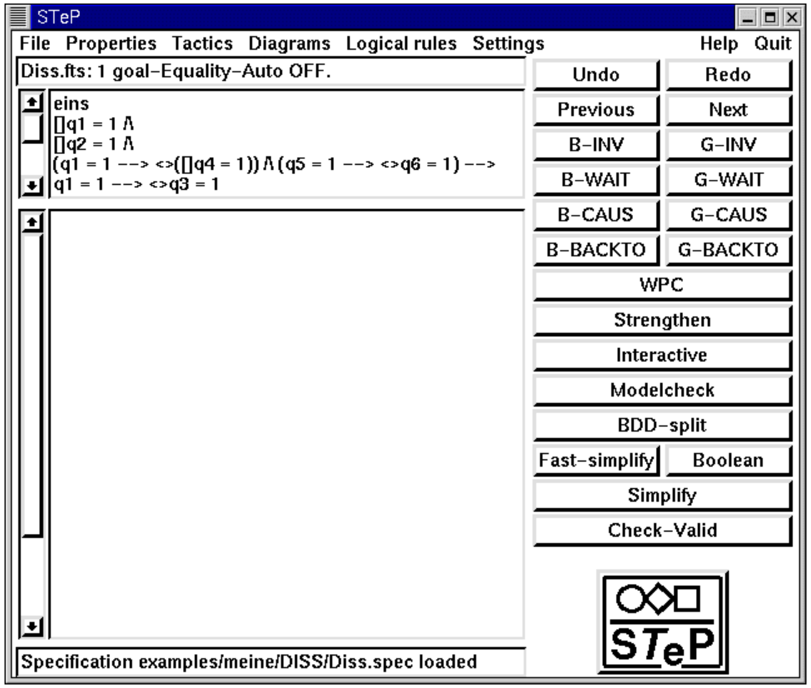


Figure 3.11: *The model-checker STeP.*

transition system. At the top of the model checker the temporal formula is shown. The assignments $q_1, ..., q_6$ of the graph-interpreted formulas are not inserted as distributed graph morphisms but are represented by the variables $q1, ..., q6$. The states of the transition system in Figure 3.10 are represented by the sets of variables depicted next to the graphical states. The notation $qi = 1$ means that the atomic formula qi for $i = 1...6$ is satisfied in a state. For example, the representation of the graphical state $q1, q2$ in Figure 3.10 is $q1 = q2 = 1$ and $q3 = q4 = q5 = q6 = 0$. The intended meaning of the formula shown in the model checker is that whenever the local formulas at the top of the Figure 3.5 and in Figure 3.6 are satisfied then the global formula is satisfied. Applying the model-checker results in the answer in Figure 3.13: The formula could be successfully checked. Therefore, the global formula is satisfied (according to Theorem 5.3.19), i.e. each document in our distributed system is eventually replicated to a remote archive.

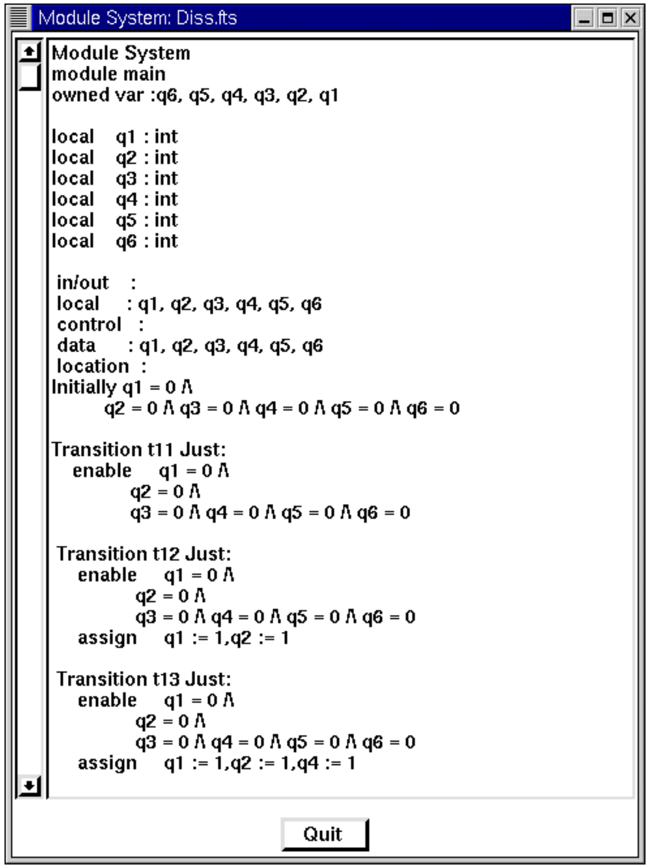


Figure 3.12: *The transition system of the case study.*

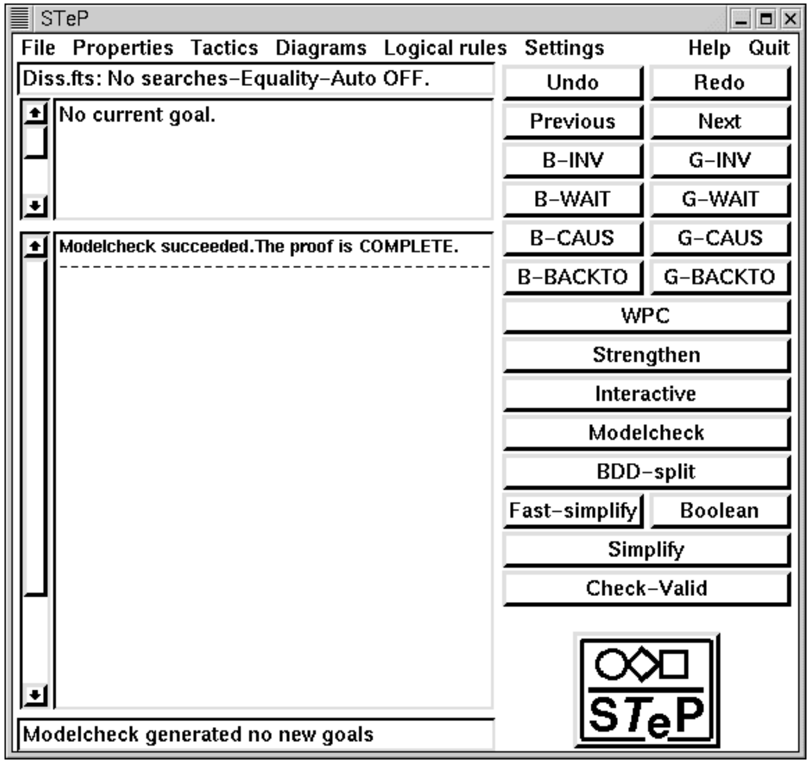


Figure 3.13: *The output after checking the formula: the model-checker STeP responded by yes.*

Chapter 4

Formal Description of Distributed Graph Transformation in the Single Pushout Approach

This chapter is concerned with the formal description of the single pushout approach to distributed graph transformation. Distributed graphs are based on *models* in a category **C**, that are total graph morphisms refining a graph by objects of the category **C** and the edges by **C**-morphisms. I investigate the category of models and partial model morphisms and provide existence conditions for pushouts in this category. Since distributed graphs are defined by models with an additional type information, these results are used to define the transformation of distributed graphs in the single-pushout approach.

I consider a distributed system consisting of concurrent processes communicating over shared ports. The *distributed system specification* consists of a *network type graph*, a *process grammar* and a *name space* for each process type in the network type graph and a *synchronization relation*. A process grammar provides a set of *process productions* and a *start production*. The process productions specify the possible actions of the process and the start production starts the process. From the viewpoint of object-oriented modeling, a process grammar can be seen as a class. Each instance of a process grammar gets the "program" specified in the process grammar. The *name space* provides the identifier for the process instances for each process type. The synchronization relation determines which process productions of which process instances have to be synchronized and over which data synchronization takes place.

I define the semantics of a distributed system specification by *transformation systems*, that are introduced in [GR99]. Transformation systems are a common semantical framework in which specification techniques can be interpreted. I show the interpretation of distributed graph transformation into this framework. This enables to use the results for the composition of transformation systems to specify a compositional semantics of a distributed system. The distributed semantics for a distributed system specification is composed from the transformation systems for the single process instances, called *open process semantics*. The open process semantics of a process instance includes the behavior of a process instance generated by the process productions of the corresponding process grammar and all possible effects of the environment on ports. The composition of the transformation systems for the process instances is induced by the synchronization relation. This relation determines how the transformation systems are connected. I compare the compositional operational semantics with the *global operational semantics* generated by the application of amalgamated productions.

I start by an introduction to typed graph transformation, since typed graphs are used to model the different process types of a distributed system later on.

4.1 Typed Graph Transformation

A *graph* $G = (G_V, G_E, s^G, t^G)$ consists of a set of nodes G_V, a set of edges G_E and two mappings $s^G : G_E \to G_V$ and $t^G : G_E \to G_V$. The *source mapping* s^G indicates the source node of an edge, the *target mapping* t^G its target node. Edges e of a graph G with source node $s^G(e) = i$ and target node $t^G(e) = j$ are often represented by $e : i \to j$. Multiple edges between two nodes are possible by this definition. A *total graph morphism* $f : G \to H$ between graphs $G = (G_V, G_E, s^G, t^G)$ and $H = (H_V, H_E, s^H, t^H)$ is a pair (f_V, f_E) of total mappings $f_V : G_V \to H_V$ between the set of nodes and $f_E : G_E \to H_E$ between the set of edges, such that f obeys the graph structure, i.e. $f_V \circ s^G = s^H \circ f_E$ and $f_V \circ t^G = t^H \circ f_E$. A *partial graph morphism* $f : G \rightharpoonup H$ is a total graph morphism $\bar{f} : dom(f) \to H$ from a subgraph $dom(f) \subseteq G$ to H. Partial graph morphisms are often represented by a span $[i_f : dom(f) \hookrightarrow G, \bar{f} : dom(f) \to H]$.

Definition 4.1.1 (category Graph and Graph$^{\mathbf{P}}$)
All graphs and all partial graph morphisms form a category **Graph**$^{\mathbf{P}}$. We denote by **Graph** the subcategory of **Graph**$^{\mathbf{P}}$ where graph morphisms are total. △

The category **Graph**$^{\mathbf{P}}$ has been studied in the algebraic single pushout approach [Löw93] and is complete and co-complete. Its subcategory **Graph** is known to be complete and co-complete as well.

Nodes and edges of a graph can be equipped by labels for carrying further application specific information. Labeling can be provided by a special label set and labeling mappings for node and edge sets, that assign a label to each of the nodes and to each of the edges of a graph (see chapter [EHK$^+$97] in [Roz97]). A more powerful concept for labeling are *type graphs*, that can be understood as a kind of entity/relationship diagram in conceptual data modeling. Compared with labeling mappings, *typed graph transformation* [CELP96] has the advantage to be structured in the sense that types distinguish between vertices and edges and edge types prescribe source and target types for the source and target vertices of the instances of the edge type. The type information is represented in a fixed graph TG, called *type graph*. The typing of a graph G in TG is given by a total graph morphism $t_G : G \to TG$. We call t_G a *typed graph in* TG. If a type graph is fixed and no confusion is possible, we refer to a typed graph $t_G : G \to TG$ simply by G and say G is a typed graph in TG. The graph morphism t_G has to be total since we require each node and each edge to have a type.

In [CELP96,HCEL96] this passing from untyped graphs to typed graphs is expressed by passing from the category **Graph** to the comma category (**Graph** $\downarrow TG$). Objects of this comma category are typed graphs $t_G : G \to TG$ in TG and morphisms between objects $t_G : G \to TG$ and $t_H : H \to TG$ are total graph morphisms $f : G \to H$, short *TG-typed graph morphisms*, such that types are preserved, i.e. $t_G = t_H \circ f$. The concept of TG-typed graph morphisms can be generalized to *TG-typed partial graph morphisms*: A TG-typed partial graph morphism is a partial graph morphism $f = [i_f : dom(f) \hookrightarrow G, \bar{f} : dom(f) \to H]$ where $G, dom(f)$ and H are typed graphs in the same type graph TG. The total graph morphisms

i_f and $\bar{f}$ are TG-typed graph morphisms, i.e. the diagram

$$\begin{tikzcd} G \arrow[dr, "t_G"'] & dom(f) \arrow[l, hook', "i_f"'] \arrow[r, "\bar{f}"] \arrow[d] & H \arrow[dl, "t_H"] \\ & TG & \end{tikzcd}$$

in **Graph** commutes. TG-typed partial graph morphisms preserve types of objects, unless they are outside of its domain.

Definition 4.1.2 (category of typed graphs)
For any graph TG in **Graph**, I denote the category of typed graphs in TG (that are objects of the comma category (**Graph** $\downarrow TG$)) and TG-typed partial graph morphisms by $\mathbf{Graph^P}_{TG}$. Composition is defined by composition in **Graph**$^{\mathbf{P}}$ and identities for each $t_G : G \to TG$ is the identity $id_G : G \to G$ for graph G in **Graph**$^{\mathbf{P}}$. △

The category $\mathbf{Graph^P}_{TG}$ is not just a comma category, since a total typing is required. However, it is build up from the comma category (**Graph** $\downarrow TG$). Since the category **Graph** is complete and co-complete the comma category (**Graph** $\downarrow TG$) has all limits and colimits as well, and the construction of pushouts and pullbacks coincides in **Graph** and (**Graph** $\downarrow TG$) up to additional typing information which is uniquely induced by mediating morphism in the case of colimits and morphism composition in the case of limits. Together with the completeness and co-completeness of category **Graph**$^{\mathbf{P}}$, the category $\mathbf{Graph^P}_{TG}$ has all limits and colimits as well.
Pushouts are the basic categorical construction in the following considerations since they define a rewriting step in the rewrite system. Therefore, I present here the pushout construction in $\mathbf{Graph^P}_{TG}$.

Construction 4.1.3 (pushouts)
For given TG-typed partial graph morphisms $f : A \rightharpoonup B$ and $g : A \rightharpoonup C$ the pushout is constructed as follows:

1. *Construct the pushout of f and g in category* **Graph**$^{\mathbf{P}}$ *[Löw93]. (The concrete construction is given later on in Proposition 4.2.16 for the construction of a pushout of partial model morphisms). A central result in [Löw93] states that (2) and (3) are pullbacks and (4) is a pushout in* **Graph**.

$$\begin{tikzcd} A \arrow[rrrd, dotted] & dom(f) \arrow[l, hook', "i_f"'] \arrow[r, "\bar{f}"] \arrow[dl, phantom, "(1)"] \arrow[dr, phantom, "(2)"] & B \arrow[dr, dotted] & \\ dom(g) \arrow[u, hook, "i_g"] \arrow[d, "\bar{g}"'] & X \arrow[l] \arrow[r] \arrow[u] \arrow[d] \arrow[dl, phantom, "(3)"] \arrow[dr, phantom, "(4)"] & dom(g^*) \arrow[u, hook, "i_{g^*}"'] \arrow[d, "\bar{g}^*"] \arrow[r, dotted] & TG \\ C \arrow[rrru, dotted] & dom(f^*) \arrow[l, hook', "i_{f^*}"'] \arrow[r, "\bar{f}^*"'] & D \arrow[ur, dotted] & \end{tikzcd}$$

2. *Add the typing of D as unique extensions of the resulting diagrams.*

△

4.2 Models and Model Morphisms for Refinement

I introduce in this section models and model morphisms. At first, I investigate the category of models and *total* model morphisms and show its co-completeness. Based on these results, I define the category of models and *partial* model morphisms and show the existence of pushouts.

4.2.1 Models and Total Model Morphisms

Graphs are well suited for the specification of various kind of graphical structures, as they occur in entity-relationship diagrams, statecharts, Petri nets, numerous diagrams in UML like collaboration diagrams, sequence diagrams etc. However, graphical descriptions are becoming confused if the specification becomes bigger. Therefore, a notion of abstraction and refinement is desirable to keep the clarity of big specifications. For instance, a state in a statechart diagram may be refined to an entire statechart again or states in diagrams can be refined to sub-diagrams. Vice versa, entire state charts are abstracted to one state or sub-diagrams are seen as one abstract state in a higher-level diagram.
Following these idea of refinement I intend to refine nodes of a graph by objects of a given category $\mathbf{C}$ and the edges by $\mathbf{C}$-morphisms. Formally, the refinement is given by a total graph morphism: its domain graph is the graph that has to be refined, its co-domain graph represents the admissible states for the refinement. The co-domain graph for the refinement space is the *underlying graph* $G_{\mathbf{C}}$ of the category $\mathbf{C}$, that is a labeled graph that has a node with label A for each object A in $\mathbf{C}$ and an edge with label m between a node with label A and a node with label B for each morphism m in $\mathcal{M}or_{\mathbf{C}}(A, B)$.
I call the total graph morphism for the refinement a *model* in analogy to the theory of sketches. In fact, the graph G can be understood as a very simple sketch, that does not contain any diagram, cone or cocone. Because of this simple case, I omitted here the introduction in the theory of sketches. Moreover, due to partial relations between models, I leave the framework of sketches as introduced in [BW95]. However, the interested reader is referred to [BW95].

Definition 4.2.1 (models)
A *model* over a graph G in category $\mathbf{C}$ is a total graph morphism $\mathcal{G} : G \to G_{\mathbf{C}}$ from the graph G to the underlying graph $G_{\mathbf{C}}$ of the category $\mathbf{C}$. We call the graph G the *base graph* of the model $\mathcal{G}$. For each node v in G_V we call $\mathcal{G}(v)$ the *refined graph* for v and for each edge e in G_E we call $\mathcal{G}(e)$ the *refined morphism* for e. △

Morphisms between models $\mathcal{G}$ and $\mathcal{H}$ over base graphs G resp. H consist of two components: The first component is given by a total graph morphism $f : G \to H$ between the base graphs, the second component consists of a family of $\mathbf{C}$-morphisms that contains a $\mathbf{C}$-morphism $\alpha(v)$ between the refined graph $\mathcal{G}(v)$ and the refined graph $\mathcal{H}(f(v))$ for each node v in G. There does not exist any direct relation between refined morphisms for edges, but the $\mathbf{C}$-morphisms between refined graphs have to obey the refinement of edges in the sense that for each preserved edge e from i to j one has $\mathcal{H}(f(e)) \circ \alpha(i) = \alpha(j) \circ \mathcal{G}(e)$. Total model morphisms can also be seen as generalized natural transformations [EM85,BGT89].

Definition 4.2.2 (total model morphism)
A *total model morphism* between models $\mathcal{G}$ and $\mathcal{H}$ over G resp. H in $\mathbf{C}$ is a pair $\hat{f} = \langle f, \alpha^f \rangle :$ $\mathcal{G} \to \mathcal{H}$ where $f : G \to H$ is a total graph morphism, called *base morphism*, and α^f is a family of morphisms in $\mathbf{C}$, $\{\alpha^f(i) : \mathcal{G}(i) \to \mathcal{H}(f(i)) | i \in G_V\}$ such that for all edges $e : i \to j$ in G_E, $\mathcal{H}(f(e)) \circ \alpha^f(i) = \alpha^f(j) \circ \mathcal{G}(e)$.

$$\begin{tikzcd} i \arrow[d, "e"'] & \mathcal{G}(i) \arrow[r, "\alpha^f(i)"] \arrow[d, "\mathcal{G}(e)"'] & \mathcal{H}(f(i)) \arrow[d, "\mathcal{H}(f(e))"] \\ j & \mathcal{G}(j) \arrow[r, "\alpha^f(j)"'] & \mathcal{H}(f(j)) \end{tikzcd}$$

△

Models in $\mathbf{C}$ and total model morphisms form a category $\mathbf{Mod}^T_{\mathbf{C}}$. The *composition* of two total model morphisms $\hat{f} = \langle f, \alpha^f \rangle : \mathcal{G} \to \mathcal{H}$ and $\hat{g} = \langle g, \alpha^g \rangle : \mathcal{H} \to \mathcal{K}$ is defined by $\hat{g} \circ \hat{f} = \langle g \circ f, \alpha^{g \circ f} \rangle : \mathcal{G} \to \mathcal{K}$ where $\alpha^{g \circ f}(i) := \alpha^g(f(i)) \circ \alpha^f(i)$ for each $i \in G_V$. The associativity of composition holds due to the associativity of the composition in **Graph** and in $\mathbf{C}$. *Identities* for models $\mathcal{G}$ are given by $\hat{id}_{\mathcal{G}} = \langle id_G, \alpha^{id_G} \rangle : \mathcal{G} \to \mathcal{G}$ where $\alpha^{id_G}(i) = id_{\mathcal{G}(i)}$ for each $i \in G_V$.

Example 4.1 (models in Set)
In order to explain the concepts of models introduced now and later on, I investigate models in the category **Set** of sets and total mappings. These models refine the nodes of a graph by sets and its edges by total mappings. A possible model of the category $\mathbf{Mod}^T_{\mathbf{Set}}$ is depicted in the Figure 4.1. Its base graph consists of two nodes connected by an edge. The source

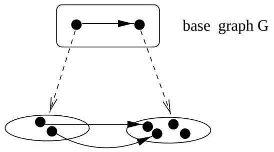


Figure 4.1: *A model in* **Set**

node is refined by a set consisting of two elements, the target node by a set consisting of four elements. The refined edge is given by a total mapping between the two sets. A model morphism is shown in Figure 4.2. The morphism satisfies the commutativity required for the refinement of edges. △

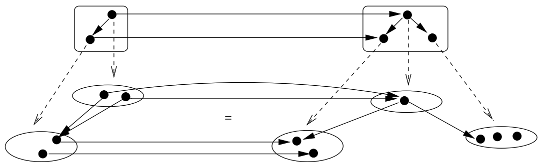


Figure 4.2: *A total model morphism in* **Set**

In the remainder of this section, I investigate existence conditions for coproducts and pushouts in the category $\mathbf{Mod}^T_\mathbf{C}$. These conditions are based on the results for the co-completeness of flattened indexed categories in [BGT89]. Since the category $\mathbf{Mod}^T_\mathbf{C}$ can be seen as a special flattened indexed category the results can be used. However, since we have a concrete category, we can simplify the conditions for the existence of colimits given for the general case.

The existence of coproducts in the category $\mathbf{Mod}^T_\mathbf{C}$ is independent of the category $\mathbf{C}$ and follows directly from the existence of coproducts in the category **Graph**. This is due to the construction of the coproduct of graphs by their disjoint union. This construction is directly lifted up to the category $\mathbf{Mod}^T_\mathbf{C}$, i.e. the coproduct of a family of models is their disjoint union as well.

Proposition 4.2.3 (coproduct)
The category $\mathbf{Mod}^T_\mathbf{C}$ *has all coproducts.* △

Proof
Given a family of models $(\mathcal{G}_i)_{i\in\mathbb{N}}$ over G_i, let G with injections $in_i : G_i \to G$ be the coproduct of the $\mathcal{G}_i$ in **Graph**. Then, we define $\mathcal{G} : G \to G_\mathbf{C}$ for all x in G (nodes as well as edges) by $\mathcal{G}(x) := \mathcal{G}_i(x)$ if x in G_i. The coproduct morphism $\hat{in}_i = \langle in_i, \alpha^{in_i}\rangle : \mathcal{G}_i \to \mathcal{G}$ is defined by $\alpha^{in_i}(x) = id_{\mathcal{G}_i(x)}$ for each x in $\mathcal{G}_{i_V}$. We claim that $\langle \mathcal{G}, (\hat{in}_i)_{i\in N}\rangle$ is the coproduct of the $\mathcal{G}_i$ in $\mathbf{Mod}^T_\mathbf{C}$.
Since G is the disjoint union of the G_i, the distributed graph $\mathcal{G}$ and the model morphisms $\hat{in}_i$ for each $i \in \mathbb{N}$ are well-defined. For a model $\mathcal{V}$ and model morphisms $\hat{f}_i = \langle f_i, \alpha^{f_i}\rangle : \mathcal{G}_i \to \mathcal{V}$ for $i \in \mathbb{N}$ there is a unique total graph morphism $u : G \to V$ such that $u \circ in_i = f_i$ for each $i \in \mathbb{N}$ since G is coproduct of the G_i in **Graph**. Then, define for each $x \in G_V$ the $\mathbf{C}$-morphism $\alpha^u(x) : \mathcal{G}(x) \to \mathcal{V}(u(x))$ by $\alpha^u(x) = \alpha^{f_i}(x)$ if $x \in G_i$. Then, $\hat{u} = \langle u, \alpha^u\rangle : \mathcal{G} \to \mathcal{V}$ is a model morphism since G is the disjoint union of the G_i. Furthermore, $\alpha^u(x) \circ \alpha^{in_i}(x) = \alpha^{f_i}(x) \circ id_{\mathcal{G}_i(x)} = \alpha^{f_i}(x)$ for all $i \in N$ and $x \in G_{i_V}$. Therefore, $\hat{u} \circ \hat{in}_i = \hat{f}_i$ for all $i \in \mathbb{N}$. The model morphism $\hat{u}$ is unique since u is unique and $\alpha^u(x)$ is unique for each $x \in G_V$ because of the uniqueness of the identity $id_{\mathcal{G}_i(x)}$. □

By restricting the category $\mathbf{Mod}^T_\mathbf{C}$ on models over a fixed base graph G and the base morphism to be the identity on G I achieve category $\mathbf{Mod}^T_\mathbf{C}(G)$, that is isomorphic to the functor category from the small category induced by G and category $\mathbf{C}$. The existence of colimits in the category $\mathbf{Mod}^T_\mathbf{C}(G)$ is based on the existence of colimits in $\mathbf{C}$ and colimits are constructed pointwise in category $\mathbf{C}$. The restricted categories $\mathbf{Mod}^T_\mathbf{C}(G)$ are used to construct a pushout in $\mathbf{Mod}^T_\mathbf{C}$ later on.

Proposition 4.2.4
The category $\mathbf{Mod}^T_\mathbf{C}(G)$ *of models over* G *in the category* $\mathbf{C}$ *is co-complete if the category* $\mathbf{C}$ *is co-complete.* △

Proof
Colimits are constructed pointwise for the family of $\mathbf{C}$-morphisms. The colimit object is defined by unique extensions generated by the resulting diagrams. □

Switching from a category $\mathbf{Mod}^T_\mathbf{C}(H)$ over a base graph H to a category $\mathbf{Mod}^T_\mathbf{C}(G)$ over a base graph G is always possible if there exists a total graph morphism $f : G \to H$. This

switching is given by a functor $V_f : \mathbf{Mod}^T_\mathbf{C}(H) \to \mathbf{Mod}^T_\mathbf{C}(G)$, called *translation functor* in the following, that uses the refined graphs $\mathcal{H}(v)$ of nodes v in the image of f for defining a model in G. Refined graphs $\mathcal{H}(v)$ of nodes v in $H - f(G)$, however, are not considered.

Definition 4.2.5 (translation functor)
Given a total graph morphism $f : G \to H$, the *translation functor* $V_f : \mathbf{Mod}^T_\mathbf{C}(H) \to \mathbf{Mod}^T_\mathbf{C}(G)$ is defined for models $\mathcal{H}$ in $\mathbf{Mod}^T_\mathbf{C}(H)$ by $V_f(\mathcal{H}) = \mathcal{H} \circ f$ and for model morphisms $\hat{h} = \langle id_H, \alpha^h\rangle$ by $V_f(\hat{h}) = \langle id_G, V_f(\alpha^h)\rangle$, where $V_f(\alpha^h)(i) = \alpha^h(f(i))$ for each $i \in G_V$. △

Each model morphism $\hat{f} = \langle f, \alpha^f\rangle : \mathcal{G} \to \mathcal{H}$ in $\mathbf{Mod}^T_\mathbf{C}$ induces a model morphism $\hat{f}' = \langle id_G, \alpha^{f'}\rangle : \mathcal{G} \to V_f(\mathcal{H})$ where $\alpha^{f'}(x) = \alpha^f(x)$ for each $x \in G_V$.

Example 4.2 (translation functor for models in Set)
The example in Figure 4.3 shows how a model with base graph H is translated along f to a model over base graph G. The refined graph for a node v of the model $\mathcal{G}$ over G is calculated as follows: Consider the image $f(v)$ of the node v in H. Take the refined graph of $f(v)$ to refine the node v in G. The same is performed for the edge. △

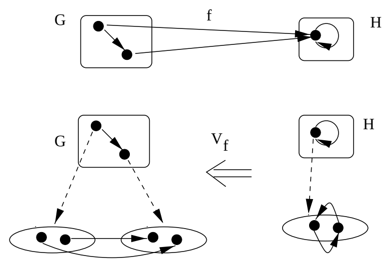


Figure 4.3: *Translating models along total graph morphisms.*

In contrast to the existence of coproducts, the existence of colimits in the category $\mathbf{C}$ is required for pushouts in the category $\mathbf{Mod}^T_\mathbf{C}$. In addition, the pushout construction of two model morphisms in $\mathbf{Mod}^T_\mathbf{C}$ uses left adjoints with respect to translation functors. Translation functors do not have in general left adjoints, their existence depends on the category $\mathbf{C}$. In [Man76] is shown that left adjoints exist, if the category $\mathbf{C}$ is algebraically. Informations about left adjoints in general can be found in [EM85,BW95].
The idea of the pushout construction of two model morphisms $\hat{f}$ and $\hat{g}$ in $\mathbf{Mod}^T_\mathbf{C}$ is as follows: First, the pushout of the total base morphisms f and g is constructed in the category **Graph**. Their pushout object, called D in the following, is the base graph of the pushout object of $\hat{f}$ and $\hat{g}$ in $\mathbf{Mod}^T_\mathbf{C}$. In order to define its refinement, I translate the model morphisms $\hat{f}$ and $\hat{g}$ into model morphisms of the category $\mathbf{Mod}^T_\mathbf{C}(D)$, where I can construct the pushout pointwise according to Proposition 4.2.4. The translation of the model morphisms $\hat{f}$ and $\hat{g}$ takes place by means of the left adjoints with respect to the translation functors induced by the pushout morphisms of the pushout of f and g in **Graph**. Let us assume $(\mathcal{D}, \hat{x}, \hat{y})$ is the pushout of the translated morphisms in $\mathbf{Mod}^T_\mathbf{C}(D)$, then the pushout object $\mathcal{D}$ is immediately the pushout object of $\hat{f}$ and $\hat{g}$ in $\mathbf{Mod}^T_\mathbf{C}$. The pushout morphisms for $\hat{f}$ and $\hat{g}$

are constructed from the pushout morphisms $\hat{x}$ and $\hat{y}$ and the universal morphisms of the free constructions.

Proposition 4.2.6 (pushouts)
The category $\mathbf{Mod}^T_{\mathbf{C}}$ has all pushouts if

1. *the category $\mathbf{C}$ has all pushouts and*
2. *left adjoints $F_f : \mathbf{Mod}^T_{\mathbf{C}}(G) \to \mathbf{Mod}^T_{\mathbf{C}}(H)$ exist for translation functors $V_f : \mathbf{Mod}^T_{\mathbf{C}}(H) \to \mathbf{Mod}^T_{\mathbf{C}}(G)$ w.r.t. a total graph morphism $f : G \to H$.*

$\triangle$

Proof
Given two model morphisms $\hat{f} = \langle f, \alpha^f \rangle : \mathcal{A} \to \mathcal{B}$ and $\hat{g} = \langle g, \alpha^g \rangle : \mathcal{A} \to \mathcal{C}$, the pushout $(\mathcal{D}, \hat{c}, \hat{b})$ of $\hat{f}$ and $\hat{g}$ in $\mathbf{Mod}^T_{\mathbf{C}}$ is constructed as follows:

- Construct the pushout $(D, c : C \to D, b : B \to D)$ of the morphisms f and g in the category **Graph**.
- Let $\hat{f}^* = \langle id_B, \alpha^{f^*} \rangle : F_f(\mathcal{A}) \to \mathcal{B}$ be the unique model morphism for $\mathcal{B}$ and the model morphism $\hat{f}' = \langle id_A, \alpha^{f'} \rangle : \mathcal{A} \to V_f(\mathcal{B})$ induced by $\hat{f}$ and $\hat{g}^* = \langle id_C, \alpha^{g^*} \rangle : F_g(\mathcal{A}) \to \mathcal{C}$ be the unique model morphism for $\mathcal{C}$ and the model morphism $\hat{g}' = \langle id_A, \alpha^{g'} \rangle : \mathcal{A} \to V_f(\mathcal{C})$ induced by $\hat{g}$. Then, construct the pushout $(\mathcal{D}, \hat{x}, \hat{y})$ of $F_b(\hat{f}^*)$ and $F_c(\hat{g}^*)$ in the category $\mathbf{Mod}^T_{\mathbf{C}}(D)$.

$$\begin{tikzcd}
F_b(F_f(\mathcal{A})) \arrow[r, "F_b(\hat{f}^*)"] \arrow[d, "F_c(\hat{g}^*)"'] \arrow[dr, phantom, "(1)"] & F_b(\mathcal{B}) \arrow[d, "\hat{y}"] \\
F_c(\mathcal{C}) \arrow[r, "\hat{x}"'] & \mathcal{D}
\end{tikzcd}$$

  Please notice, that $F_b(F_f(\mathcal{A})) \cong F_c(F_g(\mathcal{A}))$.

- Define the model morphism $\hat{c} : \mathcal{C} \to V_c(\mathcal{D})$ by $\hat{c} = V_c(\hat{x}) \circ \eta^c_{\mathcal{C}}$ and the model morphism $\hat{b} : \mathcal{B} \to \mathcal{D}$ by $\hat{b} = V_b(\hat{y}) \circ \eta^b_{\mathcal{B}}$.

Now we have to prove that $(\mathcal{D}, \hat{c}, \hat{b})$ is indeed a pushout of $\hat{f}$ and $\hat{g}$ in $\mathbf{Mod}^T_{\mathbf{C}}$.

commutativity: We have to show that $\hat{c} \circ \hat{g} = \hat{b} \circ \hat{f}$, i.e. $\langle c \circ g, V_g(\alpha^c) \circ \alpha^g \rangle = \langle b \circ f, V_f(\alpha^b) \circ \alpha^f \rangle$. By construction $b \circ f = c \circ g$ and we define $h := c \circ g$. It remains to show that $V_g(\alpha^c) \circ \alpha^g = V_f(\alpha^b) \circ \alpha^f$, i.e. that $V_f(\alpha^b) \circ \alpha^f = V_f(V_b(\alpha^y) \circ \eta^b_{\mathcal{B}}) \circ \alpha^f = V_f(V_b(\alpha^y)) \circ V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f = V_h(\alpha^y) \circ V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f = V_h(\alpha^x) \circ V_g(\eta^c_{\mathcal{C}}) \circ \alpha^g = V_g(\alpha^c) \circ \alpha^g$ or that the diagram (2) below commutes.

$$\begin{tikzcd}
& V_f(\mathcal{B}) \arrow[r, "V_f(\eta^b_{\mathcal{B}})"] & V_h(F_b(\mathcal{B})) \arrow[dr, "V_h(\alpha^y)"] & \\
\mathcal{A} \arrow[ur, "\alpha^f"] \arrow[dr, "\alpha^g"'] & (2) & & V_h(\mathcal{D}) \\
& V_g(\mathcal{C}) \arrow[r, "V_g(\eta^c_{\mathcal{C}})"'] & V_h(F_c(\mathcal{C})) \arrow[ur, "V_h(\alpha^x)"'] &
\end{tikzcd}$$

If we assume

$$(*)\ V_h(\alpha^y) \circ V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f = V_h(\alpha^y) \circ V_h(F_b(\alpha^{f^*})) \circ \eta^h_{\mathcal{A}}$$

then holds due to the universal property of a free construction

$$(V_h(\alpha^x) \circ V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f)^* = \alpha^y \circ F_b(\alpha^{f^*}).$$

Since diagram (1) is a pushout and commutes we get $(V_h(\alpha^x) \circ V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f)^* = \alpha^y \circ F_b(\alpha^{f^*}) = \alpha^x \circ F_c(\alpha^{g^*}) = (V_h(\alpha^x) \circ V_g(\eta^c_{\mathcal{C}}) \circ \alpha^g)^*$ such that diagram (2) commutes.

It remains to show that (*) is true, where it is enough to show that

$$V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f = V_h(F_b(\alpha^{f^*})) \circ \eta^h_{\mathcal{A}}.$$

For showing that, consider the unit $\eta^b : \mathbf{Id}_{\mathbf{Mod}^T_{\mathbf{C}}(B)} \to V_b \circ F_b$ of the adjunction F_b, where by construction of the unit the following diagram commutes.

$$\begin{tikzcd}
F_f(\mathcal{A}) \arrow[r, "\alpha^{f^*}"] \arrow[d, "\eta^b(F_f(\mathcal{A}))"'] & \mathcal{B} \arrow[d, "\eta^b(\mathcal{B})"] \\
V_b(F_b(F_f(\mathcal{A}))) \arrow[r, "V_b(F_b(\alpha^{f^*}))"'] & V_b(F_b(\mathcal{B}))
\end{tikzcd}$$

Because of $\eta^h_{\mathcal{A}} = V_f(\eta^b(F_f(\mathcal{A}))) \circ \eta^f$ the outer diagram below commutes.

$$\begin{tikzcd}
V_h(F_h(\mathcal{A})) \arrow[r, "V_h(F_b(\alpha^{f^*}))"] & V_h(F_b(\mathcal{B})) \\
V_f(F_f(\mathcal{A})) \arrow[u, "V_f(\eta^b(F_f(\mathcal{A})))"] \arrow[r, "V_f(\alpha^{f^*})"] \arrow[ur, phantom, "="] & V_f(\mathcal{B}) \arrow[u, "V_f(\eta^b_{\mathcal{B}})"'] \\
\mathcal{A} \arrow[u, "\eta^f_{\mathcal{A}}"] \arrow[ur, "\alpha^f"'] &
\end{tikzcd}$$

Therefore, $V_f(\eta^b_{\mathcal{B}}) \circ \alpha^f = V_h(F_b(\alpha^{f^*})) \circ \eta^h_{\mathcal{A}}$.

universal property: Let $\hat{k} = \langle k, \alpha^k \rangle : \mathcal{C} \to \mathcal{E}$ and $\hat{a} : \mathcal{B} \to \mathcal{E}$ be two model morphisms such that $\hat{a} \circ \hat{f} = \hat{k} \circ \hat{g}$, i.e. $a \circ f = k \circ g$ and $V_f(\alpha^a) \circ \alpha^f = V_g(\alpha^k) \circ \alpha^g$.

Since (D, c, b) is pushout of f, g in the category **Graph**, there is a unique total graph morphism $u : D \to E$ such that $u \circ c = k$ and $u \circ b = a$. Let $\alpha^{k^*} : F_c(\mathcal{C}) \to V_u(\mathcal{E})$ be the unique morphism of the free construction for $\alpha^k : \mathcal{C} \to V_c(V_u(\mathcal{E}))$ and the translation functor V_c. Analog, let $\alpha^{a^*} : F_b(\mathcal{B}) \to V_u(\mathcal{E})$ be the unique morphism of the free construction for $\alpha^a : \mathcal{B} \to V_b(V_u(\mathcal{E}))$ and the translation functor V_b.

$$\begin{tikzcd}
F_b(F_f(\mathcal{A})) \arrow[r, "F_b(\hat{f}^*)"] \arrow[d, "F_c(\hat{g}^*)"'] \arrow[dr, phantom, "(1)"] & F_b(\mathcal{B}) \arrow[d, "\hat{y}"] \arrow[ddr, "\alpha^{a^*}"] & \\
F_c(\mathcal{C}) \arrow[r, "\hat{x}"'] \arrow[drr, "\alpha^{k^*}"'] & \mathcal{D} \arrow[dr, "\alpha^u"] & \\
& & \mathcal{E}
\end{tikzcd}$$

We assume now

$$(**)\ (V_f(\alpha^a) \circ \alpha^f)^* = \alpha^{a^*} \circ F_b(\alpha^{f^*}) \text{ and } (V_g(\alpha^k) \circ \alpha^g)^* = \alpha^{k^*} \circ F_c(\alpha^{g^*})$$

Since $V_f(\alpha^a) \circ \alpha^f = V_g(\alpha^k) \circ \alpha^g$ by premise , we get $\alpha^{a^*} \circ F_b(\alpha^{f^*}) = \alpha^{k^*} \circ F_c(\alpha^{g^*})$. Since (1) is pushout in $\mathbf{Mod}_{\mathbf{C}}^T(D)$, there is a unique morphism $\alpha^u : \mathcal{D} \to V_u(\mathcal{E})$ such that $\alpha^u \circ \alpha^y = \alpha^{a^*}$ and $\alpha^u \circ \alpha^x = \alpha^{k^*}$. Then follows $V_c(\alpha^u) \circ \alpha^c = V_c(\alpha^u) \circ V_c(\alpha^x) \circ \eta_{\mathcal{C}}^c = V_c(\alpha^u \circ \alpha^x) \circ \eta_{\mathcal{C}}^c = V_c(\alpha^{k^*}) \circ \eta_{\mathcal{C}}^c = \alpha^k$ and analog $V_b(\alpha^u) \circ \alpha^b = \alpha^a$.

Therefore, for $\hat{u} = \langle u, \alpha^u \rangle : \mathcal{D} \to \mathcal{E}$ holds $\hat{u} \circ \hat{c} = \hat{k}$ and $\hat{u} \circ \hat{b} = \hat{a}$. The uniqueness of $\hat{u}$ follows from the uniqueness of u and α^u.

It remains to show that (**) is true: We have to show that $V_f(V_b(\alpha^{a^*} \circ F_b(\alpha^{f^*}))) \circ \eta_{\mathcal{A}}^{b \circ f} = V_f(\alpha^a) \circ \alpha^f$ resp. $V_f(V_b(\alpha^{k^*} \circ F_c(\alpha^{g^*}))) \circ \eta_{\mathcal{A}}^{b \circ f} = V_g(\alpha^k) \circ \alpha^g$, since the claim follows then from the universal property of a free construction. We have $V_f(V_b(\alpha^{a^*} \circ F_b(\alpha^{f^*}))) \circ \eta_{\mathcal{A}}^{b \circ f} = V_f(V_b(\alpha^{a^*})) \circ V_f(V_b(F_b(\alpha^{f^*}))) \circ \eta_{\mathcal{A}}^{b \circ f} = V_f(V_b(\alpha^{a^*})) \circ V_f(\eta_{\mathcal{B}}^b) \circ \alpha^f = V_f(V_b(\alpha^{a^*}) \circ \eta_{\mathcal{B}}^b) \circ \alpha^f = V_f(\alpha^a) \circ \alpha^f$. The case $V_f(V_b(\alpha^{k^*} \circ F_c(\alpha^{g^*}))) \circ \eta_{\mathcal{A}}^{b \circ f} = V_g(\alpha^k) \circ \alpha^g$ is shown analogously.

□

Example 4.3 (pushout for model morphisms in Set)
Consider the two model morphisms $\hat{f}$ and $\hat{g}$ in Figure 4.4. The two main steps of the

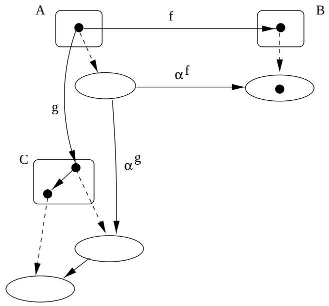


Figure 4.4: *Two model morphisms in* **Set** *from that the pushout shall be constructed.*

pushout construction are shown in Figure 4.5: The first step, shown on the left-hand side of the figure, is the pushout construction of the base morphisms in the category **Graph**. Then, we translate the diagram in Figure 4.4 into a diagram in the category $\mathbf{Mod}_{\mathbf{C}}(D)$ via the left adjoint to the translation functor induced by the graph morphism $b \circ f$ resp. $c \circ g$. The resulting diagram is shown on the right-hand side. Please notice, that translation of the model $\mathcal{B}$ into a model over D generates freely an element to define the mapping. Then, the pushout in $\mathbf{Mod}_{\mathbf{Set}}(D)$ is constructed by constructing the pushout in **Set** for each component. The resulting model over D is the pushout object of $\hat{f}$ and $\hat{g}$. The whole pushout including the pushout morphisms is shown in Figure 4.6.

Theorem 4.2.7 (co-completeness of $\mathbf{Mod}_{\mathbf{C}}^T$)
The category $\mathbf{Mod}_{\mathbf{C}}^T$ is co-complete if $\mathbf{C}$ is co-complete and left adjoints exist for translation functors $V_f : \mathbf{Mod}_{\mathbf{C}}^T(H) \to \mathbf{Mod}_{\mathbf{C}}^T(G)$ w.r.t. graph morphisms $f : G \to H$. △

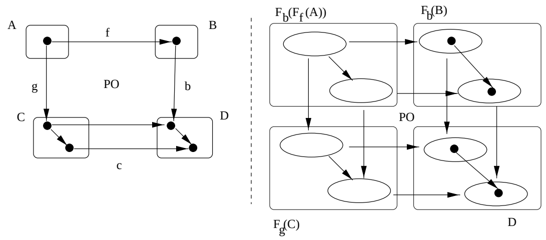


Figure 4.5: *The pushout of the graph morphisms in* **Graph** *and the pushout of the translated diagram in* $\mathbf{Mod}_{Set}(D)$.

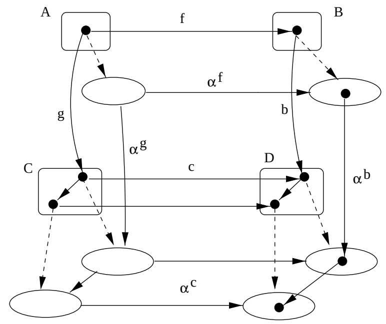


Figure 4.6: *Pushout in* $\mathbf{Mod}_{\mathbf{Set}}$.

Proof
The claim follows directly from Proposition 4.2.3 and Proposition 4.2.6 and the fact that a category that has all coproducts and all pushouts has all colimits. □

4.2.2 Models and Partial Model Morphisms

Until now, I required a total graph morphism for the base morphism of a model morphism. I consider now *partial model morphisms*, where the base morphism is partial. For partial model morphisms, we require a **C**-morphism only for the nodes in the domain of the base morphism.

Definition 4.2.8 (partial model morphism)
A *partial model morphism* $\hat{f} = \langle f, \alpha^f \rangle : \mathcal{G} \to \mathcal{H}$ between models $\mathcal{G}$ over G and $\mathcal{H}$ over H is given by a partial graph morphism $f : G \rightharpoonup H$ and a family of **C**-morphisms $\alpha^f = \{\alpha^f(i) : \mathcal{G}(i) \to \mathcal{H}(f(i)) | i \in dom(f)_V\}$, such that for all edges $e : i \to j$ in $dom(f)_E$, $\mathcal{H}(f(e)) \circ \alpha^f(i) = \alpha^f(j) \circ \mathcal{G}(e)$. △

Models and partial model morphisms define a category $\mathbf{Mod}_{\mathbf{C}}$. The category $\mathbf{Mod}_{\mathbf{C}}$ is an extension of the category $\mathbf{Mod}_{\mathbf{C}}^T$ by adding the partial model morphisms. Identities in $\mathbf{Mod}_{\mathbf{C}}$ are given by identities in $\mathbf{Mod}_{\mathbf{C}}^T$. The *composition* of two partial model morphisms

$\hat{f} = \langle f, \alpha^f \rangle : \mathcal{G} \to \mathcal{H}$ and $\hat{g} = \langle g, \alpha^g \rangle : \mathcal{H} \to \mathcal{K}$ is defined by $\hat{g} \circ \hat{f} = \langle g \circ f, \alpha^{g \circ f} \rangle : \mathcal{G} \to \mathcal{K}$ where $\alpha^{g\circ f}(i) := \alpha^g(f(i)) \circ \alpha^f(i)$ for each $i \in dom(g \circ f)_V$. The associativity of composition holds due to the associativity of the composition in $\mathbf{Graph}^P$ and in $\mathbf{C}$.

The construction of pushouts as introduced in the proof of 4.2.1 for model morphisms with total base morphisms cannot be lifted up to the category $\mathbf{Mod}^T_\mathbf{C}$ since translation functors with respect to partial graph morphisms do not have left adjoints. This is due to the fact, that the refined graphs for nodes outside of the domain would not be reflected by the translation functors, such that this information can not be reconstructed from the free construction again. Unfortunately, not only that the pushout construction cannot be lifted up, the pushout of total model morphisms in $\mathbf{Mod}^T_\mathbf{C}$ is in general neither the pushout in $\mathbf{Mod}_\mathbf{C}$ as the following counterexample shows.

Example 4.4 (counterexample)
The example in Figure 4.7 shows the pushout of the two total model morphisms $\hat{a} : \mathcal{A} \to \mathcal{C}$ and $\hat{b} : \mathcal{A} \to \mathcal{B}$ in the category $\mathbf{Mod}^T_\mathbf{Set}$. The base graphs of the models are drawn by circles, their refined nodes are depicted inside. Nodes are refined to sets and edges to total mappings. The base morphisms of the model morphisms map nodes on the nodes with the same number. The pushout of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod}^T_\mathbf{Set}$ is not a pushout in $\mathbf{Mod}_\mathbf{Set}$: Consider the

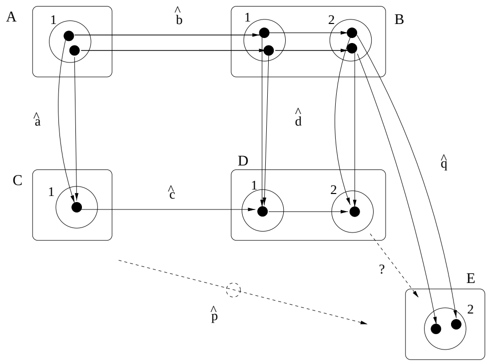


Figure 4.7: *Not each pushout in $\mathbf{Mod}^T_\mathbf{C}$ is pushout in $\mathbf{Mod}_\mathbf{C}$.*

model morphisms $\hat{p}$ and $\hat{q}$, where $\hat{p}$ is completely undefined and $\hat{q}$ is defined only for the node 2. It can be easily checked, that $\hat{p}\circ\hat{a} = \hat{q}\circ\hat{b}$, since $dom(p\circ a) = dom(q\circ b) = \emptyset$. Since pushouts in $\mathbf{Graph}$ are also pushouts in $\mathbf{Graph}^P$ there is a unique morphism $u : D \to E$ such that the required diagrams commute. But, we have two possibilities to define the $\mathbf{C}$-morphism $\alpha^u(2)$. △

This counterexample leads to the conjecture that the category $\mathbf{Mod}_\mathbf{C}$ does not have all pushouts. Since partial model morphisms consist of a partial graph morphism as base morphism and a $\mathbf{C}$-morphisms for the nodes in the domain of the base morphism, the pushout in $\mathbf{Mod}_\mathbf{C}$ for the base morphisms has to be constructed in $\mathbf{Graph}^P$. Therefore, the base graph of the pushout object in Figure 4.7 contains the two nodes and the edge. For its refinement,

we do not have any different choice than that in the example to ensure the commutativity of the diagram and the well-definition of the mapping for the refined edge. But, as explained in the example, this is not the pushout in $\mathbf{Mod}_\mathbf{C}$.

Even if the category may not have all pushouts, I provide conditions for partial model morphisms under which the pushout exists. These conditions ensure that the pushout can be constructed pointwise, i.e. there are no propagating effects as for instance the gluing of the elements in node 2 in Figure 4.7. At first, we introduce the pointwise construction of the pushout for total model morphisms. This construction is originally given in [Tae96] and also used in [TFKV99]. Based on this pointwise pushout construction for total model morphisms, the pushout for partial model morphisms is constructed.

Construction 4.2.9 (pointwise pushout construction)
Given two total injective model morphisms $\hat{a} : \mathcal{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$, the pushout $(\mathcal{D}, \hat{c}, \hat{d})$ of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod}^T_\mathbf{C}$ is constructed as follows:

1. *Construct the pushout of a and b in* $\mathbf{Graph}$.

$$\begin{tikzcd} A \arrow[r, "b"] \arrow[d, "a"'] & B \arrow[d, "d"] \\ C \arrow[r, "c"'] & D \end{tikzcd}$$

2. *The model $\mathcal{D}$ over D is constructed as follows: For all $x \in D_V$ there is at most one $y \in A_V$ with $c_V \circ a_V(y) = x$ since a and b are injective. Then, we can uniquely denote the pushout of $\alpha^a(y)$ and $\alpha^b(y)$ in* $\mathbf{C}$ *by PO_x and by $\alpha^c(a(y)) : \mathcal{C}(a(y)) \to \mathcal{D}(x)$ and $\alpha^d(b(y)) : \mathcal{B}(b(y)) \to \mathcal{D}(x)$ the pushout morphisms. Then, we define $\mathcal{D}$ for all $x \in D$:*

$$\mathcal{D}(x) := \begin{cases} PO_x & \text{, if } \exists y \in A_V : c_V \circ a_V(y) = x \\ \mathcal{C}(z) & \text{, if } \exists z \in C_V - a_V(A_V) : c_V(z) = x \\ \mathcal{B}(z) & \text{, if } \exists z \in B_V - b_V(A_V) : d_V(z) = x \\ IND(PO_{s(x)}, PO_{t(x)}) & \text{, if } \exists e \in A_E : c_E \circ a_E(e) = x \\ \alpha^c(t(e)) \circ \mathcal{C}(e) \circ \alpha^c(s(e))^{-1} & \text{, if } \exists e \in C_E - a_E(A_E) : c_E(e) = x \\ \alpha^d(t(e)) \circ \mathcal{B}(e) \circ \alpha^d(s(e))^{-1} & \text{, if } \exists e \in B_E - b_E(A_E) : d_E(e) = x \end{cases}$$

where $IND(PO_{s(x)}, PO_{t(x)})$ is the morphism induced by the universal pushout property of pushout $PO_{s(x)}$ since $\alpha^d(b(t(e)) \circ \mathcal{B}(b(e)) \circ \alpha^b(s(e)) = \alpha^c(a(t(e))) \circ \mathcal{C}(a(e)) \circ \alpha^a(s(e))$. The pushout morphisms $\hat{c} : \mathcal{C} \to \mathcal{D}$ and $\hat{d} : \mathcal{B} \to \mathcal{D}$ are defined as follows:

$$\alpha^c(x) := \begin{cases} id_{\mathcal{C}(x)} & \text{, if } x \in C_V - a_V(A_V) \\ PO\text{-morphism } \alpha^c(a(y)) \text{ of } PO_{c(x)} & \text{, if } x = a(y) \end{cases}$$

$$\alpha^d(x) := \begin{cases} id_{\mathcal{B}(x)} & \text{, if } x \in B_V - b_V(A_V) \\ PO\text{-morphism } \alpha^d(b(y)) \text{ of } PO_{d(x)} & \text{, if } x = b(y) \end{cases}$$

△

At first, the category $\mathbf{C}$ has to have pushouts to perform this construction. However, even if $\mathbf{C}$ is cocomplete, this construction does not yield always a pushout in $\mathbf{Mod}^T_{\mathbf{C}}$, as the example in Figure 4.7 shows: If we would construct the pushout of $\hat{a}$ and $\hat{b}$ according to the construction in 4.2.9, we do not get the pushout of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod}^T_{\mathbf{C}}$, since the two elements in node 2 would not be glued. Moreover, $\mathcal{D}$ would not be a model in $\mathbf{Mod}^T_{\mathbf{C}}$.
I formulate now so-called *locality conditions* for total model morphisms, that ensure that the construction in 4.2.9 yields a pushout in $\mathbf{Mod}^T_{\mathbf{C}}$. These conditions are originally introduced in [Tae96] for the category $\mathbf{C} = \mathbf{Graph}$, where they are called pushout conditions.

Definition 4.2.10 (locality conditions)
Two total model morphisms $\hat{a} = \langle a, \alpha^a \rangle : \mathcal{A} \to \mathcal{C}$ and $\hat{b} = \langle b, \alpha^b \rangle : \mathcal{A} \to \mathcal{B}$ satisfy the *locality conditions* if

1. a and b are injective,
2. for all $e \in C_E - a(A_E)$, $a(y) = s(e)$ implies that $\alpha^b(y)$ is an isomorphism and
3. for all $e \in B_E - b(A_E)$, $b(y) = s(e)$ implies that $\alpha^a(y)$ is an isomorphism.

$\triangle$

The total model morphisms in Figure 4.7 do not satisfy the locality conditions, since the $\mathbf{C}$-morphism for $\hat{a}$ is not bijective. Its non-injectivity is the reason, that the two elements in node 2 are identified in the pushout D and that we cannot find a unique mediating morphism. If two total model morphisms satisfy the locality conditions, the pushout can be constructed pointwise, i.e. the construction in 4.2.9 yields a pushout in $\mathbf{Mod}^T_{\mathbf{C}}$. This fact is proven in [Tae96] for the special case, where the category $\mathbf{C}$ is the category $\mathbf{Graph}$. Since this proof uses only the fact that $\mathbf{Graph}$ has pushouts, the proof can be generalized. The requirement of isomorphisms in the locality conditions is necessary to ensure the well-definition of the model $\mathcal{D}$ in Construction 4.2.9. For the defintion of the model morphimsm $\hat{c}$ and $\hat{d}$, the property of a coretraction would be sufficient.

Proposition 4.2.11 (local pushout)
Under the notations of Construction 4.2.9, if $\hat{a}$ and $\hat{b}$ satisfy the locality conditions in Definition 4.2.10 and the category $\mathbf{C}$ has pushouts, then $(\mathcal{D}, \hat{c}, \hat{d})$ is a pushout of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod}^T_{\mathbf{C}}$. $\triangle$

Proof
$\mathcal{D}$ is a model over D since each node is refined to an $\mathbf{C}$-object and each edge e is refined to a $\mathbf{C}$-morphism induced by the universal pushout property or the $\mathbf{C}$-morphisms given in $\mathcal{C}$ resp. $\mathcal{B}$. In the first case, the $\mathbf{C}$-morphism is well-defined by the universal pushout property. In the second case, the $\mathbf{C}$-morphism is well-defined due to the locality conditions: If $e \in C_E - a_E(A_E)$ and $s(e) \notin a_E(A_E)$ then $\alpha^c(s(e))$ is the identity and $\alpha^{c^{-1}}(s(e))$ exists. If $e \in C_E - a_E(A_E)$ and $s(e) \in a_E(A_E)$ then $\alpha^c(s(e))$ is an isomorphism due to the locality conditions. Please notice, that it would not be enough to require only a coretraction instead of an isomorphism in the locality conditions to ensure the well-definition of the $\mathbf{C}$-morphism. The same can be shown for edges in $e \in B_E - b_E(A_E)$. The morphisms $\hat{c}$ and $\hat{d}$ are total model morphisms, since for each $e : i \to j \in C_E$, $\alpha^c(j) \circ \mathcal{C}(e) = \mathcal{D}(c(e)) \circ \alpha^c(i)$ for all $e \in a(A_E)$ due to the pushout property of $PO_{c(i)}$ and $PO_{c(j)}$ and for all $e \in C_E - a(A_E)$ due

to construction since $\alpha^c(i)$ is the identity. Analogously it can be shown that α^d is a total model morphism.
The commutativity $\hat{c} \circ \hat{a} = \hat{d} \circ \hat{b}$ follows directly from the construction and it remains to show the universal pushout property. Given a model $\mathcal{X}$ over X and total model morphisms $\hat{f} : \mathcal{C} \to \mathcal{X}$ and $\hat{g} : \mathcal{B} \to \mathcal{X}$ with $\hat{f} \circ \hat{a} = \hat{g} \circ \hat{b}$. Since $c \circ a = d \circ b$ is a pushout diagram in $\mathbf{Graph}$, there is a unique graph morphism $u : D \to X$ such that $u \circ d = g$ and $u \circ c = f$. The total model morphism $\hat{u} = \langle u, \alpha^u \rangle : \mathcal{D} \to \mathcal{X}$ is defined for all $v \in c(a(A_V))$ by the universal morphism induced by PO_x in $\mathbf{C}$. For $v \in D_V$ with $c(z) = x$ (resp. $d(z) = x$) for $z \in C_V - a(A_V)$ (resp. $z \in B_V - b(A_V)$), we define $\alpha^u(v) = \alpha^f(z)$ (resp. $\alpha^u(v) = \alpha^g(z)$). We show next the well definition of $\hat{u}$: α^u is defined for all $x \in D_V$ since c and d are jointly surjective and for each edge $e : i \to j \in D_E$, $\alpha^u(j) \circ \mathcal{D}(e) = \mathcal{X}(u(e)) \circ \alpha^u(i)$ for the following reasons:

1. $e \in c(a(A_E))$: $\alpha^u(i)$ and $\alpha^u(j)$ are $\mathbf{C}$-morphisms induced by a pushout and there is an edge $e_B : v \to w \in B_E$ with $d(e_B) = e$ and an edge $e_C : y \to z \in C_E$ with $c(e_C) = e$. Then holds $\alpha^u(j) \circ \mathcal{X}(e) \circ \alpha^d(v) = \alpha^u(j) \circ \alpha^d(w) \circ \mathcal{B}(e_B) = \alpha^g(w) \circ \mathcal{B}(e_B) = \mathcal{X}(u(e)) \circ \alpha^g(v) = \mathcal{X}(u(e)) \circ \alpha^u(i) \circ \alpha^d(v)$. We can show analogously that $\alpha^u(j) \circ \mathcal{X}(e) \circ \alpha^c(y) = \mathcal{X}(u(e)) \circ \alpha^u(i) \circ \alpha^c(y)$. Since $\alpha^c(y)$ and $\alpha^d(v)$ are jointly epimorph, we have $\alpha^u(j) \circ \mathcal{X}(e) = \mathcal{X}(u(e)) \circ \alpha^u(i)$.
2. $e \in c(C_E - a(A_E))$: Let $e_C \in C_E$ with $c(e_C) = e$ then $\alpha^u(j) \circ \mathcal{D}(e) = \alpha^u(j) \circ \alpha^c(t(e_C)) \circ \mathcal{C}(e_C) \circ \alpha^c(s(e_C))^{-1} = \alpha^f(t(e_C)) \circ \mathcal{C}(e_C) \circ \alpha^c(s(e_C))^{-1} = \mathcal{X}(u(e)) \circ \alpha^f(s(e_C)) \circ \alpha^c(s(e_C))^{-1} = \mathcal{X}(u(e)) \circ \alpha^u(i)$ because of the definition of $\mathcal{D}(e)$ where $\alpha^u(j)$ is either induced or equal to $\alpha^f(t(e_C))$ per definition. Moreover, $\alpha^c(s(e_C))$ is an isomorphism according to the locality conditions.
3. For all $e \in d(B_E - b(A_E))$: $\alpha^u(j) \circ \mathcal{D}(e) = \mathcal{X}(u(e)) \circ \alpha^u(i)$ follows analogously to the previous case.

Next we have to show that $\hat{u} \circ \hat{c} = \hat{f}$ and $\hat{u} \circ \hat{d} = \hat{g}$. $u \circ c = a$ holds because of the pushout properties of D. For all $v \in C_V$ we have $\alpha^u(c(v)) \circ \alpha^c(v) = \alpha^f(v)$ according to the definition of α^u. $\hat{u} \circ \hat{d} = \hat{b}$ can be shown analogously. The uniqueness of $\hat{u}$ holds by definition. $\square$

If two total model morphisms satisfy the locality conditions, the pointwise pushout construction in 4.2.9 yields always a pushout in the category $\mathbf{Mod}^T_{\mathbf{C}}$. However, the pointwise pushout construction may also yield a pushout if the model morphisms do not satisfy the locality conditions. This depends on the underlying category $\mathbf{C}$. I show later on that the pointwise pushout construction in 4.2.9 yields also a pushout for model morphisms in the category $\mathbf{AGr}$ of attributed graphs and attributed graph morphisms used for our case study, that do not satisfy the locality conditions (cf. Proposition 4.4.12).
If the pushout for total model morphisms can be constructed pointwise in the category $\mathbf{Mod}^T_{\mathbf{C}}$, the pointwise pushout is also a pushout in the category $\mathbf{Mod}_{\mathbf{C}}$. This is due to the fact, that a pushout of total graph morphisms is also a pushout in the category of graphs and partial graph morphisms [Löw93] and the pointwise pushout construction allows the unique definition of the universal pushout morphism.

Proposition 4.2.12 (total local pushout is pushout in $\mathbf{Mod}_{\mathbf{C}}$)
Let diagram (1) below constructed pointwise according to Construction 4.2.9. If (1) is a pushout diagram in the category $\mathbf{Mod}^T_{\mathbf{C}}$, then (1) is a pushout diagram in the category

$\mathbf{Mod_C}$.

$$\begin{tikzcd} \mathcal{A} \arrow[r, "\hat{b}"] \arrow[d, "\hat{a}"'] \arrow[dr, phantom, "(1)"] & \mathcal{B} \arrow[d, "\hat{d}"] \arrow[ddr, "\hat{q}"] & \\ \mathcal{C} \arrow[r, "\hat{c}"] \arrow[drr, "\hat{p}"'] & \mathcal{D} \arrow[dr, "\hat{u}"] & \\ & & \mathcal{E} \end{tikzcd}$$

△

Proof
The diagram (1) commutes since it is a pushout diagram in $\mathbf{Mod_C^T}$. We assume now model morphisms $\hat{p}$ and $\hat{q}$ such that $\hat{p} \circ \hat{a} = \hat{q} \circ \hat{b}$. Since a pushout in **Graph** is pushout in $\mathbf{Graph^P}$ [Löw93], there is a unique partial graph morphism $u : D \to E$ such that $u \circ c = p$ and $u \circ d = q$. Then, we define for all $x \in dom(u)$ the $\mathbf{C}$-morphism $\alpha^u(x) : \mathcal{D}(x) \to \mathcal{E}(u(x))$ as follows:

1. If $x \in d(B) \cap c(C)$ there is exactly one $y \in A$ such that $d(b(y)) = c(a(y))$ and $\mathcal{D}(x)$ with $\alpha^c(a(y))$ and $\alpha^d(b(y))$ is pushout of $\alpha^a(y)$ and $\alpha^b(y)$ in $\mathbf{C}$. By premise, $\alpha^q(b(y)) \circ \alpha^b(y) = \alpha^p(a(y)) \circ \alpha^a(y)$ and we define $\alpha^u(x)$ as the universal pushout morphism of $\mathcal{D}(x)$ such that $\alpha^u(x) \circ \alpha^c(a(y)) = \alpha^p(a(y))$ and $\alpha^u(x) \circ \alpha^d(b(y)) = \alpha^q(b(y))$.
2. If $x \in d(B) \backslash c(C)$, there is exactly one $y \in B \backslash b(A)$ such that $d(y) = x$. By construction, $\alpha^d(y) = id_{\mathcal{B}(y)}$ and we define $\alpha^u(x) = \alpha^q(y)$, such that $\alpha^u(x) \circ \alpha^d(y) = \alpha^q(y)$.
3. If $x \in c(C) \backslash b(B)$, there is exactly one $y \in C \backslash a(A)$ such that $c(y) = x$. By construction, $\alpha^c(y) = id_{\mathcal{C}(y)}$ and we define $\alpha^u(x) = \alpha^p(y)$, such that $\alpha^u(x) \circ \alpha^c(y) = \alpha^p(y)$.

Then, for $\hat{u} = \langle u, \alpha^u \rangle : \mathcal{D} \to \mathcal{E}$ holds $\hat{u} \circ \hat{c} = \hat{p}$ and $\hat{u} \circ \hat{d} = \hat{q}$. The uniqueness of $\hat{u}$ follows from the uniqueness of its components. □

Since the locality conditions ensure, that the pointwise pushout construction yields a pushout in $\mathbf{Mod_C^T}$ (cf. Proposition 4.2.11), the following corollary follows immediately.

Corollary 4.2.13 (locality conditions ensure pushout in $\mathbf{Mod_C}$)
Let diagram (1) below constructed pointwise according to Construction 4.2.9. If $\hat{a}$ and $\hat{b}$ satisfy the locality conditions in Definition 4.2.10, then (1) is a pushout diagram in the category $\mathbf{Mod_C}$.

$$\begin{tikzcd} \mathcal{A} \arrow[r, "\hat{b}"] \arrow[d, "\hat{a}"'] \arrow[dr, phantom, "(1)"] & \mathcal{B} \arrow[d, "\hat{d}"] \arrow[ddr, "\hat{q}"] & \\ \mathcal{C} \arrow[r, "\hat{c}"] \arrow[drr, "\hat{p}"'] & \mathcal{D} \arrow[dr, "\hat{u}"] & \\ & & \mathcal{E} \end{tikzcd}$$

△

Proof
If $\hat{a}$ and $\hat{b}$ satisfy the locality conditions, the pointwise pushout construction is a pushout in $\mathbf{Mod_C^T}$ according to Proposition 4.2.11. According to Proposition 4.2.12 follows that (1) is pushout in $\mathbf{Mod_C}$. □

I provide now the pushout construction for partial model morphisms, mainly divided into two steps: In the first step, the partial model morphisms are restricted to total model morphisms, in the second step the pushout of the total model morphism is constructed. The construction represented below yields a pushout in $\mathbf{Mod_C}$ if the pushout of the total model morphisms in $\mathbf{Mod_C^T}$ got by the restriction of the partial ones is also a pushout in $\mathbf{Mod_C}$.

Construction 4.2.14 (pushout in $\mathbf{Mod_C}$)
Given a pair of model morphisms $\hat{a} = \langle a, \alpha^a \rangle : \mathcal{A} \to \mathcal{B}$ and $\hat{b} = \langle b, \alpha^b \rangle : \mathcal{A} \to \mathcal{C}$, the pushout $(\mathcal{D}, \hat{c}, \hat{d})$ of $\hat{a}$ and $\hat{b}$ is constructed in four steps:

1. *Let $a \nabla b$ be the largest subgraph of A which satisfies:*
   (a) *$a \nabla b \subseteq dom(a) \cap dom(b)$ and*
   (b) *for all $x \in a \nabla b$ and $y \in A$, $a(x) = a(y)$ or $b(x) = b(y)$ implies $y \in a \nabla b$.*

   Then, we define $\mathcal{K} : a \nabla b \to G_{\mathbf{C}}$, called the gluing object, *by $\mathcal{K} = \mathcal{A}|_{a \nabla b}$.*
2. *Construction of the domains of $\hat{c}$ and $\hat{d}$.*
   (a) *The domain $dom(c)$ of the partial graph morphism c is the largest subgraph of C, who is contained in $(C - b(A)) \cup b(a \nabla b)$. Then, $\bar{\mathcal{C}} : dom(c) \to G_{\mathbf{C}}$ is defined by $\bar{\mathcal{C}} := \mathcal{C}|_{dom(c)}$ and $\hat{b}|_{\mathcal{K}} = \langle b|_{a \nabla b}, \alpha^{b|_{\mathcal{K}}} \rangle$ is defined by $\alpha^{b|_{\mathcal{K}}}(x) = \alpha^b(x)$ for all $x \in a \nabla b$.*
   (b) *Symmetrically, $dom(d)$ is the largest subgraph of B, who is contained in $(B - a(A)) \cup a(a \nabla b)$. Then $\bar{\mathcal{B}} := \mathcal{B}|_{dom(d)}$ and $\hat{a}|_{\mathcal{K}} = \langle a|_{a \nabla b}, \alpha^{a|_{\mathcal{K}}} \rangle$ is defined by $\alpha^{a|_{\mathcal{K}}}(x) = \alpha^a(x)$ for all $x \in a \nabla b$.*
3. *Construct the pushout $(\mathcal{D}, \hat{x}, \hat{y})$ of $\hat{a}|_{\mathcal{K}}$ and $\hat{b}|_{\mathcal{K}}$ in the category $\mathbf{Mod_C^T}$.*
4. *Construction of the pushout morphisms $\hat{c} = \langle c, \alpha^c \rangle : \mathcal{C} \to \mathcal{D}$ and $\hat{d} = \langle d, \alpha^d \rangle : \mathcal{B} \to \mathcal{D}$: $c = [C \hookleftarrow dom(c) \xrightarrow{x} D]$ and $\alpha^c(v) = \alpha^x(v)$ for all $v \in dom(c)_V$. Analog, $d = [D \hookleftarrow dom(d) \xrightarrow{y} D]$ and $\alpha^d(v) = \alpha^y(v)$ for all $v \in dom(d)_V$.*

$$\begin{tikzcd} A \arrow[rr, "a"] \arrow[dd, "b"'] & & B & \\ & a\nabla b \arrow[ul, hook] \arrow[r, "a|_{a\nabla b}"] \arrow[d, "b|_{a\nabla b}"'] \arrow[dr, phantom, "PO"] & dom(d) \arrow[u, hook] \arrow[d, "y"] & dom(g) \arrow[l, hook'] \arrow[ul, hook] \arrow[dd] \\ C & dom(c) \arrow[l, hook'] \arrow[r, "x"'] & D \arrow[dr, "u"] & \\ & dom(f) \arrow[u, hook] \arrow[ul, hook] \arrow[rr] & & E \end{tikzcd}
\qquad
\begin{tikzcd} \mathcal{A} \arrow[rr, "\hat{a}"] \arrow[dd, "\hat{b}"'] & & \mathcal{B} \arrow[dddr, "\hat{g}"] & \\ & \mathcal{K} \arrow[ul, hook] \arrow[r, "\hat{a}|_{\mathcal{K}}"] \arrow[d, "\hat{b}|_{\mathcal{K}}"'] \arrow[dr, phantom, "PO"] & \bar{\mathcal{B}} \arrow[u, hook] \arrow[d, "\hat{y}"] & \\ \mathcal{C} \arrow[drrr, "\hat{f}"'] & \bar{\mathcal{C}} \arrow[l, hook'] \arrow[r, "\hat{x}"'] & \mathcal{D} & \\ & & & \mathcal{E} \end{tikzcd}$$

△

Definition 4.2.15 (restricted morphisms)
I call the total model morphisms $\hat{a}|_{\mathcal{K}}$ and $\hat{b}|_{\mathcal{K}}$ the *restricted morphisms of* $\hat{a}$ and $\hat{b}$. △

Proposition 4.2.16 (pushout in $\mathbf{Mod_C}$)
Under the notations of the Construction 4.2.14, $(\mathcal{D}, \hat{c}, \hat{d})$ is a pushout of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod_C}$ if the pushout of the restricted morphisms $\hat{a}|_{\mathcal{K}}$ and $\hat{b}|_{\mathcal{K}}$ in $\mathbf{Mod}^T_\mathbf{C}$ is also pushout in $\mathbf{Mod_C}$. △

Proof
Commutativity: The diagram $c \circ b = d \circ a$ is a pushout in $\mathbf{Graph}^P$, such that $d \circ a = c \circ b$ with $dom(d \circ a) = dom(c \circ b) = a \nabla b$ (cf. [Löw93]). For each $x \in a\nabla b$, $\alpha^d(a(x)) \circ \alpha^a(x) = \alpha^y(a(x)) \circ \alpha^{a|_\mathcal{K}}(x) = \alpha^x(b(x)) \circ \alpha^{b|_\mathcal{K}}(x) = \alpha^c(b(x)) \circ \alpha^b(x)$.
Universal property: Let $\hat{f} = \langle f, \alpha^f \rangle : \mathcal{C} \to \mathcal{E}$ and $\hat{g} = \langle g, \alpha^g \rangle : \mathcal{B} \to \mathcal{E}$ be two model morphisms in $\mathbf{Mod_C}$ such that $\hat{f} \circ \hat{b} = \hat{g} \circ \hat{a}$.
If we assume

$$(*)\ \hat{f} \circ \hat{b}|_\mathcal{K} = \hat{g} \circ \hat{a}|_\mathcal{K},$$

there is a unique model morphism $\hat{u} = \langle u, \alpha^u \rangle : \mathcal{D} \to \mathcal{E}$ such that $\hat{u} \circ \hat{x} = \hat{f}$ and $\hat{u} \circ \hat{y} = \hat{g}$ due to the premise that the pushout of $\hat{a}_\mathcal{K}$ and $\hat{b}_\mathcal{K}$ is a pushout in $\mathbf{Mod_C}$. By construction of $\hat{c}$ and $\hat{d}$, we get $\hat{u} \circ \hat{c} = \hat{f}$ and $\hat{u} \circ \hat{d} = \hat{g}$.
To prove (*) we need $x \in dom(f \circ b) \Rightarrow x \in a\nabla b$: If $x \in dom(f \circ b)$ then $x \in dom(b)$ and $b(x) \in dom(f)$. Since $dom(f) \subseteq dom(c)$ ([Löw93]), we get $b(x) \in dom(c)$. If we assume $x \notin a\nabla b$ then $b(x) \notin b(a\nabla b)$. Then, $b(x) \notin (C - b(A)) \cup b(a\nabla b)$ that implies $x \notin dom(c)$, what is a contradiction. We can show analogously that $x \in dom(g \circ a) \Rightarrow x \in a\nabla b$. Due to this fact, we are able to prove (*):

- $x \in dom(f \circ b|_{a\nabla b}) \Leftrightarrow b(x) \in dom(f)$ and $x \in a\nabla b \Leftrightarrow x \in dom(f \circ b) \Leftrightarrow x \in dom(g \circ a) \Leftrightarrow x \in dom(g \circ a|_{a\nabla b})$.

  For each $x \in dom(f \circ b|_{a\nabla b})$, we have $f(b|_{a\nabla b}(x)) = f(b(x)) = g(a(x)) = g(a|_{a\nabla b}(x))$.

- For each $x \in dom(f \circ b_{a\nabla b})$, $\alpha^f(b|_\mathcal{K}(x)) \circ \alpha^{b|_\mathcal{K}}(x) = \alpha^f(b(x)) \circ \alpha^b(x) = \alpha^g(a(x)) \circ \alpha^a(x) = \alpha^g(a|_\mathcal{K}(x)) \circ \alpha^{a|_\mathcal{K}}(x)$.

□

If the restricted morphisms satisfy the locality conditions, their pushout in $\mathbf{Mod}^T_\mathbf{C}$ can be constructed pointwise and is a pushout in $\mathbf{Mod_C}$ as well. Therefore, the pushout of two partial model morphisms exist in $\mathbf{Mod_C}$, if their restricted morphisms satisfy the locality conditions. Please notice, that the locality conditions do not have to be satisfied from the partial model morphism, e.g. the partial base morphism may be non-injective.

Theorem 4.2.17 (pushout in $\mathbf{Mod_C}$)
Under the notations of the Construction 4.2.14, $(\mathcal{D}, \hat{c}, \hat{d})$ is pushout of $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod_C}$ if

1. *the pointwise pushout construction in 4.2.9 for the restricted morphisms $\hat{a}|_\mathcal{K}$ and $\hat{b}|_\mathcal{K}$ yields a pushout in $\mathbf{Mod}^T_\mathbf{C}$,*
2. *the restricted morphisms $\hat{a}|_\mathcal{K}$ and $\hat{b}|_\mathcal{K}$ satisfy the locality conditions in Definition 4.2.10.*

△

Proof
The first point follows from Proposition 4.2.12 and Proposition 4.2.16. The second point follows from Corollary 4.2.13 and Proposition 4.2.16. □

Example 4.5 (pushout for partial model morphisms in Set)
Using partial base morphisms, the pushout captures not only the intuitive idea of gluing but additionally that of deletion, shown in Figure 4.8. The presentation of models in this figure shows only the refined graphs and refined graph morphisms for the base graph, but not the base graphs themselves. The base graphs, however, can be constructed form this representation. The gluing object $\mathcal{K}$ contains only the refined graph for the node 3 in $\mathcal{A}$,

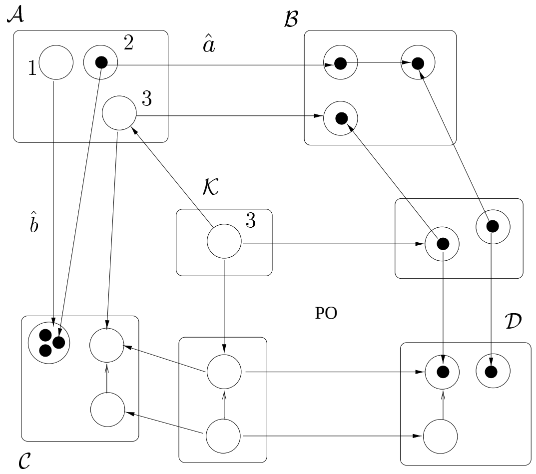


Figure 4.8: *A pushout of model morphisms with partial base morphisms.*

since node 1 is not in the intersection of the domains from a and b and node 2 is not in the gluing object since 1 is not in the gluing object and b maps 1 and 2 on the same node in C. The restricted morphisms $\hat{a}|_\mathcal{K}$ and $\hat{b}|_\mathcal{K}$ on $\mathcal{K}$ are total model morphisms that satisfy the locality conditions. Please notice, that the partial model morphisms would not satisfy the locality conditions, since b is not injective and the mapping from node 2 in $\mathcal{A}$ to $\mathcal{C}$ is not bijective. The pushout morphisms are given by the model morphisms with partial base morphisms from $\mathcal{C}$ to $\mathcal{D}$ resp. from $\mathcal{B}$ to $\mathcal{D}$. △

4.3 Distributed Graph Transformation in the Single Pushout Approach with local C-states

This section presents the single pushout (SPO) approach to distributed graph transformation with local **C**-states intended for the specification of distributed systems and their rule-based manipulation. States of the distributed system are specified by models in **C**, where the base graph is equipped by an additional type information. The typed base graph of a model represents the topology of the distributed system and the model refines the components of this topology to objects of the category **C** representing their local states. State transformations are specified by *distributed productions*, that are partial model morphisms in **C** preserving the type information of the base graphs. The application of distributed productions is based on a pushout in the category $\mathbf{Mod_C}$.

4.3.1 The category of Distributed Graphs and Distributed Morphisms

In this work, I consider a distributed system consisting of concurrent processes communicating via shared ports (or channels, interfaces). The topological description of a distributed system consisting of processes and ports follows the approach of *Grammars for Distributed Systems* (GDS) of Montanari and Rossi [MPR99]. Whereas Montanari and Rossi describe the topology by hyperedges, where hyperedges are processes and nodes are ports, I represent the topology of a distributed system by a typed graph in the type graph DS for distributed systems shown in Figure 4.9. The type graph DS provides two types of nodes, namely *processes*

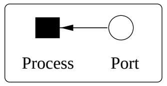


Figure 4.9: *Type graph DS for distributed systems.*

and *ports*. Edges are pointing from ports to processes, such that connections between processes are possible only via ports. Typed graphs in DS can be seen as hypergraphs, whereas partial DS-typed graph morphisms are more general than hypergraph morphisms due to their partiality. Furthermore, the deletion of ports is not possible in [MPR99], whereas our approach allows arbitrary network configurations. In addition, the concept of typed graphs for describing the topology of a distributed system simplifies the theoretical considerations later on.

Processes and ports within a distributed system may be of different type as well. This type information is represented by a *network type graph* NTG, that is a graph typed in DS. The typing in DS ensures that the network type graph NTG has the structure of a distributed system consisting of processes connected via ports. Therefore, each typed graph in the network type graph NTG is also a typed graph in DS. By defining this kind of two-level typing we achieve a kind of meta-typing as presented in [Löw98].

Definition 4.3.1 (network type graphs)
Each typed graph $t_{NTG} : NTG \to DS$ of the category $\mathbf{Graph^P}_{DS}$ is a *network type graph*. We define by $PType(NTG) = \{x \in NTG_V | t_{NTG}(x) = process\}$ the set of *process types* of NTG and by $PortType(NTG) = \{x \in NTG_V | t_{NTG}(x) = port\}$ the set of *port types* of NTG. I write shortly $PType$ resp. $PortType$ if the typed graph NTG is fixed and clear from the context. △

For the following representation of network type graphs $t_{NTG} : NTG \to DS$, we depict only the graph NTG. The typing in DS can be reconstructed from the graphical layout of nodes and edges.

Example 4.6 (network type graph)
The network type graph NTG in the Figure 4.10 on the left-hand side consists of two types for processes, namely process type A and B. There exists one port type and it can be easily checked that the graph NTG is a typed graph in DS. On the right-hand side of the figure a typed graph in NTG is shown. It consists of two processes of type A and four B-processes. As this graph is a typed graph in NTG, it is as well a typed graph in DS. △

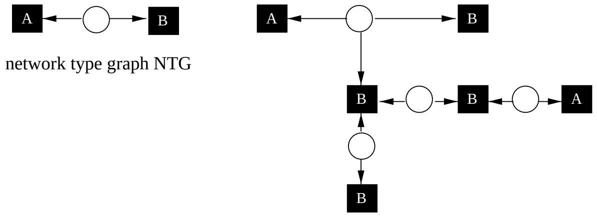


Figure 4.10: *A network type graph NTG and a typed graph in NTG.*

If we specify a concrete distributed system, a network type graph NTG is fixed. For instance, the network type graph in Example 2.2 shows the network type graph for the distributed configuration management system of our case study. The category $\mathbf{Graph^P}_{NTG}$ (cf. Def. 4.1.2) of typed graphs in NTG and partial NTG-typed morphisms provides the possible graphs for the topology and the possible relations between topologies.

Definition 4.3.2 (category of network graphs)
Given a network type graph $t_{NTG} : NTG \to DS$, the category $\mathbf{Graph^P}_{NTG}$ is called *category of network graphs* with respect to NTG. Objects of $\mathbf{Graph^P}_{NTG}$ are called *network graphs* and morphisms are called *network graph morphisms*. If the network type graph is fixed and no confusion is possible, I denote network graphs $t_G : G \to NTG$ shortly by G. △

For the following considerations I assume a fixed network type graph NTG. I implicitly assume that each network graph and each network morphism occurring in the sequel is a network graph resp. network morphism with respect to NTG.

For the specification of local states of processes, objects of a category **C** are taken. The choice of the actual category depends on the application the designer has in mind. Possible categories may be graphs, partial algebras, ALR-graphs etc. The category for the local states of the case study in Section 2.3 are *attributed graphs*. We will explain this category in full detail in Section 4.4. However, not only the processes get a local state, but also the ports. The local states of ports indicate the data used in the communication between processes. The topology represented by a network graph and the local states represented by **C**-objects are integrated in a *distributed graph* $\hat{G}$.

Definition 4.3.3 (distributed graph)
Given a network type graph NTG and a category **C**, a *distributed graph over G with local* **C**-*states* is a pair $\hat{G} = \langle t_G, \mathcal{G} \rangle$, where $t_G : G \to NTG$ is a network graph with respect to NTG and $\mathcal{G} : G \to G_{\mathbf{C}}$ is a model in **C**. △

A *distributed morphism* between two distributed graphs is a partial model morphism between the models of the distributed graphs (cf. Definition 4.2.8). The partial base morphism between the network graphs has to preserve additionally the process and port types.

Definition 4.3.4 (distributed morphism)
Given a network type graph NTG, a category $\mathbf{C}$ and two distributed graphs $\hat{G} = \langle t_G, \mathcal{G}\rangle$ and $\hat{H} = \langle t_H, \mathcal{H}\rangle$ over G resp. H, a *distributed morphism* $\hat{f} : \hat{G} \to \hat{H}$ is a partial model morphism $\hat{f} = \langle f, \alpha^f\rangle : \mathcal{G} \to \mathcal{H}$ in $\mathbf{C}$ between the models $\mathcal{G}$ and $\mathcal{H}$ such that $f : G \rightharpoonup H$ is a network graph morphism with respect to NTG. A distributed morphism $\hat{f}$ is called *total* if the model morphism $\hat{f}$ is total. △

Examples of distributed graphs and distributed morphism can be found in our case study in Section 2.4.1 and the remainder of Chapter 2. Distributed graphs and morphisms are also used in Chapter 3.

All distributed graphs and distributed morphisms with respect to a network type graph NTG and a category $\mathbf{C}$ form a category $\mathbf{DGr_C}(NTG)$. Composition of distributed morphisms is defined by composition of model morphisms in category $\mathbf{Mod_C}$ and identities are defined as identities in $\mathbf{Mod_C}$, since identities preserve types obviously. In the remainder of this chapter, we assume a cocomplete category $\mathbf{C}$.
Pushouts of distributed morphisms in the category $\mathbf{DGr_C}(NTG)$ are constructed by the pushout of the partial model morphisms in the category $\mathbf{Mod_C}$. The typing of the pushout object results as unique extension. Of course, the pushout of distributed morphisms exists only if the pushout exists for the partial model morphisms in $\mathbf{Mod_C}$.

Proposition 4.3.5 (pushouts in $\mathbf{DGr_C}(NTG)$)
The distributed morphisms $\hat{a} : \hat{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$ have a pushout in $\mathbf{DGr_C}(NTG)$ if the underlying partial model morphisms $\hat{a}$ and $\hat{b}$ have a pushout in $\mathbf{Mod_C}$. △

Proof
The pushout of the distributed morphisms $\hat{a}$ and $\hat{b}$ is constructed as the pushout of the underlying partial model morphisms $\hat{a}$ and $\hat{b}$ in the category $\mathbf{Mod_C}$ (cf. Theorem 4.2.17). The typing of the base graph of the pushout object results as unique extension of the resulting diagrams got by the pushout construction $\hat{a}$ and $\hat{b}$ in $\mathbf{Mod_C}$. The pushout properties in $\mathbf{DGr_C}(NTG)$ follow from the pushout properties in $\mathbf{Mod_C}$. □

Corollary 4.3.6
The distributed morphisms $\hat{a} : \hat{A} \to \hat{C}$ and $\hat{b} : \hat{A} \to \hat{B}$ have a pushout in $\mathbf{DGr_C}(NTG)$ if the restricted morphisms $\hat{a}|_{\mathcal{K}}$ and $\hat{b}|_{\mathcal{K}}$ of $\hat{a}$ and $\hat{b}$ (cf. Definition 4.2.15) satisfy the locality conditions in Definition 4.2.10 and category $\mathbf{C}$ has pushouts. △

Proof
Follows directly form Proposition 4.3.5 and Theorem 4.2.17. □

4.3.2 Conditional Transformation of Distributed Graphs

I introduce now the transformation of distributed graphs by *distributed productions* ($p : \hat{L} \xrightarrow{\hat{r}} \hat{R}$) consisting of a *production name* p and a distributed morphism $\hat{r}$ as defined in Definition 4.3.4. The distributed graph $\hat{L}$ is called *left-hand side*, the distributed graph $\hat{R}$ the *right-hand side* of the production. The left-hand side describes which objects a distributed graph must contain such that the production can be applied. The distributed morphism $\hat{r}$ indicates which objects in the left-hand side correspond to which one in the right-hand side. Objects where the distributed morphism is undefined are deleted. All objects which are preserved by the morphism form the application context. If an object in the right-hand side of the production has no pre-image under the morphism, it is added.

Definition 4.3.7 (distributed production)
A *distributed production* $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ consists of a *production name* p and a distributed morphism $\hat{r}$. △

Examples of distributed productions are given in our case study in Section 2.4, when we introduce distributed productions in Subsection 2.4.1 and when we introduce the process grammars of the case study in Subsection 2.4.2.1 and Subsection 2.4.2.2.

To apply a distributed production to a distributed graph $\hat{G}$, one has to make sure that $\hat{G}$ contains at least all objects necessary for performing the desired operations and the application context. These are exactly the objects of the production's left-hand side $\hat{L}$. Hence I introduce the notion of a *match* as a total distributed morphism $\hat{m} : \hat{L} \to \hat{G}$ matching the left-hand side of the production with a part of $\hat{G}$. The match must be total, otherwise not all needed objects of the left-hand side have a corresponding image in the distributed graph $\hat{G}$. In addition, I require that the restricted morphisms of the production morphism $\hat{r}$ and the match $\hat{m}$ satisfy the locality conditions to ensure the characterization of the production application by a pushout in the category $\mathbf{DGr_C}(NTG)$. This condition is called *gluing condition* following the notation of the gluing condition in the double pushout approach to graph transformation [CMR+97], that also ensures the existence of a direct derivation.

Definition 4.3.8 (gluing condition, match)
A total distributed morphism $\hat{m} : \hat{L} \to \hat{G}$ satisfies the *gluing condition* for a distributed production $p : (\hat{r} : \hat{L} \to \hat{R})$ if the restricted morphisms $\hat{m}|_{\mathcal{K}}$ and $\hat{r}|_{\mathcal{K}}$ of $\hat{m}$ and $\hat{r}$ (cf. Definition 4.2.15) satisfy the locality conditions in Definition 4.2.10.
A *match* for a distributed production $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ in some distributed graph $\hat{G}$ is a total distributed morphism $\hat{m} : \hat{L} \to \hat{G}$ that satisfies the gluing condition. △

The application of a distributed production $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ at a match $\hat{m} : \hat{L} \to \hat{G}$ is characterized by the pushout of the production morphism $\hat{r}$ and the match in the category $\mathbf{DGr_C}(NTG)$. The existence of the pushout is ensured by the gluing condition for the match and Corollary 4.3.6. The result is a derived distributed graph $\hat{H}$, the pushout diagram is called a *direct derivation*. The derived graph may be used for further production applicationss such that we get a sequence of direct derivations.

Definition 4.3.9 (direct derivation, derivation)
Let $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ be a distributed production and $\hat{m} : \hat{L} \to \hat{G}$ be a match, the *direct derivation* from $\hat{G}$ to $\hat{H}$ with p at $\hat{m}$, written $\hat{G} \overset{p,\hat{m}}{\Longrightarrow} \hat{H}$, is a pushout of $\hat{r}$ and $\hat{m}$ in $\mathbf{DGr_C}(NTG)$.
Let $(p : \hat{r})_{p \in P}$ be a P-indexed set of distributed productions and $Der(P) = \{\hat{G} \overset{p,\hat{m}}{\Longrightarrow} \hat{H} | p \in P\}$ be the set of all possible direct derivations using productions of P, a *derivation sequence* or short *derivation* is a total mapping $\rho : I \to Der(P)$ from a possibly infinite prefix $I = \{0, 1, ...\}$ of $\mathbb{N}$ to $Der(P)$, such that for all $i, j \in I$ with $\rho(i) = (\hat{G}_i \overset{p_i,\hat{m}_i}{\Longrightarrow} \hat{H}_i)$, $\rho(j) = (\hat{G}_j \overset{p_j,\hat{m}_j}{\Longrightarrow} \hat{H}_j)$ holds that $j = i + 1$ implies $\hat{G}_j = \hat{H}_i$. We denote a derivation sequence by $\rho = (\hat{G}_0 \overset{p_0,\hat{m}_0}{\Longrightarrow} \hat{G}_1 \overset{p_1,\hat{m}_1}{\Longrightarrow} ...)$. △

If we compare these definitions with the explanations of the direct derivation in Chapter 2, Section 2.4.1, we see that the match in Section 2.4.1 does not have to satisfy the gluing condition. This is based on the fact that we provide in Definition 4.3.8 and Definition 4.3.9 existence conditions for a direct derivation in category $\mathbf{DGr_C}(NTG)$ for an arbitrary category $\mathbf{C}$ and Chapter 2 uses a concrete category $\mathbf{C}$, namely the category $\mathbf{AGr}$ of attributed graphs. In this case, the direct derivation exists also if the match does not satisfy the gluing condition. We will show this in Section 4.4, Theorem 4.4.13.

4.3.2.1 Application Conditions for Distributed Productions

Distributed productions describe intuitively how given distributed graphs shall be transformed into derived distributed graphs. A distributed production is applicable if there is a match of the left-hand side in the given distributed graph, i.e. we have to find the objects of the left-hand side in the given distributed graph. I introduce now the possibility to specify additional *application conditions* for each particular distributed production that restrict the application of productions. Application conditions for productions were introduced in [HHT96,HMTW95], in [Fis98] they are given in an HLR-approach.

An application condition for a distributed production consists of a finite set of *conditional constraints* over a distributed graph. The idea of a conditional constraint is the requirement of structures in a distributed graph $\hat{G}$ under the assumption of the existence of a certain sub-structure in $\hat{G}$.

Definition 4.3.10 (conditional constraint, application condition)
A *conditional constraint* $cc_{\hat{L}} = (\hat{x} : \hat{L} \to \hat{X}, c_{\hat{X}})$ over a distributed graph $\hat{L}$ consists of

- a total distributed morphism $\hat{x} : \hat{L} \to \hat{X}$ and
- an I-indexed set of total distributed morphisms $c_{\hat{X}} = (\hat{c}_i : \hat{X} \to \hat{C}_i)_{i \in I}$, where I may be empty as well.

A total distributed morphism $\hat{m} : \hat{L} \to \hat{G}$ *satisfies* a conditional constraint $cc_{\hat{L}}$ over $\hat{L}$, short $\hat{m} \models cc_{\hat{L}}$, if for each total distributed morphism $\hat{p} : \hat{X} \to \hat{G}$ with $\hat{p} \circ \hat{x} = \hat{m}$ there is an i in

I and a total distributed morphism $\hat{q} : \hat{C}_i \to \hat{G}$ such that $\hat{q} \circ \hat{c}_i = \hat{p}$.

$$\begin{tikzcd} \hat{C}_i \arrow[drr, "\hat{q}"'] & \hat{X} \arrow[l, "\hat{c}_i"'] \arrow[dr, "\hat{p}"] & \hat{L} \arrow[l, "\hat{x}"'] \arrow[d, "\hat{m}"] \\ & & \hat{G} \end{tikzcd}$$

An *application condition* $A_{\hat{L}}$ over $\hat{L}$ consists of a finite set of conditional constraints over $\hat{L}$. A total distributed morphism $\hat{m} : \hat{L} \to \hat{G}$ *satisfies* $A_{\hat{L}}$, short $\hat{m} \models A_{\hat{L}}$, if $\hat{m} \models cc_{\hat{L}}$ for each $cc_{\hat{L}}$ in $A_{\hat{L}}$. △

I require that the model morphisms of a conditional constraint are total, since a constraint shall express the requirement of certain structures in a distributed graph similar to a match for a distributed production. If we would allow distributed morphisms with partial base morphisms this requirement would be weakened.

Example 4.7 (application condition)
I consider for this example the category $\mathbf{DGr_{Set}}(NTG)$, where NTG is the network type graph in Figure 4.10 and local states are modeled by sets and relations between local states by total mappings. The application condition in Figure 4.11 consists of one conditional constraint over a distributed graph $\hat{L}$. I have chosen a representation where the base graphs and base morphisms are not explicitly shown, but only the refined graphs and refined morphisms. The distributed graph $\hat{L}$ consists of one process (of an arbitrary type) where the local state

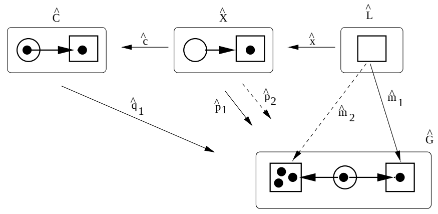


Figure 4.11: *An application condition consisting of one conditional constraint.*

is the empty set. The graphical constraint over $\hat{L}$ in the figure requires that whenever there is a connected port and an element in the local state of the process in $\hat{L}$, the element is also in the local state of the port. Consider now the distributed graph $\hat{G}$ consisting of two processes connected by means of a port. We can find two total distributed morphisms from $\hat{L}$ to $\hat{G}$ that are indicated in the figure by the morphisms $\hat{m}_1$ and $\hat{m}_2$. The morphism $\hat{m}_1$ satisfies the constraint since there exists only one total distributed morphism $\hat{p}_1$ and a total distributed morphism $\hat{q}_1$ such that the diagrams commute. For the morphism $\hat{m}_2$, however, we are able to find three distributed morphisms from $\hat{X}$ to $\hat{G}$ such that the right diagram commutes. For two of them there does not exist a distributed morphism from $\hat{C}$ to $\hat{G}$ such that the left diagram commutes. Therefore, the distributed morphism $\hat{m}_2$ does not satisfy the conditional constraint resp. the application condition. △

A *conditional distributed production* is a distributed production together with an application condition over the left-hand side of the production. Examples of conditional distributed productions can be found in Chapter 2, Subsection 2.4.1.1 and in the process grammars of the case study in Subsection 2.4.2.1 and Subsection 2.4.2.2. These examples show the special case of *negative application conditions* that forbid certain structures in the context of a match. A negative application condition consists of conditional constraints $cc_{\hat{L}} = (\hat{x}, c_{\hat{X}})$ with $c_{\hat{X}} = (\hat{c}_i)_{i \in I}$ where the index set I is empty. A conditional production can be applied to a distributed graph $\hat{G}$ if one can find a match $\hat{m}$ in $\hat{G}$ and $\hat{m}$ satisfies the application condition of the production.

Definition 4.3.11 (conditional distributed production)
A *conditional distributed production* is a pair $p : (\hat{r}, A(p))$ consisting of a distributed production $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ and an application condition $A(p)$ over $\hat{L}$.
The conditional production is *applicable* to a distributed graph $\hat{G}$ at match $\hat{m} : \hat{L} \to \hat{G}$ if $\hat{m}$ satisfies $A(p)$. In this case, the *conditional direct derivation* is given by the direct derivation $\hat{G} \overset{p,\hat{m}}{\Rightarrow} \hat{H}$ of the distributed production p at $\hat{m}$. △

Several (conditional) distributed productions are combined in a *grammar for distributed graphs* that additionally contains an initial distributed graph. I consider the derivation sequences starting at the initial graph which can be generated by the productions of the grammar. This set of derivation sequences will be used for specifying the behavior of a system later on.

Definition 4.3.12 (grammar for distributed graphs)
A *grammar for distributed graphs* is a pair $\mathcal{GG} = \langle \hat{G}_{init}, (p : (\hat{r}, A(p)))_{p \in P} \rangle$ where $\hat{G}_{init}$ is a distributed graph and $(p : (\hat{r}, A(p)))_{p \in P}$ is an P-indexed set of conditional distributed productions. I denote by $Der(\mathcal{GG}) = \{\rho = (\hat{G}_0 \overset{p_0}{\Rightarrow} \hat{G}_1 \overset{p_1}{\Rightarrow} ...) | \hat{G}_0 = \hat{G}_{init}, p_i \in P\}$ the set of all derivation sequences $\rho : I \to Der(P)$ starting at $\hat{G}_0$ using conditional distributed productions of P. △

4.3.3 Componentwise Specification of Distributed Systems

I consider in this section the componentwise specification of distributed systems. For the following considerations, I fix a network type graph NTG and assume implicitly that each distributed graph occurring in the following is a distributed graph, where the network graph is a network graph with respect to NTG. I abbreviate the set of process types $PType(NTG)$ of the network type graph NTG by $PType$.

I start by the definition of a distributed production slightly different from that one given in Definition 4.3.7, that I call *process production*. A process production incorporates the locality and visibility aspects of a distributed system in the following sense: Each process has its own local state that is invisible for other processes. When a process wants to know parts of the local state of another process, it has to communicate with the process over a common port. Communication takes place by writing the data into the port. The local state of ports is visible for all processes in the distributed system. If we consider this principle of visibility of local states, the definition of a distributed production as given in Definition 4.3.7 is unsatisfactory, since the left-hand side may contain several process nodes (see the example in Figure 4.12 at the bottom). To apply such a production to a distributed graph, we have to find for each process in the left-hand side a process in the distributed graph such that

the local states of the production processes match the local states in the distributed graph. To find this match for several processes at the same time requires a global view on the distributed system. Since such a global view does not exist in a distributed system, I restrict distributed productions to process productions. The left-hand side of a process production contains one and only one process node possibly together with some ports. To ensure a local application of a process production, its application condition has to be restricted as well. The structures checked in a process application condition refer only to the local state of the process itself and to ports, but not to other processes. Altogether, a process production ensures that the boundary of visibility of a process in a distributed system is not violated by searching a match for its application.

Definition 4.3.13 (conditional process production)
A conditional distributed production $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R}, A(p))$ with $\hat{L} = \langle t_L, \mathcal{L} \rangle$ and $\hat{R} = \langle t_R, \mathcal{R} \rangle$ is a *conditional process production* of type $x \in PType$ if and only if

- L contains one and only one process node, i.e. there is a $v \in L_V$ with $t_{NTG}(t_L(v)) =$ *process* and for each node $v' \in L_V$ with $t_{NTG}(t_L(v')) = process$ follows $v = v'$,
- the process node v is of type x, i.e. $t_L(v) = x$,
- r does not create new process nodes, i.e. if there is a $v'' \in R_V$ with $t_{NTG}(t_R(v'')) =$ *process* then $v'' = r(v)$ and
- for each conditional constraint $cc_{\hat{L}} = (\hat{x} : \hat{L} \to \hat{X}, (\hat{c}_i : \hat{X} \to \hat{C}_i)_{i \in I}) \in A(p)$, $v'' \in X_V$ with $t_{NTG}(t_X(v'')) = process$ implies $v'' = x(v)$ and for each $i \in I$, $v'' \in C_{i_V}$ with $t_{NTG}(t_{C_i}(v'')) = process$ implies $v'' = c_i(x(v))$.

△

A process production can not directly create new processes by creating new process nodes. The creation of a new process node takes place by the synchronization of a port creating process production and a special distributed production, called *start production*, that contains the new process node. I will come back to this point later on. I will provide next an example of a process production. More examples can be found in Chapter 2 in the Subsections2.4.2.1 and 2.4.2.2.

Example 4.8 (process production)
Figure 4.12 shows two distributed productions. The production at the upper half of the figure is a conditional process production of type A, where the application condition for this production consists of one conditional constraint over $\hat{L}$. The process production creates a new port node together with a new element in its local state. The application condition requires that this creation can be performed only if there does not exist already an element in the existing port of A. This production is a process production, since all distributed graphs of the production as well as the application condition contain exactly one process node of type A. This production is applicable without knowledge of local states of remote processes occurring in the distributed system. The distributed production at the bottom of the figure is not a process production. The distributed graphs of this production contain a process node of type A and B. In order to apply this production, the local states of a process of type A and of a process of type B has to be known at the same time, what is impossible in a distributed system. △

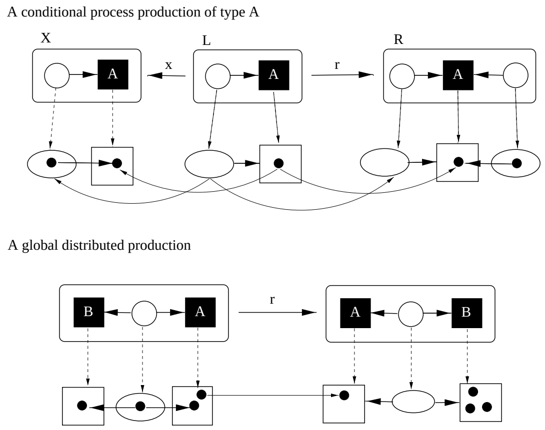


Figure 4.12: *Example of a process production of type A and a global production.*

A *process grammar* of type x specifies all possible moves of process instances of type x by a set of process productions of this type. Instead of a start graph, I add a so-called *start production of type* x to the process grammar. A start production of type x is a distributed production, where its left-hand side contains the ports necessary for starting the process and the right-hand side contains the process of type x embedded into the ports, that describes the initial state of a process of type x. A start production reflects better than a start graph the fact, that instances of processes can be created dynamically and do not have to exist necessarily from the beginning of the distributed system. Please notice, that the start production is not a process production, since there does not is a process node in its left-hand side. Examples of start productions are shown in Figure 4.13 and Figure 4.14 and in the case study in Chapter 2 (Figure 2.12 and Figure 2.21). Start productions will also be used to describe the generation of processes in the distributed system. If an existing process creates a new process, it creates only the necessary ports. A synchronization of this port creating production and the start production results in the creation of the process in the distributed system, that is triggered by the process that has created the port. We will come back to this point later on.

Definition 4.3.14 (start production)
Given a network type graph NTG, a *start production* of type $x \in PType(NTG)$ is a distributed production $st(x) : (\hat{L} \xrightarrow{\hat{r}} \hat{R})$ as defined in 4.3.7, where $\hat{L} \subseteq \hat{R}$ contains only port nodes and $\hat{R}$ contains additionally one process node of type x. The production morphism $\hat{r}$ is the inclusion from $\hat{L}$ in $\hat{R}$. $\triangle$

As mentioned above, a process grammar for a certain process type consists of a start production and a set of process productions of the corresponding type. From the viewpoint of object-oriented programming, the process grammar of a certain type can be seen as a class. Processes of this type in a distributed system are instances of this class, that have a copy of the "program" (i.e. the process productions and the start production) provided in the

process grammar. A *distributed grammar* with respect to a network type graph NTG has a process grammar and a set of indices for each process type in NTG. The index set represents the name space for instances of the process type that may occur in the distributed system.

Definition 4.3.15 (process grammar, distributed grammar)
Given a network type graph NTG, a *process grammar* of type $x \in PType$ is a pair $\mathcal{GG}(x) = \langle st(x), (p : \hat{r})_{p \in P_x} \rangle$, where $(p : \hat{r})_{p \in P_x}$ is an P_x-indexed set of process productions of type x and $st(x)$ is a start production of type x.
A *distributed grammar* $\mathcal{GG} = (\mathcal{GG}(x), Ind(x))_{x \in PType}$ with respect to NTG consists of a process grammar $\mathcal{GG}(x)$ and an index set $Ind(x)$, called the *name space of* x, for each $x \in PType$. $\triangle$

The distributed grammar of our case study in Chapter 2 is given in Example 2.11. It consists of the process grammars in Table 2.1 and Table 2.2. Since these process grammars are too extensive, I introduce a smaller example used in the remainder of this chapter.

Example 4.9 (distributed grammar)
I provide a distributed grammar $\mathcal{GG} = (\mathcal{GG}(A), Ind(A), \mathcal{GG}(B), Ind(B))$ with respect to the network type graph in Figure 4.10. I use the category $\mathbf{DGr}_{Set}(NTG)$ where locals states are sets with total mappings. Since the network type graph has the two process types A and B, we have to specify a process grammar $\mathcal{GG}(A)$ and a process grammar $\mathcal{GG}(B)$. The start production of the process grammar $\mathcal{GG}(A)$ in Figure 4.13 is empty on its left-hand side and creates a process node of type A containing two elements. Therefore, the process A does not need any ports to start and its initial state has two elements. There are two process productions for process A: The process production p creates a new port that "costs" one of its elements in the local state. The process production p' puts an element from the local state of the process into the port.

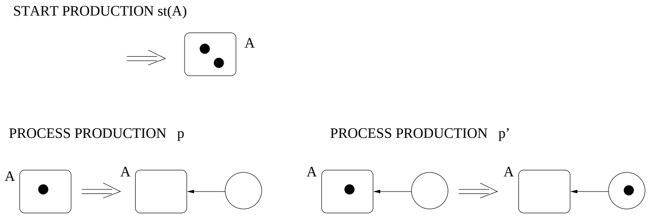


Figure 4.13: *Process grammar for A.*

The start production of the process grammar $\mathcal{GG}(B)$ in Figure 4.14 does not require any ports on its left-hand side as well and creates an empty process node of type B together with a connected port. There is only one process production q for the process B, that creates an element in a port and the process B at the same time.
The index set of the name space for process instances of type A is given by $Ind(A) = \{A1\}$ and for the process B by $Ind(B) = \{B1\}$. For the sake of simplicity, I have chosen singleton index sets. This helps to present the following concepts more legible. However, any bigger set (possibly infinite) is possible as well. $\triangle$

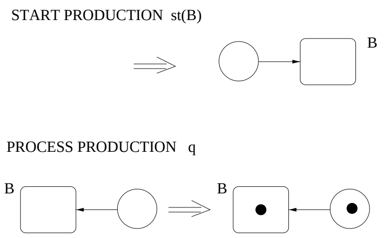


Figure 4.14: *Process grammar for B.*

I intend to use process productions to model the behavior of distributed systems with synchronization, where the synchronized communication between processes takes place over common ports. For that, each of the process productions involved in the synchronized move has to perform the same actions on the ports, i.e. process productions can be synchronized if they coincide in their port actions. This requirement is characterized by the existence of a common *port production*. A port production of a process production is the sub-production got by restricting the process production on the ports.

Definition 4.3.16 (port graph, port production)
Let $\hat{G} = \langle t_G, \mathcal{G} \rangle$ be a distributed graph over G, its *port graph* is the biggest subgraph $Port(\hat{G}) = \langle t_{Port(G)}, Port(\mathcal{G}) \rangle \subseteq \hat{G}$ over $Port(G)$, where the network graph $Port(G)$ contains only port nodes, i.e. for each node x in $Port(G)_V$ holds $t_{NTG}(t_{Port(G)}(x)) = port$ and $Port(\mathcal{G}) = \mathcal{G}|_{Port(G)}$ is the restriction of $\mathcal{G}$ on $Port(G)$.
The *port production* of a conditional distributed production $p : (\hat{r} : \hat{L} \to \hat{R}, A(p))$ is given by $port(p) : (\hat{r}|_{Port(\hat{L})} : Port(\hat{L}) \to Port(\hat{R}), A(port(p)))$, where $A(port(p))$ has for each conditional constraint $cc_{\hat{L}} = (\hat{x} : \hat{L} \to \hat{X}, (\hat{c}_i : \hat{X} \to \hat{C}_i)_{i \in I}) \in A(p)$ a conditional constraint $port(cc_{\hat{L}}) = (\hat{x}|_{Port(\hat{L})} : Port(\hat{L}) \to Port(\hat{X}), (\hat{c}_i|_{Port(\hat{X})} : Port(\hat{X}) \to Port(\hat{C}_i))_{i \in I})$. △

Example 4.10 (port production)
Figure 4.15 shows the port productions of the process productions given in the process grammars $\mathcal{GG}(A)$ and $\mathcal{GG}(B)$ in Example 4.9. The productions p and $st(B)$ as well as the productions p' and q have the same port production, i.e. they are synchronizable. All other productions are not synchronizable.
Examples for port productions of the process productions of the case study in Chapter 2 are given in Example 2.12. △

As mentioned above, process productions are synchronizable if they have the same port production. However, not all process productions that have the same port production have to be synchronized. Therefore, I introduce a *synchronization relation* on the production names of the process instances given in a distributed grammar to indicate which process productions of instances have to be synchronized. The synchronization relation relates only production names, where the port productions are equal. This ensures, that related process productions coincide in the port production and are synchronizable, i.e. they perform the same actions on the ports. The process moves of synchronized productions are intended to take place at the same time. The synchronization relation is defined on the instances

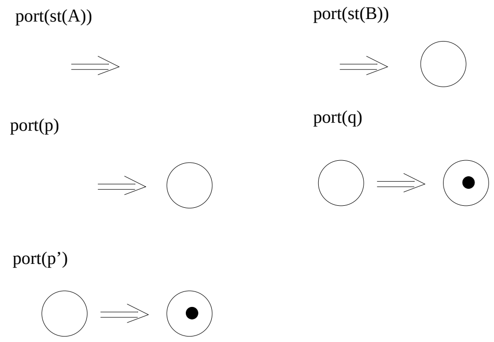


Figure 4.15: *The port productions of $\mathcal{GG}(A)$ and $\mathcal{GG}(B)$*

of the process productions, i.e. the same process production in the process grammar may have a synchronization constraint in one process instance, whereas it can move without synchronization in another instance.

Definition 4.3.17 (synchronization relation)
Given a distributed grammar $\mathcal{GG} = (\mathcal{GG}(x), Ind(x))_{x \in PType}$ with $\mathcal{GG}(x) = \langle st(x), (p : \hat{r})_{p \in P_x} \rangle$ for each $x \in PType$. Let $P_x^i = \{st^i(x)\} \cup (p^i)_{p \in P_x}$ be the set of productions names of process instance $i \in Ind(x)$ for each $x \in PType$ and $P_{\mathcal{GG}} = \bigcup_{x \in PType, i \in Ind(x)} P_x^i$ be the set of the production names of all process instances in the distributed grammar. Then, a *synchronization relation* $\sim_S \subseteq P_{\mathcal{GG}} \times P_{\mathcal{GG}}$ is a relation on $P_{\mathcal{GG}}$, such that $p^i \sim_S p^j$ implies $port(p^i) = port(p^j)$. △

Example 4.11 (synchronization relation)
A synchronization relation for the distributed grammar in Example 4.9 may look as follows: $p \sim_S st(B)$ and $p' \sim_S q$. Since we have only one instance for each process type, I skipped the index for the instance at the production names. The intended meaning of the synchronization constraint for the productions p and $st(B)$ is, that the process instance $A1$ creates the process instance $B1$ if $A1$ creates the port. The intended meaning of the synchronization constraint for the productions p' and q is, that the element of the process instance $A1$ shall be sent synchronously to process instance $B1$. The intended meaning becomes clearer if I introduce the operational semantics for a distributed grammar.
The synchronization relation for the case study in Chapter 2 is given in Example 2.13. △

To sum up, a distributed system is specified by a network type graph that describes the possible process and port types and their connections, a process grammar for each process type given in the network type graph, a name space for the instances of each process type and a synchronization relation for indicating the synchronization constraints on process instance moves.

Definition 4.3.18 (distributed system specification)
A *distributed system specification* $DSP = (NTG, \mathcal{GG}, \sim_S)$ consists of

1. a network type graph NTG as defined in Definition 4.3.1,

2. a distributed grammar $\mathcal{GG} = (\mathcal{GG}(x), Ind(x))_{x \in PType}$ with respect to NTG as defined in Definition 4.3.15 and
3. a synchronization relation as defined in Definition 4.3.17.

$\triangle$

4.3.4 A Compositional Operational Semantics based on Transformation Systems

This section provides the compositional operational semantics of a distributed system with respect to a distributed system specification. The operational semantics is composed by the operational semantics of the process instances. The basis for the operational semantics and the composition are *transformation systems*. Transformation systems are introduced by Große-Rhode in [GR98,GR99] as common semantical framework in which specifications written in different languages can be interpreted. I show that distributed graph transformation can be interpreted in this framework as well. This relation enables us to make use of the composition of transformation systems as defined in [GR98,GR99]. In the following, I give an introduction in transformation systems and their composition and show afterwards how distributed graph transformation fits into this framework.

4.3.4.1 Transformation Systems and their Composition

A transformation system is a two layered structure, given by a *control flow graph* that models the temporal ordering of actions, and a data level, given by data states and data transformations that are associated to the states and transitions of the control flow graph respectively. Via this separation the control flow graph defines how atomic data transformations are put together to processes. The paths in the control flow graph model the sequential execution of actions, branching indicates nondeterministic choice.

The control flow graph is an edge-labeled graph with one distinguished node, called *initial state*. The nodes of the control flow graph are called *control states* in the following. Morphisms between control flow graphs are given by total graph morphisms that map initial states on initial states and are compatible with the labels.

Definition 4.3.19 (control flow graph)
A *control flow graph* consists of a tuple $CG = \langle G, L, l, init \rangle$ where G is a graph, L is a set of labels, $l : G_E \to L$ is a labeling mapping for edges and $init \in G_V$ is the *initial state*.
A *control flow graph morphism* between control flow graphs $CG = \langle G, L, l, init \rangle$ and $CG' = \langle G', L', l', init' \rangle$ is a pair $h^{CG} = \langle h^G, h^L \rangle : CG \to CG'$ where $h^G : G \to G'$ is a total graph morphism such that $h^G(init) = init'$ and $h^L : L \to L'$ is a total mapping such that $h^L(l(e)) = l'(h^G(e))$ for each edge $e \in G_E$. $\triangle$

The data states of a transformation system can be chosen from an arbitrary cocomplete category $\mathbf{C}$, e.g. partial algebras to a given algebraic signature. The data transformation is given by a $\mathbf{C}$-morphism. The association of data states and data transformations to the control states and edges of the control flow graph is defined via a total graph morphism from the control flow graph in the underlying graph of $\mathbf{C}$ that serves as the domain of all possible data states and data transformations.

Definition 4.3.20 (transformation system)
Let $\mathbf{C}$ be a category, a *transformation system* in $\mathbf{C}$ is a pair $A = \langle CG, a \rangle$, where $CG = \langle G, L, l, init \rangle$ is a control flow graph and $a : G \to G_{\mathbf{C}}$ is a total graph morphism from G in the underlying graph $G_{\mathbf{C}}$ of the category $\mathbf{C}$.
A *transformation morphism* between transformation systems $A = \langle CG, a \rangle$ and $A' = \langle CG', a' \rangle$ is a pair $h = \langle h^{CG}, \alpha^h \rangle : A \to A'$, where $h^{CG} : CG' \to CG$ is a control flow graph morphism and $\alpha^h : V_{h^G}(a) \to a'$ is a family of $\mathbf{C}$-morphisms $\{\alpha^h(i) : a(h^G(i)) \to a'(i) | i \in G'_V\}$ with $V_{h^G}(a) = a \circ h^G$. $\triangle$

Please notice, that the morphism $\alpha^h : V_{h^G}(a) \to a'$ is a morphism of the model category $\mathbf{Mod}^T_{\mathbf{C}}(G')$ and the directions of the control flow graph morphism and the morphism on the data level are opposite. This fact plays an important role for the composition of transformation systems by limits and colimits: Consider the case of two transformation systems A_1 and A_2 that shall be composed. The synchronization constraints for the composition of these two transformation systems are given by an auxiliary transformation system A_0, the actual synchronization is then expressed by transformation morphisms $h_1 : A_0 \to A_1$ and $h_2 : A_0 \to A_2$. The control flow graph morphisms $h_1^{CG} : CG_{A_1} \to CG_{A_0}$ and $h_2^{CG} : CG_{A_2} \to CG_{A_0}$ indicate the transitions that shall be performed synchronously. These are exactly those transition in CG_{A_1} and CG_{A_2}, that are mapped onto the same transition in the control flow graph CG_{A_0}. The limit of h_1^{CG} and h_2^{CG} forms the control flow graph, where all synchronization constraints are satisfied. The data states and data transformations of the composed control flow graph are given by the colimit of the morphisms α^{h_1} and α^{h_2}.

Definition 4.3.21 (composition of transformation systems)
Given transformation systems $A_i = \langle CG_{A_i}, a_i \rangle$ in $\mathbf{C}$ with $CG_{A_i} = \langle G_i, L_i, l_i, init_i \rangle$ for $i = 0, 1, ..., n$ and transformation morphisms $h_j = \langle h^{CG_j}, \alpha^{h_j} \rangle : A_0 \to A_j$ for $j = 1, ..., n$. The *composition* of the A_j via h_j for $j = 1, ..., n$ is given by the transformation system $A = \langle CG, a \rangle$ and the transformation morphisms $h'_j = \langle h'^{CG_j}, \alpha^{h'_j} \rangle : A_j \to A$, such that $CG = \langle G, L, l, init \rangle$ with $h'^{CG_j} : CG \to CG_{A_j}$ is the limit of the h^{CG_j}.

$$\begin{tikzcd}
 & CG_{A_0} & \\
CG_{A_1} \arrow[ur, "h^{CG_1}"] & \ldots & CG_{A_n} \arrow[ul, "h^{CG_n}"'] \\
 & CG \arrow[ul, "h'^{CG_1}"] \arrow[ur, "h'^{CG_n}"'] &
\end{tikzcd}$$

The graph morphism $a : G \to G_{\mathbf{C}}$ is defined as the colimit of the morphisms $\alpha^{h_j} : V_{h'^G_j \circ h^G_j}(a_0) \to V_{h'^G_j}(a_j)$ in the category $\mathbf{Mod}^T_{\mathbf{C}}(G)$.

$$\begin{tikzcd}
 & V_{h'^G_i \circ h^G_i}(a_0) \arrow[dl, "\alpha^{h_1}"'] \arrow[dr, "\alpha^{h_n}"] & \\
V_{h'^G_1}(a_1) \arrow[dr, "\alpha^{h'_1}"'] & \ldots & V_{h'^G_n}(a_n) \arrow[dl, "\alpha^{h'_n}"] \\
 & a &
\end{tikzcd}$$

$\triangle$

The composition of transformation systems via transformation morphisms can be categorically characterized as the colimit in the category of transformation systems and transformation morphisms. More details with respect to this topic can be found in [GR98].

I show now that distributed graph transformation fits into the framework of transformation systems. In the case of distributed graph transformation, the category for the domain of all possible data states and data transformations is given by the category $\mathbf{DGr_C}(NTG)$ of distributed graphs and distributed morphisms with local $\mathbf{C}$-states. The control flow graph of a transformation system models the temporal ordering of actions and describes how transformations of distributed graphs are put together. Depending on the structure of the control flow graph and the assignment of data states and data transformations to the nodes resp. edges of the control flow graph, a different behavior is modeled. I show now the construction of a transformation system that represents the behavior of a given set of derivation sequences D generated by a grammar $\mathcal{GG}$. The control flow graph of this transformation system in $\mathbf{DGr_C}(NTG)$ has as control states all distributed graphs occurring in the derivation sequences and there is an edge from a control state $\hat{G}$ to a control state $\hat{H}$ labeled by a pair $(p, \hat{m})$ if there is a direct derivation from $\hat{G}$ to $\hat{H}$ with the distributed production p at the match $\hat{m}$ occurring in a derivation sequence. The data state for a control state $\hat{G}$ is the distributed graph $\hat{G}$ itself and the data transformation for an edge from $\hat{G}$ to $\hat{H}$ with the label $(p, \hat{m})$ is the $\mathbf{DGr_C}(NTG)$-morphism from $\hat{G}$ to $\hat{H}$ given by the direct derivation from $\hat{G}$ to $\hat{H}$ with p at $\hat{m}$. The set of labels for the control flow graph is therefore given by the set of production names and the matches of the derivation sequences. The initial state is given by the initial graph of the grammar.

Definition 4.3.22 (transformation system for a set of derivation sequences)
Let $\mathcal{GG} = \langle \hat{G}_0, (p : \hat{r})_{p \in P} \rangle$ be a grammar for distributed graphs and $D \subseteq Der(\mathcal{GG})$ be a set of derivation sequences. The transformation system in $\mathbf{DGr_C}(NTG)$ for D is given by $A(D) = \langle CG_A, a \rangle$ with $CG_A = \langle G, L, l, \hat{G}_0 \rangle$, where

- L contains all pairs $(p, \hat{m})$ such that $p \in P$ and there is a derivation sequence $\rho \in D$ and an $i \in \mathbb{N}$ such that $\rho(i) = (\hat{G} \overset{p,\hat{m}}{\Rightarrow} \hat{H})$,
- $G_V = \{\hat{G} | \rho = (\hat{G}_0 \Rightarrow ... \Rightarrow \hat{G} \Rightarrow ...) \in D\}$ is the set of all distributed graphs that occur in a derivation of D and
- there is an edge $e^{\rho(i)} : \hat{G} \to \hat{H}$ with $l(e^{\rho(i)}) = (p, \hat{m})$ in G_E if and only if there is a derivation sequence $\rho \in D$ and an $i \in \mathbb{N}$ such that $\rho(i) = (\hat{G} \overset{p,\hat{m}}{\Rightarrow} \hat{H})$.

The graph morphism $a : G \to G_{\mathbf{DGr_C}(NTG)}$ is defined by $a(\hat{G}) = \hat{G}$ for each $\hat{G} \in G_V$ and by $a(e^{\rho(i)}) = \hat{r}^*$ if the direct derivation $\rho(i) = (\hat{G} \overset{p,\hat{m}}{\Longrightarrow} \hat{H})$ from $\hat{G}$ to $\hat{H}$ with $p : (\hat{L} \overset{\hat{r}}{\to} \hat{R}, A(p))$ is given by the following pushout diagram in $\mathbf{DGr_C}(NTG)$.

$$\begin{array}{ccc} \hat{L} & \xrightarrow{\hat{r}} & \hat{R} \\ {\scriptstyle \hat{m}}\downarrow & & \downarrow{\scriptstyle \hat{m}^*} \\ \hat{G} & \xrightarrow[\hat{r}^*]{} & \hat{H} \end{array}$$

△

4.3.4.2 Compositional Operational Semantics of Distributed Systems

This part uses transformation systems for the description of the compositional operational semantics of a distributed system specification. For single process instances, I investigate two different kinds of operational semantics that depend on the actual environment. The *isolated process semantics* specifies the behavior of an isolated process, where no effects of the environment occur. Unlike the *open process semantics*, that assumes a complete environment in the sense that all possible effects on ports are considered that can occur due to actions of other process instances in the distributed system. Both, the isolated and the open process semantics are given by transformation systems. By composition of the open process semantics for the process instances given in the distributed grammar I get the operational semantics of the distributed system.
I illustrate the concepts mainly by the distributed grammar in Example 4.9 and the synchronization relation in Example 4.11. The parts of the compositional operational semantics of the case study shown in Subsection 2.5.2.2 serve as additional examples.

The isolated process semantic for a process instance considers all derivation sequences that use process productions of this instance starting with the start production. To ensure that we describe only the semantics of one process instance, the start production is not allowed to be used twice.

Definition 4.3.23 (isolated process semantics)
Given a distributed grammar $\mathcal{GG} = (\mathcal{GG}(x), Ind(x))_{x \in PType}$ with $\mathcal{GG}(x) = \langle st(x), (p : \hat{r})_{p \in P_x} \rangle$ and $st(x) : (\hat{L}_0 \to \hat{R}_0)$ for each $x \in PType$. Then, the *isolated process semantics* for the process instance $i \in Ind(x)$ is given by the transformation system $A(D)$ for the set $D = \{\rho = (\hat{L}_0 \overset{st(x)}{\Rightarrow} \hat{R}_0 \overset{p_1}{\Rightarrow} \hat{G}_2 \overset{p_2}{\Rightarrow} ...) | p_j \in P_x^i, j \in \mathbb{N}\}$. △

Example 4.12 (isolated process semantics)
I consider again the distributed grammar given in the Example 4.9, that contains two process instances $A1$ and $B1$ of type A and B, respectively. The isolated process semantics of the process instance $A1$ is shown in Figure 4.16 on the left-hand side. The figure shows the data level of the transformation system, that are the derivation sequences. In the first step, we apply the start production $st(A)$ on the initial state of the isolated process semantics for $A1$ (that is the empty distributed graph, since the left-hand side of the start production is empty). This step creates the instance $A1$. We are not able to create more instances, since the start production cannot be applied again. Then, only the production p is applicable, that creates a new port and deletes one element of the local state of $A1$. Now, the productions p and p' are applicable. After application of the production p resp. p' , we cannot apply any production, since there does not exist an element in the local state of $A1$.
In the transformation system for $B1$ on the right-hand side of the figure, the start production $st(B)$ is firstly applied to create the instance $B1$. Hereafter, the process production q can be applied infinitely often to create new elements in the local state of the process $B1$ and the port.

The open process semantics of a process instance considers not only the own process productions, but additionally the port productions of other process instances. The port productions of a process instance describe the actions the process instance may perform on the ports of

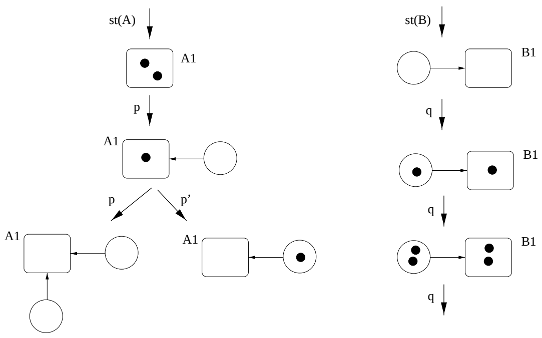


Figure 4.16: *The isolated process semantics of instance $A1$ (left-hand side) and instance $B1$ (right-hand side).*

the distributed system. Since ports are visible for all process instances, the port productions form the part of the process actions, that are visible for the environment. If we consider the port productions of all process instances given in the distributed grammar, we get all possible actions that can occur on ports. I collect all port productions of a distributed grammar in a set, called the *port behavior* of the distributed grammar. The port behavior of a distributed grammar can be seen as the maximal environment in the sense that each action on a port in a current environment of a process instance can be specified by a port production in the port behavior. There does not exist actions on ports, that cannot be specified by a port production of the port behavior.
I identify port productions of process productions that are related by the synchronization relation, since related process productions have the same port production and move at the same time. Therefore, their effect on the environment is completely described by the identified port production.

Definition 4.3.24 (port behavior for a distributed system specification)
Given a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ with $\sim_S \subseteq P_{\mathcal{GG}} \times P_{\mathcal{GG}}$ as defined in Definition 4.3.17. Let $P_{\mathcal{GG}}/_{\equiv_S}$ be the set $P_{\mathcal{GG}}$ factorized by the equivalence relation $\equiv_S$ induced by $\sim_S$, the *port behavior for DSP* is the $P_{\mathcal{GG}}/_{\equiv_S}$-indexed set of productions $([p] : port(\hat{L}) \xrightarrow{\hat{r}'} port(\hat{R}))_{[p]\in P_{\mathcal{GG}}/_{\equiv_S}}$, where $\hat{r}' = \hat{r}|_{Port(\hat{L})}$ for $p : (\hat{r} : \hat{L} \to \hat{R})$. △

Example 4.13 (port behavior)
The port productions of the distributed grammar in Example 4.9 are shown in Figure 4.15. Due to the synchronization relation in Example 4.11, the port behavior consists of the three port productions in Figure 4.17. The port productions $[p]$ and $[st(B)]$ resp. $[p']$ and $[q]$ are identified, since $p \sim_S st(B)$ resp. $p' \sim_S q$. △

For the open process semantics of a process instance, I consider derivation sequences starting at the empty graph using the process productions of the process instance and port productions of the port behavior. I do not consider the port productions of the process productions of the instance, since the complete process production can be applied to achieve the effects

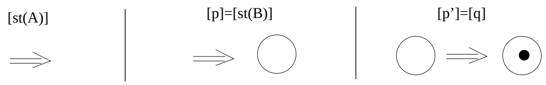


Figure 4.17: *The port productions of the port behavior.*

on the ports. In addition, I do not consider the port productions of process productions of other process instances, that have to be synchronized with a process production of the instance, since synchronized productions have the same port production and have to be applied at the same time. The remaining port productions of the port behavior specify the effects on ports, that cannot be modeled by the process productions of the instance. I call this set of port productions the *remote port behavior* of the instance.
I start the derivation sequences at the empty graph, since a process instance has not necessarily;y to be created in the first step, but can be created in every possible port environment, where the necessary ports for the process start production occur. This fact is a crucial point for describing the compositional operational semantics for a distributed system, where the topology can change dynamically. Just as in the case of the isolated process semantics, the start production can be applied at most once. Moreover, each process production, that is synchronized with a start production is allowed to be applied at most once, too. This is motivated by the fact, that synchronized process productions have to move at the same time. If the start production can be applied only once, also each process production synchronized with the start production can move only once.

Definition 4.3.25 (open process semantics)
Let $DSP = (NTG, \mathcal{GG}, \sim_S)$ with $\mathcal{GG} = (\mathcal{GG}(x), Ind(x))_{x\in PType}$ be a distributed system specification and $([p] : \hat{r}')_{[p]\in P_{\mathcal{GG}}/_{\equiv_S}}$ be the port behavior for DSP. Then, the *remote port behavior* for the instance $i \in Ind(x)$ is given by the set $RPB(i) \subseteq P_{\mathcal{GG}}/_{\equiv_S}$ such that $[p] \in RPB(i)$ if and only if there does not exist a process production $p^i \in P^i_x$ of the instance i with $p^i \in [p]$.
The *open process semantics* for the instance $i \in Ind(x)$ is given by the transformation system $OS(i) = A(D)$ for the set of derivation sequences $D = \{\rho = (\emptyset \overset{p_1}{\Rightarrow} \hat{G}_1 \overset{p_2}{\Rightarrow} \hat{G}_2 ...) | p_j \in P^i_x \cup RPB(i), j \in \mathbb{N}\}$, such that for all $\rho \in D$, $\rho(k) = (\hat{G}_k \overset{st}{\Rightarrow} \hat{H}_k)$ and $\rho(j) = (\hat{G}_j \overset{st}{\Rightarrow} \hat{H}_j)$ implies $k = j$ if $st = st^i(x)$ is the start production of the instance i, $st = p \in P^i_x$ is a process production of i that has to be synchronized with a start production of another instance $j \in Ind(y), y \in PType$, i.e. $p \equiv_S st^j(y)$, or $st = [p] \in RPB(i)$ is a port production of a process production p that has to be synchronized with a start production, i.e. $p \equiv_S st^j(y)$. △

Example 4.14 (open process semantics)
We consider again the distributed grammar in Example 4.9; its port behavior is given in Example 4.13. None of the port productions in the port behavior is in the remote port behavior of the instance $A1$, since each port production in the port behavior is a port production of a process production of $A1$. Therefore, the open process semantics of $A1$ uses only the own process productions $st(A)$, p and p'. Just as in the isolated process semantics, the start production $st(A)$ can be applied at most once. Moreover, since the process production p is synchronized with the start production $st(B)$, also p can be applied

at most once. Therefore, the open process semantics of the instance $A1$ on the left-hand side of Figure 4.18 is restricted with respect to the isolated process semantics in Figure 4.16. The

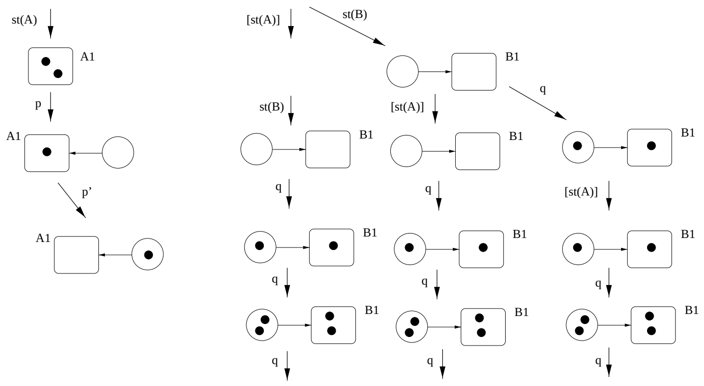


Figure 4.18: *The open process semantics for* $A1$ *(left-hand side) and* $B1$ *(right-hand side).*

remote port behavior of the process instance $B1$ contains the port production $[st(A1)]$, i.e. the open process semantics of $B1$, shown on the right-hand side of Figure 4.18, is generated by the productions $st(B), q$ and $[st(A)]$. The productions $st(B)$ and $[st(A)]$ can be applied at most once. Therefore, the open process semantics differs from the isolated process semantics of $B1$ in Figure 4.16 in such a way, that in an arbitrary step of the sequence in Figure 4.16 the port production $[st(A)]$ can be applied once. △

The Example 2.16 of Chapter 2 shows parts of the open process semantics for revision archive instances of our case study.

The operational semantics of a distributed system given by a distributed system specification is composed by the open process semantics of the process instances in the distributed grammar. The composition takes place over an auxiliary transformation system as explained in Definition 4.3.21. This auxiliary transformation system is generated by the port behavior of the distributed system specification and represents the behavior on the ports of the distributed system. Each open process semantics restricted to the ports is then represented into the port transformation system.

Definition 4.3.26 (port transformation system)
Given a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$, the *port transformation system* for DSP is the transformation system $Port(DSP) = A(D)$ for the set of derivations $D = \{\rho = (\emptyset \overset{[p_1]}{\Rightarrow} \hat{G}_1 \overset{[p_2]}{\Rightarrow} \hat{G}_2 \overset{[p_3]}{\Rightarrow} ...) | [p_i] \in P_{\mathcal{GG}}/_{\equiv_S}, i \in \mathbb{N}\}$, where $([p] : \hat{r}')_{[p] \in P_{\mathcal{GG}}/_{\equiv_S}}$ is the port behavior for DSP. △

Example 4.15 (port transformation system)
We use the port productions $[st(A)], [st(B)]$ and $[q]$ of the port behavior in Example 4.13 to construct the port transformation system for the distributed grammar in Example 4.9

and the synchronization relation in Example 4.11. The port transformation system (a part of) is shown at the top of the Figure 4.19. In contrast to the open process semantics,

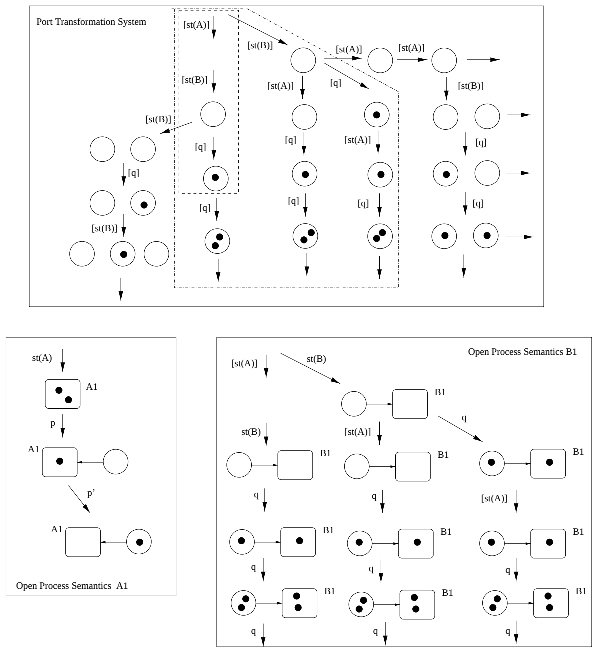


Figure 4.19: *The port transformation system.*

the port productions of start productions are applied as often as possible. Since the port transformation system is used for the composition, the additional information is allowed here. The port transformation system represents all the effects on ports, that are possible with the process productions of $A1$ and $B1$. In particular, the port behavior of $A1$ and $B1$ is represented in the port transformation system (indicated by the dashed boxes). △

As mentioned above, the composition of the open process semantics for each instance in the distributed grammar takes place over the port transformation system. The composition has to follow the synchronization relation in the following sense: If an edge $e : \hat{G} \overset{p,\hat{m}}{\Rightarrow} \hat{H}$ of the control flow graph of the open process semantics is mapped on an edge $e' : \hat{G}' \overset{[p'],\hat{m}'}{\Rightarrow} \hat{H}'$ in the port control flow graph then $p \in [p']$, in particular $[p']$ is the port production of p. Moreover, the ports and its local states in the distributed graphs $\hat{G}$ and $\hat{G}'$ resp. $\hat{H}$ and $\hat{H}'$ have to be the same, i.e. $port(\hat{G}) = \hat{G}'$ and $port(\hat{H}) = \hat{H}'$. In addition, since $[p']$ is the port production of p, we require that also the restricted data transformation from $port(\hat{G})$ to $port(\hat{H})$ in the open process semantics is the same as the data transformation from G' to H' in the port transformation system performed on the same elements, i.e. $\hat{m}' = \hat{m}_{port(\hat{L})}$ with $p : (\hat{r} : \hat{L} \to \hat{R})$. All these conditions ensure that the transition e' and its data transformation can be embedded into the open process semantics in the sense that only the process instance occurs additionally, but the ports and their transformation are exactly the same as in the open process semantics.

Definition 4.3.27 (distributed semantics)
Let $DSP = (NTG, \mathcal{GG}, \sim_S)$ be a distributed system specification and $OS(i_x) = \langle CG_{i_x}, a_{i_x} \rangle$ be the open process semantics of the instance $i \in Ind(x)$ for $x \in PType$. Then, we define the transformation morphism $h_{i_x} = \langle h^{CG_{i_x}}, \alpha^{h_{i_x}} \rangle : Port(DSP) \to OS(i_x)$ as follows:

- For the control flow graph morphism $h^{CG_{i_x}} = \langle h^{G_{i_x}}, h^{L_{i_x}} \rangle : \langle G_{i_x}, L_{i_x}, l_{i_x}, init_{i_x} \rangle \to \langle G, L, l, init \rangle$ is
  - $h^{L_{i_x}} : L_{i_x} \to L$ defined by

    $$h^{L_{i_x}}(p, \hat{m}) = \begin{cases} ([p], \hat{m}|_{port(\hat{L})}) & \text{, if } p : (\hat{L} \overset{\hat{r}}{\to} \hat{R}, A(p)) \in P^i_x \\ (p, \hat{m}) & \text{, if } p \in RPB(i) \end{cases}$$

    for all $(p, \hat{m}) \in L_i$,
  - $h^{G_{i_x}}_V(\hat{G}) = port(\hat{G})$ for each $\hat{G} \in G_{i_x}$ and $h^{G_{i_x}}_E(e) : port(\hat{G}) \overset{h^{L_{i_x}}(p,\hat{m})}{\to} port(\hat{H})$ for each $e : \hat{G} \overset{(p,\hat{m})}{\to} \hat{H}$.
- The morphism $\alpha^{h_{i_x}} : V_{h^{G_{i_x}}}(a) \to a_i$ is defined for all $h^{G_{i_x}}(\hat{G}) \in G_V, \hat{G} \in G_{i_x}$ by the inclusion $\alpha^{h_{i_x}}(h^{G_{i_x}}(\hat{G})) : port(\hat{G}) \hookrightarrow \hat{G}$.

The *distributed semantics* $DS(DSP)$ for DSP is given by the composition of the open process semantics $(OS(i_x))_{i \in Ind(x), x \in PType}$ via the transformation morphisms $(h_{i_x})_{i \in Ind(x), x \in PType}$. △

I have to show that the transformation morphisms are well defined, what is based on the well definition of the control flow graph morphisms. Its well definition follows from the production name and the match given for the label of edges in the control flow graph.

Proposition 4.3.28
Under the assumptions and notations of Definition 4.3.27, the transformation morphisms $(h_{i_x} = \langle h^{CG_{i_x}}, \alpha^{h_{i_x}} \rangle : Port(DSP) \to OS(i_x))_{i \in Ind(x), x \in PType}$ *are well-defined.* △

Proof
By definition of the transformation morphism $h_{i_x} = \langle h^{CG_{i_x}}, \alpha^{h_{i_x}} \rangle : Port(DSP) \to OS(i_x)$ for each $i \in Ind(x), x \in PType$, h_{i_x} is well-defined if the control flow graph morphism $h^{CG_{i_x}} = \langle h^{G_{i_x}}, h^{L_{i_x}} \rangle$ is well-defined. It can be easily checked that $h^{L_{i_x}}$ is well-defined. The mapping $h^{G_{i_x}}$ for nodes is well defined, since there do not exist two nodes for the same distributed graph in the control flow graph of the port transformation system by definition of a transformation system for a set of derivation sequences (cf. Def. 4.3.22). The mapping $h^{G_{i_x}}$ for edges is well defined due to the requirement, that the port transformation in the open process semantics is the same as in the port transformation system and the transformed elements in the ports are the same as well. That the same elements are transformed is ensured by the requirement that the match used in the port transformation system is the same as the restriction of the match in the open process semantics. □

Example 4.16 (distributed semantics)
We construct the distributed semantics for the distributed system specification containing the distributed grammar in Example 4.9 and the synchronization relation in Example 4.11. The distributed grammar contains the two instances $A1$ and $B1$, their open process semantics were introduced in Example 4.14. The composition of the open process semantics for $A1$ and $B1$ takes place over the port transformation system constructed in Figure 4.19.

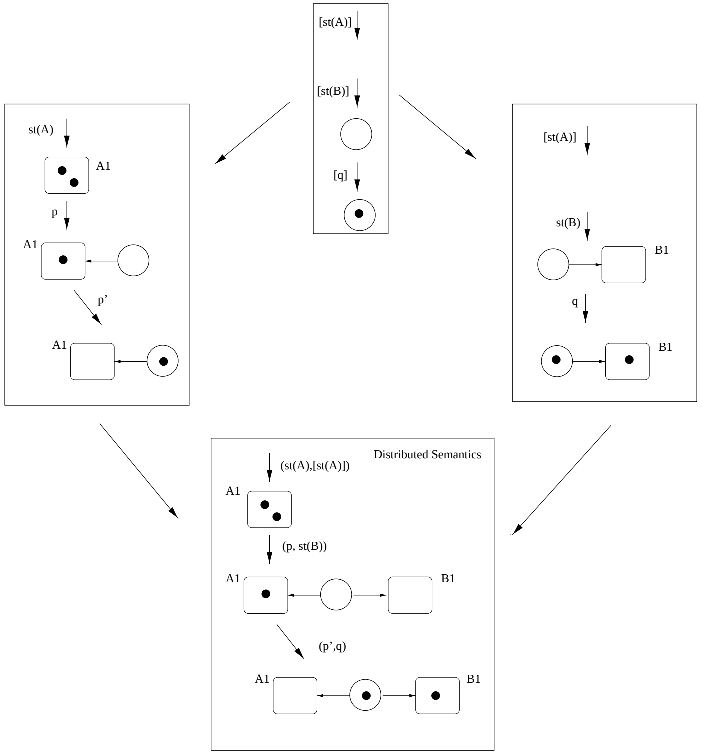


Figure 4.20: *The construction of the distributed semantics.*

If we restrict each distributed graph and each transformation step in the open process semantics to the ports, i.e. we delete all process instance nodes, we get the port behavior of the process instance. By construction of the port transformation system, we can find this port behavior of the process instance in the port transformation system. In Figure 4.19, this is indicated by the dashed boxes. The intersection of the port behavior of the instance $A1$ and $B1$ forms the steps of the distributed semantics. In the Figure 4.19, the intersection is the complete port behavior of the instance $A1$. Therefore, all steps of the instance $A1$ occur also in the distributed semantics, but several steps of the instance $B1$ are removed for the distributed semantics. The removed steps of $B1$ are those, that represent applications of productions, that have to be synchronized, but where a corresponding step does not exist in $A1$. As mentioned, the steps of the distributed semantics are given by the intersection of the port steps in the port transformation system. The local data of the steps in the distributed system are constructed by the pushout of the corresponding data states in the open process semantics. The ports in the port transformation system serve as the gluing objects. Figure 4.20 shows the construction of the local data steps. I have shown only those parts of the port transformation system and the open process semantics of $A1$ and $A2$ that are necessary for the construction of the local data states of the three steps of the distributed semantics.

The distributed semantic in the example consists of three steps. At first, the process instance $A1$ is created. Then, the instance $B1$ is created and is connected by a port to the instance $A1$. The synchronization of the process production p and $st(B)$ has ensured, that the process instance $A1$ generates the process $B1$ by creating the port. In the last step, an element of the local state of $A1$ is sent synchronously over the port to the process instance $B1$, what is ensured by the synchronization constraint for production p' and q. △

The Example 2.17 shows the composition of the open process semantics for two revision archive instances of our case study in Chapter 2. As the example in Figure 4.20 shows, ports remain unchanged and process instances are added by the composition of the open process semantics.

Proposition 4.3.29
Let $DS(DSP)$ with $(h'_{i_x} : OS(i_x) \to DS(DSP))_{i \in Ind(x), x \in PType}$ be the distributed semantics with respect to the distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ composed via the transformation morphisms $(h_{i_x} : Port(DSP) \to OS(i_x))_{i \in Ind(x), x \in PType}$. Then, for each node $\hat{G}$ in the control flow graph of $DS(DSP)$, the distributed morphism $\alpha^{h'_{i_x}}({h'}^G_{i_x}(\hat{G})) = \langle f'_{i_x}, \alpha^{f'_{i_x}} \rangle : {h'}^G_{i_x}(\hat{G}) \to \hat{G}$ has the following properties:

1. *Let ${h'}^G_{i_x}(\hat{G}) = \hat{G}'$, then $f'_{i_x} : G' \to G$ is injective and $f'_{i_x}|_{port(G')} : port(G') \to port(G)$ is bijective and*
2. *for each $v \in G'_V$, $\alpha^{f'_{i_x}}(v)$ is an isomorphism.*

△

Proof
By construction, for each node $\hat{G}$ in the control flow graph of $DS(DSP)$, the diagram below is a colimit in the category $\mathbf{DGr_C}(NTG)$.

$$\begin{tikzcd}
 & h^G_{i_x}({h'}^G_{i_x}(\hat{G})) \arrow[dl, bend right, "\alpha^{h_1}(h^G_1({h'}^G_1(\hat{G})))"'] \arrow[dr, bend left, "\alpha^{h_n}(h^G_n({h'}^G_n(\hat{G})))"] & \\
{h'}^G_1(\hat{G}) \arrow[dr, "\alpha^{h'_1}({h'}^G_1(\hat{G}))"'] & \cdots & {h'}^G_n(\hat{G}) \arrow[dl, "\alpha^{h'_n}({h'}^G_n(\hat{G}))"] \\
 & \hat{G} &
\end{tikzcd}$$

The morphism $\alpha^{h_{i_x}}(h^G_{i_x}({h'}^G_{i_x}(\hat{G}))) = \langle f_{i_x}, \alpha^{f_{i_x}} \rangle : h^G_{i_x}({h'}^G_{i_x}(\hat{G})) \hookrightarrow {h'}^G_{i_x}(\hat{G})$ is an inclusion by definition of the distributed semantics $DS(DSP)$. Therefore, since the diagram above is

a colimit diagram in $\mathbf{DGr_C}(NTG)$ and f_{i_x} restricted to the ports is bijective, the graph morphism f'_{i_x} is injective and it is bijective restricted to the ports.

Let $v \in G'_V$ be a port node in G'_V, then there is a $w \in G''_V$ with $f_{i_x}(w) = v$. By definition of the composition $\alpha^{f_{i_x}}(w)$ is the identity, such that $\alpha^{f'_{i_x}}(v)$ is an isomorphism by pushout construction in $\mathbf{DGr_C}(NTG)$. If $v \notin f_{i_x}(G''_V)$ then v is a process node and $\alpha^{f'_{i_x}}(v)$ is an isomorphism due to the fact that the colimit does not freely generates new elements in the processes, since the ports remain unchanged. □

4.3.5 Amalgamation and Distribution

In the previous section I considered process productions that are applied locally to one process instance to generate the open process semantics for this process instance. The open process semantic contains the local states and the transitions for one process instance. The composition of the open process semantics to the distributed semantics is controlled by a synchronization relation, that determines which process instance steps have to take place synchronized. For this compositional approach no global productions and no global states are necessary.

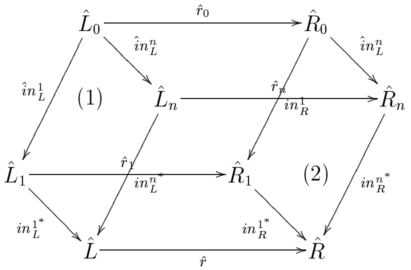
Another approach in graph transformation to model the synchronization of productions are *amalgamated productions* [CMR+97,Tae97,TFKV99]. Here, distributed systems with global states and global productions are considered. In this setting, the synchronized application of two productions is described by the global application of the amalgamated production of the productions that shall be synchronized.

In this section, I relate my approach to the approach with amalgamated productions. I show that the compositional distributed semantics can be simulated by the global distributed semantics given by the amalgamated productions. Vice versa, the global distributed semantics given by the amalgamated productions can be simulated by the compositional distributed semantics, if the matches for the global amalgamated productions can be split into local matches for the instances. This is similar to the *distribution condition* in [CMR+97].

The amalgamation of distributed productions takes place over a common *sub-production*. A distributed production $p_0 : (\hat{L}_0 \stackrel{\hat{r}_0}{\to} \hat{R}_0)$ with total distributed morphisms $\hat{in}_L : \hat{L}_0 \to \hat{L}$ and $\hat{in}_R : \hat{R}_0 \to \hat{R}$ is a sub-production of a distributed production $p : (\hat{L} \stackrel{\hat{r}}{\to} \hat{R})$ if $\hat{in}_R \circ \hat{r}_0 = \hat{r} \circ \hat{in}_L$. I introduce here only the amalgamation of distributed productions without application conditions. The amalgamation of conditional productions can be found in [Hec95], the generalization to conditional distributed productions in [TFKV99].

Definition 4.3.30 (amalgamated production, amalgamated derivation)
Let $p_0 : (\hat{L}_0 \stackrel{\hat{r}_0}{\to} \hat{R}_0)$ with $\hat{in}^i_L : \hat{L}_0 \to \hat{L}_i$ and $\hat{in}^i_R : \hat{R}_0 \to \hat{R}_i$ be a subproduction of the distributed productions $p_i : (\hat{L}_i \stackrel{\hat{r}_i}{\to} \hat{R}_i)$ for $i = 1...n$. The amalgamated production $p_1 + ... + p_n : (\hat{L} \stackrel{\hat{r}}{\to} \hat{R})$ for p_i over p_0 is the distributed production, where the diagrams (1) and (2) are the colimits of $(\hat{in}^i_L)_{i=1...n}$ resp. $(\hat{in}^i_R)_{i=1...n}$ in the category $\mathbf{DGr_C}(NTG)$ and $\hat{r}$

is induced by the universal property of (1).



A direct derivation $\hat{G} \overset{p,\hat{m}}{\Rightarrow} \hat{H}$ using the amalgamated production $p = p_1 + ... + p_n$ is called *amalgamated direct derivation.* △

Example 4.17 (amalgamated productions)
Port productions are subproductions of process productions and process productions are synchronizable, if they have the same port production, i.e. they have a common subproduction. Therefore, we can construct the amalgamated production for the process productions that are related by the synchronization relation of a distributed system specification. The synchronization relation in Example 4.11 relates the process productions p and $st(B)$ resp. p' and q. The Figure 4.21 shows the corresponding amalgamated productions. The amalgamated

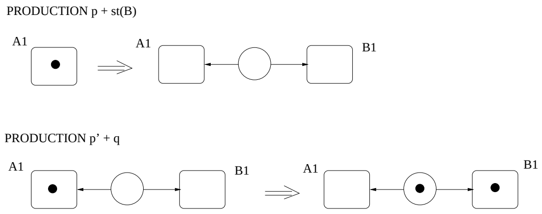


Figure 4.21: *The amalgamated productions for the distributed system specification.*

productions are global distributed productions, since they refer to two process instances. The amalgamated production at the top of the figure shows the synchronization of the process production p and the start production $st(B)$. The creation of the new process instance $B1$ is done by the process instance $A1$. The amalgamated production at the bottom shows, that the deletion of the element of the local state of $A1$ induces a synchronized creation of an element in the local state of $B1$ and the port.

Example 2.14 shows amalgamated productions for the case study in Chapter 2. △

Each distributed system specification induces a set of amalgamated productions by constructing the amalgamated production for the process productions that are related by the synchronization relation. The *global distributed semantics* is given by the transformation system for all derivation sequences starting at the empty distributed graph using the amalgamated productions induced by the distributed system specification.

Definition 4.3.31 (global distributed semantics)
Given a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ with $\sim_S \subseteq P_{\mathcal{GG}} \times P_{\mathcal{GG}}$ as defined in Definition 4.3.17. Let $P_{\mathcal{GG}}/_{\equiv_S}$ be the set $P_{\mathcal{GG}}$ factorized by the equivalence relation $\equiv_S$ induced by $\sim_S$. For each $[p] \in P_{\mathcal{GG}}/_{\equiv_S}$ with $[p] = \{p_1...p_n\}$, let $[p]^+$ be the amalgamated production for $p_1, ..., p_n$ over the port production $port(p_i)$ $(i \in \{1, ..., n\})$ and $Ama(DSP) = \{[p]^+ | [p] \in P_{\mathcal{GG}}/_{\equiv_S}\}$ be the set of amalgamated productions induced by DSP. Then, the *global distributed semantics* is the transformation system $GS(DSP) = A(D)$ for the set of derivations $D = \{\rho = (\emptyset \overset{p_1}{\Rightarrow} \hat{G}_1 \overset{p_1}{\Rightarrow} \hat{G}_2 \overset{p_2}{\Rightarrow} ...) | p_i \in Ama(DSP), i \in \mathbb{N}\}$. △

The global distributed semantics for the distributed system specification given in the Example 4.9 and Example 4.11 is isomorphic to the distributed semantics given in Figure 4.20. A part of the global semantics of the case study in Chapter 2 is given in Example 2.15.

We start the comparison between the compositional operational semantics and the global distributed semantics with the observation, that each step in the distributed semantics can be simulated by the direct derivation of an amalgamated production. The amalgamated production used for this simulation is created from the process productions of the local steps of the open process semantics, from that the transition in the distributed semantics is constructed. For instance, the transition labeled $(p, st(B))$ in the distributed semantics in Figure 4.20 uses the local process productions p and $st(B)$. The transition can be simulated by the amalgamated production $q + st(B)$.

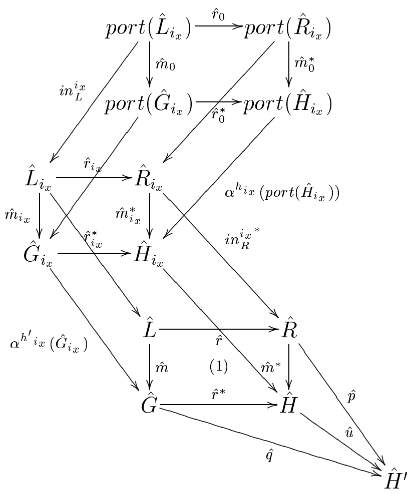
Proposition 4.3.32 (composed distributed steps are amalgamated steps)
Let $DS(DSP) = \langle CG_{DS}, a_{DS} \rangle$ with $(h'_{i_x} : OS(i_x) \to DS(DSP))_{i \in Ind(x), x \in PType}$ be the distributed semantics for a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ composed via the transformation morphisms $(h_{i_x} : Port(DSP) \to OS(i_x))_{i \in Ind(x), x \in PType}$. Then, for each edge $e : a \to b \in CG_{DS}$ with $l(e) = ((p_1, \hat{m}_1), ..., (p_n, \hat{m}_n))$, the distributed morphism $a_{DS}(e) = \hat{r}^*$ is given by the direct amalgamated derivation

$$\begin{tikzcd}
\hat{L} \arrow[r, "\hat{r}"] \arrow[d, "\hat{m}"'] \arrow[dr, phantom, "(1)"] & \hat{R} \arrow[d, "\hat{m}^*"] \\
a_{DS}(a) \arrow[r, "\hat{r}^*"'] & a_{DS}(b)
\end{tikzcd}$$

with the amalgamated production $\sum_{i=1}^{n} p_i : \hat{r}$ at the match $\hat{m}$ induced by the matches $\hat{m}_i$ and the universal property of the colimit object $\hat{L}$ (see also the diagram below). △

Proof

Consider the diagram below for the following explanations.



We have two show that digram (1) above is a pushout diagram. The diagram commutes for following reasons: For each $i \in Ind(x), x \in PType$, $\hat{r}^* \circ \hat{m} \circ in_L^{i_x *} = \hat{r}^* \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x}) \circ \hat{m}_{i_x} = \alpha^{h'_{i_x}}(\hat{H}_{i_x}) \circ \hat{r}^*_{i_x} \circ \hat{m}_{i_x} = \alpha^{h'_{i_x}}(\hat{H}_{i_x}) \circ \hat{m}^*_{i_x} \circ \hat{r}_{i_x} = \hat{m}^* \circ in_R^{i_x *} \circ \hat{r}_{i_x} = \hat{m}^* \circ \hat{r} \circ in_L^{i_x *}$. Since $in_L^{i_x *}$ for $i \in Ind(x), x \in PType$ are jointly surjective, we have $\hat{m}^* \circ \hat{r} = \hat{r}^* \circ \hat{m}$.
Given a distributed graph $\hat{H}'$ and distributed morphisms $\hat{q}$ and $\hat{p}$ with $\hat{p} \circ \hat{r} = \hat{q} \circ \hat{m}$. Since $\hat{q} \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x}) \circ \hat{m}_{i_x} = \hat{q} \circ \hat{m} \circ in_L^{i_x *} = \hat{p} \circ \hat{r} \circ in_L^{i_x *} = \hat{p} \circ in_R^{i_x *} \circ \hat{r}_{i_x}$, there is a unique distributed morphism $u_{i_x} : \hat{H}_{i_x} \to \hat{H}'$ such that $u_{i_x} \circ \hat{r}^*_{i_x} = \hat{q} \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x})$ and $u_{i_x} \circ \hat{m}^*_{i_x} = \hat{p} \circ in_R^{i_x *}$.
Moreover, $u_{i_x} \circ \alpha^{h_{i_x}}(port(\hat{H}_{i_x})) \circ \hat{r}_0^* = u_{i_x} \circ \hat{r}^*_{i_x} \circ \alpha^{h_{i_x}}(port(\hat{G}_{i_x})) = \hat{q} \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x}) \circ \alpha^{h_{i_x}}(port(\hat{G}_{i_x})) = \hat{q} \circ \circ \alpha^{h'_{j_y}}(\hat{G}_{j_y}) \circ \alpha^{h_{j_y}}(port(\hat{G}_{j_y})) = u_{j_y} \circ \alpha^{h_{j_y}}(port(\hat{H}_{j_y})) \circ \hat{r}_0^*$ for all $i \in Ind(x), j \in Ind(y), x, y \in PType$. We can show analogously $u_{i_x} \circ \alpha^{h_{i_x}}(port(\hat{H}_{i_x})) \circ \hat{m}_0^* = u_{j_y} \circ \alpha^{h_{j_y}}(port(\hat{H}_{j_y})) \circ \hat{m}_0^*$ for all $i \in Ind(x), j \in Ind(y), x, y \in PType$. Since $\hat{r}_0^*$ and $\hat{m}_0^*$ are jointly surjective, we have $u_{i_x} \circ \alpha^{h_{i_x}}(port(\hat{H}_{i_x})) = u_{j_y} \circ \alpha^{h_{j_y}}(port(\hat{H}_{j_y}))$ for all $i \in Ind(x), j \in Ind(y), x, y \in PType$. Therefore, there exists a unique distributed morphism $\hat{u} : \hat{H} \to \hat{H}'$ such that $\hat{u} \circ \alpha^{h'_{i_x}}(\hat{H}_{i_x}) = u_{i_x}$ for all $i \in Ind(x), x \in PType$.
Then holds $\hat{u} \circ \hat{r}^* \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x}) = \hat{u} \circ \alpha^{h'_{i_x}}(\hat{H}_{i_x}) \circ \hat{r}^*_{i_x} = u_{i_x} \circ \hat{r}^*_{i_x} = \hat{q} \circ \alpha^{h'_{i_x}}(\hat{G}_{i_x})$. Since the $\alpha^{h'_{i_x}}(\hat{G}_{i_x})$ are jointly surjective, we have $\hat{u} \circ \hat{r}^* = \hat{q}$. We can show analogously $\hat{u} \circ \hat{m}^* = \hat{p}$.
The uniqueness of $\hat{u}$ can be easily shown by the uniqueness of the morphisms $\hat{u}$ and u_{i_x} for $i \in Ind(x), x \in PType$. □

I call the amalgamated production $\sum_{i=1}^{n} p_i : \hat{r}$ of Proposition 4.3.32 for an edge e in the control flow graph of the distributed semantics the *amalgamated production for* e and denote the production by $ama(e) : \hat{r}$. The induced match $\hat{m}$ is called the *amalgamated match* for e. Due to the previous proposition, each step in the distributed semantics can be simulated by an amalgamated direct derivation. Therefore, the distributed semantics can be embedded into the global semantics.

Proposition 4.3.33 (global semantics includes distributed semantics)
Let $DS(DSP) = \langle CG_{DS}, a_{DS} \rangle$ be the distributed semantics for the distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ and $GS(DSP) = \langle CG_{GS}, a_{GS} \rangle$ be the global semantics for DSP. Then, there exists a transformation morphism $h = \langle h^{CG}, \alpha^h \rangle : GS(DSP) \to DS(DSP)$, such that $h^{CG} : CG_{DS} \to CG_{GS}$ is injective and $\alpha^h : V_{h^G}(a_{GS}) \to a_{DS}$ is an isomorphism. △

Proof
We define the transformation morphism $h = \langle h^{CG}, \alpha^h \rangle$ with $h^{CG} = \langle h^G, h^L \rangle$ inductively:

- $h^G(init_{DS}) = init_{GS}$ and $\alpha^h(init_{GS}) = \emptyset$
- Let $h^G(v_{DS}) = v_{GS}$ for $v_{DS} \in G_{DS}$ and $e_{DS} : v_{DS} \to w_{DS}$ an edge in G_{DS} with $l(e_{DS}) = ((p_1, \hat{m}_1), ..., (p_n, \hat{m}_n))$. Let $ama(e_{DS})$ resp. $\hat{m}$ be the amalgamated production for e_{DS} resp. the amalgamated match for e_{DS}. Since the productions p_1,...,p_n may be port productions that are equal to the subproduction used for the amalgamation of $p_1 ... p_n$ by definition, the amalgamated production $ama(e_{DS})$ is isomorphic to the amalgamated production $p(e_{DS}) = \sum_{j \in PS(e_{DS})} p_j : \hat{r}$, where $PS(e_{DS}) = \{i | i = 1...n, p_i \text{ is process production}\}$. Then, there exists an edge $e_{GS} : v_{GS} \to w_{GS}$ with $l(e_{GS}) = (p(e_{DS}), \hat{m})$ in CG_{GS} since we have a match $\hat{m}$ for the amalgamated production $p(e_{DS})$. Due to Proposition 4.3.32, the distributed graphs $a_{GS}(w_{GS})$ and $a_{DS}(w_{DS})$ are isomorphic. Therefore, we define $h^G(w_{DS}) = w_{GS}$ and $h^G(e_{DS}) = e_{GS}$ and $\alpha^h(w_{GS}) : a_{GS}(w_{GS}) \to a_{DS}(w_{DS})$ is the isomorphism between w_{GS} and w_{DS}.

The control flow morphism h^{CG} is total, since each control state in the the control flow graph CG_{DS} resp. CG_{GS} is reachable from the initial state.
The morphism $h^{CG} = \langle h^G, h^L \rangle$ is injective for following reasons: We assume $h^L(l(e_{DS})) = h^L(l(e'_{DS})) = (p, \hat{m})$ for $e_{DS}, e'_{DS} \in CG_{DS}$ and $l(e_{DS}) = ((p_1, \hat{m}_1), ..., (p_n, \hat{m}_n))$ and $l(e'_{DS}) = ((p'_1, \hat{m}'_1), ..., (p'_n, \hat{m}'_n))$. By construction of h^L, we have $p(e_{DS}) = p(e'_{DS})$. By definition of the composition of the open process semantics, the port productions in $l(e_{DS})$ and $l(e'_{DS})$ have to coincide. In order to achieve the same match $\hat{m}$ for the same productions used for the amalgamation, the local matches $\hat{m}_i$ and $\hat{m}'_i$ for $i = 1, .., n$ have to coincide as well. Therefore, we have $e_{DS} = e'_{DS}$. The morphism h^G is injective for edges since h^L is injective and by construction of h^G. The morphism h^G is injective for nodes, since non-injectivity would require, that the same amalgamated production at the same match yields two different derived graphs. □

The previous proposition shows that the distributed compositional semantics can be embedded up to isomorphism into the global semantics. The other direction is not true in general, i.e. the global semantics constructed by the amalgamated productions cannot be embedded up to isomorphism into the distributed semantics. This is based on the fact, that the global semantics operates on a global graph and there does not exist an explicit separation between the process instances for the amalgamated matches. Therefore, process productions of one instance can be applied to other process instances.

Example 4.18 (global semantics hurts instance boundaries)
For this example we consider a process grammar for processes of type P that contains the process production a shown at the top of Figure 4.22. This process production deletes

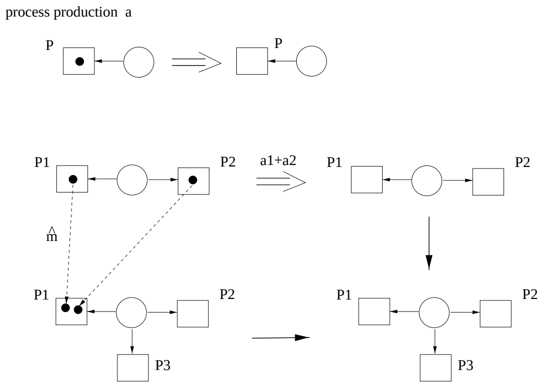


Figure 4.22: *An amalgamated derivation, that cannot be simulated by the compositional distributed semantics.*

one element from the local state of the process. The distributed grammar shall provide three instances of type P, namely the process instance $P1$, $P2$ and $P3$. By definition, each process instance contains an instance of the process production a, denoted by $a1$, $a2$ and $a3$. The synchronization relation shall require that the productions $a1$ and $a2$ are applied synchronously. By definition of the global semantics, we amalgamate the productions $a1$ and $a2$ over the port production to the production $a1 + a2$ shown in the figure to model the synchronized application of $a1$ and $a2$.
Consider now the situation given in the distributed graph in the lower left-hand side of the figure, that shows a distributed system's state, where the three process instances $P1$,$P2$ and $P3$ are connected over a common port. We apply the amalgamated production $a1 + a2$ to this graph at the given match $\hat{m}$. The match hurts the instance boundaries, since it maps the process node $P2$ of the amalgamated production to the process instance $P1$. The application of the amalgamated production deletes both elements from the local state of process instance $P1$.
This transition cannot be simulated by the compositional operational semantics, since in the compositional operational semantics process productions of an instance are applied only to the corresponding process instance. Therefore, it is not possible to delete two elements from $P1$ in one step, since there does not exit a process production for deleting two elements. That means, we cannot split the match for the amalgamated production in a local match for the process production $a1$ to the process instance $P1$ and a local match for the process production $a2$ to the process instance $P2$. $\triangle$

If we restrict the global semantics to those transitions, where the amalgamated productions are applied only to the corresponding process instances , i.e. the match for the amalgamated production can be split into local matches for the process instances, the global semantics is isomorphic to the distributed semantics.

Definition 4.3.34 (restricted global semantics)
Let DSP be a distributed system specification and $Ama(DSP)$ be the set of amalgamated

productions induced by DSP (see Definition 4.3.31). Then, the *restricted global semantics* is the transformation system $RGS(DSP) = A(D)$ for the set of derivations $D = \{\rho = (\emptyset \overset{p_1,\hat{m}_1}{\Rightarrow} \hat{G}_1 \overset{p_2,\hat{m}_2}{\Rightarrow} \hat{G}_2 \overset{p_3,\hat{m}_3}{\Rightarrow} ...)|p_i \in Ama(DSP), i \in \mathbb{N}\}$, such that for all $\rho \in D$, $i, j \in \mathbb{N}$, $i \leq j$ with $\rho(i) = (\hat{G}_i \overset{p_i,\hat{m}_i}{\Rightarrow} \hat{H}_i)$ and $\rho(j) = (\hat{G}_j \overset{p_j,\hat{m}_j}{\Rightarrow} \hat{H}_j)$ holds: if $p_i, p_j \in P_x^k$ for $k \in Ind(x), x \in PType$, are process productions of the same process instance k of type x then $m_j(v) = r_{j1}^* \circ ... \circ r_i^* \circ m_i(v)$ for the process node v of the process productions p_i and p_j. $\triangle$

It is easy to see, that the restricted global semantics is a sub-transformation system of the global semantics as defined in Definition 4.3.31, since the restricted global semantics is generated by a subset of derivations used for the generation of the global semantics.
If we compare the distributed semantics with the restricted global semantics, we can can show that both semantics, i.e. their transformation systems, are isomorphic. This means, that we can consider each distributed computation in the distributed compositional semantics from a global point of view given in the restricted global semantics. Vice versa, each global step in the restricted global semantics can be split into local steps.
The following proposition states, that the transformation morphism h defined in Proposition 4.3.33 is an isomorphism, if we restrict the global semantics to the restricted global semantics.

Proposition 4.3.35
Let $h = \langle h^{CG}, \alpha^h \rangle : GS(DSP) \to DS$ be the transformation morphism defined in Proposition 4.3.33 between the distributed semantics $DS(DSP) = \langle CG_{DS}, a_{DS} \rangle$ and the global semantics $GS(DSP) = \langle CG_{GS}, a_{GS} \rangle$ for a distributed system specification DSP and $h^R = \langle h^{RCG}, \alpha^{h^R} \rangle : RGS(DSP) \to DS(DSP)$ be the transformation morphism h restricted to the restricted global semantics $RGS(DSP) = \langle RCG, a_{RCG} \rangle$. Then, $h^{RCG} : CG_{DS} \to RCG_{GS}$ is bijective and $\alpha^{h^R} : V_{h^{RG}}(a_{RGS}) \to a_{DS}$ is an isomorphism. $\triangle$

Proof
By construction of the control flow graph morphism h^{CG} in Proposition 4.3.33, the control flow graph CG_{DS} is mapped to those parts of CG_{GS}, that apply productions of process instances only to the corresponding process instance. This is due to the fact, that the amalgamated match is induced by the local matches for the process instances. Therefore, the restriction $h^{RCG} : CG_{DS} \to RCG_{GS}$ is total. By Proposition 4.3.33 follows the injectivity of h^{RCG}. We show inductively, that h^{RCG} is surjective as well. For $init_{RCG}$ is $h^{RCG}(init_{DS}) = init_{RGS}$. Let $v_{RGS} \in RCG$, $v_{RGS} \neq init_{RGS}$, there exists an edge $e : w_{RGS} \to v_{RGS}$ in RCG by construction of RCG. By induction premise, there is a $w_{DS} \in CG_{DS}$ with $h^{RCG}(w_{DS}) = w_{RGS}$. Let $l(e) = (p, \hat{m})$ with $p = p_1 + .. + p_k$, $k \in \mathbb{N}$. Due to the restriction to the restricted distributed semantics, the match $\hat{m}$ can be split into matches $\hat{m}_1, ..., \hat{m}_k$ for the process productions $p_1, .., p_k$. By construction of CG_{DS} there exists an edge $e_{DS} : w_{DS} \to v_{DS}$ with $l(e_{DS}) = ((p'_1, \hat{m}_1), ..., (p'_n, \hat{m}_n))$ such that $\{p_1, .., p_k\} \subseteq \{p'_1, ..p'_n\}$ and all productions $p'_i \notin \{p_1, .., p_k\}$ are port productions for $p_1, .., p_k$. By definition of h^{RCG}, we have $h^{RCG}(e_{DS}) = e$ and therefore $h^{RCG}(v_{DS}) = v_{RGS}$.
The family of morphisms α^{h^R} is an isomorphism, since each component is an isomorphism according to Proposition 4.3.33. $\square$

Theorem 4.3.36 (restricted global and distributed semantics are isomorphic)
Given a distributed system specification DSP, the restricted global semantics $RGS(DSP)$ as defined in Definition 4.3.34 and the distributed semantics $DS(DSP)$ as defined in Definition 4.3.27 are isomorphic. $\triangle$

Proof
Follows from the existence of the isomorphism given in Proposition 4.3.35. $\square$

4.4 Distributed Transformation of Attributed Graphs

This section is concerned with a concrete instance of the category $\mathbf{C}$, that is, the category $\mathbf{AGr}(\Sigma)$ of *attributed graphs* introduced in [CL95]. I have chosen this category, since it is used for the case study of the distributed configuration management in Chapter 2.
At first, I give a brief introduction in attributed graph transformation and show then the co-completeness of the model category $\mathbf{Mod}^T_{\mathbf{AGr}(\Sigma)}$ used to construct the pushout in the category $\mathbf{DGr}_{\mathbf{AGr}(\Sigma)}(NTG)$. I conclude with the introduction to distributed graph transformation for distributed graphs with attributed graphs as local states.

4.4.1 Attributed Graph Transformation

Types allow to distinguish nodes or edges within a graph. However, the types of elements must be preserved by morphisms. If types should take the role of parameters (generic types), this strict typing requires a number of morphisms, one for each combination of types, to describe this situation. In order to decrease the amount of morphisms a higher level kind of typing concept is introduced, called *attributed graph transformation*, where high-level types are called *attributes*. The basic idea of attributes is to allow the handling of types abstractly. This is realized by the presence of corresponding variables and a concept of assignment of these variables in an actual situation. Attributed graphs integrate structural (i.e. graphical) aspects of a system with data type aspects (i.e. calculations of values). This leads to compact descriptions in which e.g. well known arithmetic operations need not artificially be coded into graphical structures.

Attributed graphs are defined by partial algebras. I start with a brief introduction in this topic, a more detailed treatment is given in [Bur93,Rei84].

A *signature* $\Sigma = (S, OP)$ consists of a set of sort names S and a family of operation names $OP = (OP_{s_1...s_n,s})_{s_1,...,s_n,s \in S}$ indexed by their arities. Operation names $op \in OP_{s_1,...,s_n,s}$ are denoted by $op : s_1...s_n \to s$. A *sub-signature* of Σ is a signature $\Sigma' = (S', OP')$ where $S' \subseteq S$ and $OP' = (OP'_{s_1...s_n,s})_{s_1,...,s_n,s \in S'}$ is a family of operations such that each set $OP'_{s_1...s_n,s}$ in OP' is a subset of $OP_{s_1...s_n,s}$ in OP.
A *partial* Σ*-algebra* $A = ((A_s)_{s \in S}, (op^A)_{op \in OP})$ is given by an S-indexed family of carrier sets $(A_s)_{s \in S}$ and for each operation name $op : s_1...s_n \to s$ there is a partial mapping $op^A : A_{s_1} \times ... \times A_{s_n} \to A_s$. We denote by $dom(op^A)$ the *domain* of a partial mapping op^A. A Σ-algebra is *total* if op^A is a total mapping for each op in OP.
For a given partial Σ-algebra A, let $B = (B_s)_{s \in S}$ be a S-subset of $A = (A_s)_{s \in S}$, i.e. $B_s \subseteq A_s$ for each sort $s \in S$. Then B is called a *closed subset* of A, if and only if for all $op : s_1...s_n \to s \in OP$ and all $(a_1, ..., a_n) \in (B_{s_1} \times ... \times B_{s_n}) \cap dom(op^A)$ one also have $op^A(a_1, ..., a_n) \in B_s$.
A partial Σ-algebra B is then a *subalgebra* of the partial Σ-algebra A, if and only if

- $(B_s)_{s \in S}$ is a closed subset of A
- for each $op : s_1...s_n \to s \in OP$, op^B is the restriction of op^A to $(B_{s_1} \times ... \times B_{s_n})$, i.e. one has $dom(op^B) = dom(op^A) \cap (B_{s_1} \times ... \times B_{s_n})$.

For a given Σ-algebra A and a sub-signature Σ' of Σ, the Σ'*-reduct on* A is the Σ'-algebra A' where $A'_s = A_s$ for each $s \in S'$ and $op^{A'} = op^A$ for each $op \in OP'$.

A *total* Σ*-homomorphism* $f : A \to B$ between partial Σ-algebras A and B is given by a family of total mappings $f = (f_s : A_s \to B_s)_{s \in S}$ such that for every operation $op : s_1...s_n \to s$ and for each $(a_1, ..., a_n)$ in $(A_{s_1} \times ... \times A_{s_n})$ holds that whenever $(a_1, ..., a_n)$ is in $dom(op^A)$ then $(f_{s_1}(a_1), ..., f_{s_n}(a_n))$ is in $dom(op^B)$ and $op^B(f_{s_1}(a_1), ..., f_{s_n}(a_n)) = f_s(op^A(a_1, ..., a_n))$. A *partial* Σ*-homomorphism* $f : A \rightharpoonup B$ is a total Σ-homomorphism $\bar{f} : dom(A) \to B$ from a closed sub-algebra $dom(A) \subseteq A$ to B.
Given an S-indexed set $X = (X_s)_{s \in S}$ of *variables*, the sorted set $T_\Sigma(X) = (T_\Sigma(X)_s)_{s \in S}$ of *terms* is defined inductively: Each variable x of X_s is a term of sort s and for all operation names $op : s_1...s_n \to s$ and all $t_1 \in T_\Sigma(X)_{s_1}, ..., t_n \in T_\Sigma(X)_{s_n}$ also $op(t_1, ..., t_n)$ is a term of sort s. The *term algebra* $T_\Sigma(X)$ w.r.t. $\Sigma = (S, OP)$ and X has for each sort name s in S the set of terms $T_\Sigma(X)_s$ as carrier set for sort s and for each operation name $op : s_1...s_n \to s$ and terms t_i in $T_\Sigma(X)_{s_i}$ for $i = 1, ...n$ we define $op^{T_\Sigma(X)}(t_1, ..., t_n) := op(t_1, ..., t_n)$. An *assignment* of variables $X = (X_s)_{s \in S}$ in a partial Σ-algebra A is a family of total mappings $(ass_s : X_s \to A_s)_{s \in S}$. The *evaluation* of terms under an assignment $(ass_s : X_s \to A_s)_{s \in S}$ in a partial Σ-algebra A is a partial Σ-homomorphism $ass^A : T_\Sigma(X) \rightharpoonup A$ such that $ass^A_s(x) = ass_s(x)$ for all $x \in X_s$.

In the sequel I introduce attributed graphs as they are presented by Claßen and Löwe in [CL95]. An *attributed graph* is a partial algebra over a signature consisting of two sub-signatures connected by unary operations. One sub-signature describes the graphical part, the other one the attribute part. The unary operations connect some sorts of the graphical part with sorts of the data part. They indicate which graphical sort shall be carry attributes and when, which kind of attribute. This kind of signature is called *attributed graph signature*.

Definition 4.4.1 (attributed graph signature)
An *attributed graph signature* $\Sigma_{ags} = \Sigma_{GS} + \Sigma_{DS} + \Omega_{attr}$ consists of a signature $\Sigma_{GS} = (S_{GS}, OP_{GS})$ where OP_{GS} contains unary operations only, called *graph signature*, a signature $\Sigma_{DS} = (S_{DS}, OP_{DS})$, called *data signature*, and an $S_{GS} \times S_{DS}$-indexed family of operations $\Omega_{attr} = (\Omega_{attr_{s,s'}})_{s \in S_{GS}, s' \in S_{DS}}$. $\triangle$

Example 4.19 (attributed graph signature)
The attributed graph signature used for the case study is given below.

$\Sigma_{GS} =$
  sorts $Doc, Carrier$
  opns $m : Carrier \to Doc$

$\Sigma_{DS} = \textbf{string} + \textbf{nat} +$
opns $empty : \rightarrow string$
$next : natstring \rightarrow string$
$make - private : string \rightarrow string$

$\Omega_{attr} =$
opns $DocID : Carrier \rightarrow nat$
$revnr : Carrier \rightarrow string$
$contens : Carrier \rightarrow string$

The graph signature contains a sort *Doc* for the documents and a sort *Carrier* for carrying the attributes of a document. This attribute carrier is necessary to change attributes. For more information see [LKW93]. The data signature consists of the usual signature for strings and natural numbers. I add some special operations to the data signature needed in the case study. The attribution operations in Ω_{attr} add the actual attributes *DocID*, *revnr* and *content* to the attribute carrier of a document. △

A partial Σ_{ags}-algebra A for an attributed graph signature Σ_{ags} provides a partial GS-algebra A^{GS} for the graph signature GS by the GS-reduct on A, a partial DS-algebra A^{DS} for the data signature DS by the DS-reduct on A and partial mappings $attr^A$ for each operation name $attr$ in Ω_{attr}. Since I do not intend to deal with partial graph structures, we restrict the GS-algebra A^{GS} to be total. As well the operations for operation names in Ω_{attr} have to be total since we require a total attribution of graphical objects. Therefore, admissible Σ_{ags}-algebras are partial Σ_{ags}-algebras where the GS-reduct is total and each operation for an operation name in Ω_{attr} is a total mapping.

Definition 4.4.2 (attributed graph)
Given an attributed graph signature Σ_{ags}, an *attributed graph* w.r.t. Σ_{ags} is a partial Σ_{ags}-algebra A where its GS-reduct algebra A^{GS} is total and the Ω_{attr} operations are total mappings. We call the GS-reduct A^{GS} *graphical algebra*, the DS-reduct A^{DS} *data algebra* and the family of total mappings $attr^A : A^{GS}_s \rightarrow A^{DS}_{s'}$ for all $attr : s \rightarrow s' \in \Omega_{attr}$ *attribution operations.* △

A relation between attributed graphs A_1 and A_2 w.r.t. an attributed graph signature Σ_{ags} is given by a partial Σ_{ags}-homomorphism $f : A_1 \rightharpoonup A_2$. However, I intend a total relation between the data algebras, such that each element of A^{DS}_1 has an image in A^{DS}_2. This is motivated by technical reasons that require total homomorphisms for the data part since colimits of arbitrary partial algebras and partial homomorphisms do not exist in general.

Definition 4.4.3 (attributed graph morphism)
An *attributed graph morphism* $h : A_1 \rightarrow A_2$ w.r.t. an attributed graph signature Σ_{ags} between two attributed graphs A_1 and A_2 w.r.t. Σ_{ags} is a partial Σ_{ags} -homomorphism where the restriction $h^{DS} : A^{DS}_1 \rightarrow A^{DS}_2$ to the data part is a total DS-homomorphism. △

Examples of attributed graphs and attributed graph morphisms with respect to the attributed graph signature of the case study in Example 4.19 are given in Chapter 2, Section 2.3 and following sections.

All attributed graphs and attributed graph morphisms w.r.t. Σ_{ags} form a category $\textbf{Alg}(\Sigma_{\textbf{ags}})$. In the following I fix an attributed graph signature Σ_{ags} and assume that each attributed graph is an attributed graph w.r.t. Σ_{ags} and each attributed graph morphism is an attributed graph morphism w.r.t. Σ_{ags}. I denote the category $\textbf{Alg}(\Sigma_{\textbf{ags}})$ in the following shortly by **AGr**.

Proposition 4.4.4
The category **AGr** *has all colimits.* △

Proof
The pushout construction is given in [CL95]: Given two attributed graph morphisms $h : A \rightarrow B$ and $g : A \rightarrow C$, the pushout is constructed in three steps:

1. Construct the pushout of $h^{GS} : A^{GS} \rightarrow B^{GS}$ and $g^{GS} : A^{GS} \rightarrow C^{GS}$ in the category of total GS-algebras and partial GS-homomorphisms [Löw93].
2. Construct the pushout of $h^{DS} : A^{DS} \rightarrow B^{DS}$ and $g^{DS} : A^{DS} \rightarrow C^{DS}$ in the category of partial DS-algebras and total DS-homomorphisms [Bur86].
3. Add the attribute assignments as unique extensions [LKW93].

The coproduct of a family of attributed graphs $(A_i)_{i \in I}$ is constructed analogously. □

The existence of colimits in $\textbf{Alg}(\Sigma_{ags})$ is based on the restrictions I made on partial Σ_{ags}-algebras to be an attributed graph resp. on partial Σ_{ags}-homomorphisms to be an attributed graph morphism. In [Löw93] is shown that the category $\textbf{Alg}(\Sigma)$ for a signature Σ with total Σ-algebras and partial Σ-homomorphisms has colimits if and only if Σ has unary operation names only. Therefore, I take total unary GS-algebras in order to allow partial homomorphisms between GS-algebras. In [Bur93] the existence of colimits for partial Σ-algebras and total Σ-homomorphisms for arbitrary signatures Σ is shown. Therefore, we restrict to total homomorphisms between the data algebras. Then, we construct the pushout in $\textbf{Alg}(\Sigma_{ags})$ componentwise for the graph and for the data part. The attribution operations are given by the universal property of pushouts.
The transformation of an attributed graph is defined by an *attributed production.* An attributed production consists of a production name and an attributed graph morphism $r : L \rightarrow R$ that has to satisfy some additional conditions: The data algebra of the attributed graphs L and R is the *term algebra* for the data signature DS and the DS-homomorphism of the production morphism r is the identity on the term algebra as I do not intend to rewrite the data algebra. Due to this restriction to attributed graphs and attributed graph morphisms I transform actually only unary algebras. The transformation of unary algebras is studied in [Löw93].
If the data algebra should be rewritten as well, a more general theory of partial algebra transformation is required. For the transformation of arbitrary partial algebras with total homomorphisms in the DPO approach I refer to [LR98].

Definition 4.4.5 (attributed production)
Given an attributed graph signature Σ_{ags} as defined in 4.4.1 and an S_{DS}-indexed set of variables $X = (X_s)_{s \in S_{DS}}$. An *attributed* Σ_{ags}*-production* is a pair $(p : L \overset{r}{\rightarrow} R)$ where p is

a *production name* and $r : L \to R$ is an attributed graph morphism w.r.t. Σ_{ags} such that $r^{DS} = id_{T_{DS}(X)}$ is the identity on the term algebra $T_{DS}(X)$ w.r.t. DS and X and r^{GS} is injective. I call L the *left-hand side* and R the *right-hand side* of the attributed production p. △

An attributed production $(p : L \xrightarrow{r} R)$ can be applied to an attributed graph G if there is an occurrence of the left-hand side L in G, given by a total attributed morphism $m : L \to G$. The application is characterized by the pushout of production morphism r and the occurrence m in category **AGr**.

Definition 4.4.6 (match, direct derivation)
Given an attributed graph signature Σ_{ags} and an attributed production $(p : L \xrightarrow{r} R)$. A *total* attributed graph morphism $m : L \to G$ w.r.t. Σ_{ags}, where m^{GS} is injective, is called a *match* for p in G.
A *direct derivation* via p at m from G to H, denoted by $G \overset{p,m}{\Rightarrow} H$, consists of the pushout of r and m in **AGr**. △

Examples of attributed productions and their application are given in Chapter 2, Section 2.3.

4.4.2 The Category of Distributed Graphs with Attributed Graphs as Local States

I use the category $\mathbf{Alg}(\Sigma_{ags})$ of attributed graphs and attributed graph morphisms with respect to an attributed graph signature Σ_{ags} for the category of local states. As mentioned above, I abbreviate $\mathbf{Alg}(\Sigma_{ags})$ by **AGr** and assume that each attributed graph is an attributed graph over Σ_{ags}. For the definition of the category $\mathbf{DGr_{AGr}}(NTG)$ of distributed graphs with attributed graphs as local states (cf. Section 4.3) I use the network type graph NTG in Figure 2.2.
The category $\mathbf{DGr_{AGr}}(NTG)$ is based on the model category $\mathbf{Mod}^T_{\mathbf{AGr}}$. The definition of a rewrite step in $\mathbf{DGr_{AGr}}(NTG)$ depends strongly on the cocompleteness of $\mathbf{Mod}^T_{\mathbf{AGr}}$.
The category $\mathbf{Mod}^T_{\mathbf{AGr}}$ is cocomplete if **AGr** is cocomplete and left adjoints for translation functors as defined in 4.2.5 exist (cf. Proposition 4.2.6). The cocompleteness of **AGr** is shown in Proposition 4.4.4, such that only the existence of left adjoints remains. I show their existence in three steps:

1. I prove the existence of left adjoints for translation functors with respect to the category $\mathbf{UAlg}(\Sigma)$ of total algebras and partial homomorphisms to a unary signature Σ. I fix in the sequel a unary signature Σ and denote the category shortly by **UAlg**.

2. I prove the existence of left adjoints for translation functors with respect to the category $\mathbf{PAlg}(\Sigma)$ of partial algebras and partial homomorphisms to an arbitrary signature Σ. Again, I fix in the sequel a signature Σ and denote the category shortly by **PAlg**.

3. I combine the left adjoints got by the previous two points to construct the left adjoint for translation functors with respect to the category of attributed graphs.

The left adjoints in point 1 and 2 mainly differ due to the different notion of homomorphisms between algebras. The totality of homomorphisms makes the free construction slightly more complicated, since new elements has freely to be generated to guarantee the totality of

homomorphisms. This is not necessary in the partial case. However, an identification of elements has to be considered in both categories.

4.4.2.1 Left adjoints for translation functors in UAlg

Given a total graph morphism $f : G \to H$ and a model $\mathcal{G}$ in **UAlg**, the idea of the free construction is to refine all nodes of the model over H by the empty algebra (the initial object in the category **UAlg**) if the node is not in the image of f. All nodes in the image of f are refined according to the algebra of the the refined node of its pre-images. The refinement of edges in H depends on the refinement of the edges in G. An identification of elements is necessary if the assignments for the edge given by $\mathcal{G}$ are not unique.

Proposition 4.4.7 (left adjoints w.r.t. $\mathbf{UAlg}(\Sigma)$)
The translation functor $V_f : \mathbf{Mod}^T_{\mathbf{UAlg}}(H) \to \mathbf{Mod}^T_{\mathbf{UAlg}}(G)$ *induced by the total graph morphism* $f : G \to H$ *has a left adjoint* $F_f : \mathbf{Mod}^T_{\mathbf{UAlg}}(G) \to \mathbf{Mod}^T_{\mathbf{UAlg}}(H)$. △

Proof
Let $\mathcal{G} \in \mathbf{Mod}^T_{\mathbf{UAlg}}(G)$, the free construction $\langle F_f(\mathcal{G}), \eta^f_{\mathcal{G}}\rangle$ over $\mathcal{G}$ w.r.t. V_f is defined as follows: For each $v \in H_V$, $\mathcal{H}^0(v) := \bigsqcup_{x \in f_V^{-1}(v)} \mathcal{G}(x)$ with injections $in_x : \mathcal{G}(x) \to \mathcal{H}^0(v)$ is the coproduct of the family $(\mathcal{G}(x))_{x \in f_V^{-1}(v)}$ in **UAlg**. For each edge $e \in H_E$, $\mathcal{H}^0(e) \subseteq \mathcal{H}^0(s(e)) \times \mathcal{H}^0(t(e))$ contains a pair (a, b) if and only if there is an edge $e' \in f_E^{-1}(e)$ such that $\mathcal{G}(e')(a) = b$. Then, we define a relation $\sim_v \subseteq \mathcal{H}^0(v) \times \mathcal{H}^0(v)$ for each $v \in H_V$ by $(a, b) \in \sim_v$ if and only if there is an edge $e \in H_E$ such that $t(e) = v$ and $(x, a), (x, b) \in \mathcal{H}^0(e)$. Let $\equiv_v$ be the least congruence relation on $\mathcal{H}^0(v)$ w.r.t. the inclusion that is generated by $\sim_v$ and where $\mathcal{H}(e) \subseteq \mathcal{H}^0/_{\equiv_{s(e)}} \times \mathcal{H}^0/_{\equiv_{t(e)}}$ defined by $([a], [b]) \in \mathcal{H}(e)$ if and only if $(a, b) \in \mathcal{H}^0(e)$ is a homomorphism.
Then, $F_f(\mathcal{G})(v) = \mathcal{H}^0(v)/_{\equiv_v}$ for each $v \in H_V$ and $F_f(\mathcal{G})(e) = \mathcal{H}(e)$ for each $e \in H_E$. The universal morphism $\eta^f_{\mathcal{G}} = \langle id_G, \alpha^{\eta^f_{\mathcal{G}}}\rangle : \mathcal{G} \to V_f(F_f(\mathcal{G}))$ is defined for each $x \in G_V$ by $\alpha^{\eta^f_{\mathcal{G}}}(x) = nat_{f(x)} \circ in_x$, where $nat_{f(x)} : \mathcal{H}^0(f(x)) \to \mathcal{H}^0(f(x))/_{\equiv_{f(x)}}$ is the natural homomorphism with respect to $\equiv_{f(x)}$.
We show now the universal property of $\langle F_f(\mathcal{G}), \eta^f_{\mathcal{G}}\rangle$: Let $\mathcal{H}'$ be a model over H and $\hat{g} = \langle id_G, \alpha^g\rangle : \mathcal{G} \to V_f(\mathcal{H}')$ be a model morphism in $\mathbf{Mod}^T_{\mathbf{UAlg}}(G)$. Then, the model morphism $\hat{g}^* = \langle id_H, \alpha^{g^*}\rangle : F_f(\mathcal{G}) \to \mathcal{H}'$ is defined as follows: For each $v \in H_V$, there is a unique homomorphism $c_v : \mathcal{H}^0(v) \to \mathcal{H}'(v)$ due to the universal coproduct property of $\mathcal{H}^0(v)$, such that $c_v \circ in_x = \alpha^g(x)$ for each $x \in f_V^{-1}(v)$. Due to the homomorphism theorem (cf. [EM85] page 79), there is a unique morphism $d_v : \mathcal{H}^0(v)/_{\equiv_v} \to \mathcal{H}'(v)$ such that $d_v \circ nat_v = c_v$ for each $v \in H_V$. We define for each $v \in H_V$, $\alpha^{g^*}(v) = d_v$ and get $V_f(\hat{g}^*) \circ \eta^f_{\mathcal{G}} = \hat{g}$ since for each $x \in G_V$, $V_f(\alpha^{g^*})(x) \circ \eta^f_{\mathcal{G}}(x) = \alpha^{g^*}(f(x)) \circ nat_{f(x)} \circ in_x = d_{f(x)} \circ nat_{f(x)} \circ in_x = c_{f(x)} \circ in_x = \alpha^g(x)$. The uniqueness of $\eta^f_{\mathcal{G}}$ follows from the uniqueness of the homomorphisms d_v and c_v for each $v \in H_V$. □

Example 4.20 (left adjoint w.r.t. UAlg)
An example for the left adjoint is given in Figure 4.23. The total graph morphism f is shown at the top of the figure. The mapping is indicated by the characters at the nodes. Important for the example is the identification of the edges from a to b in G to the loop in H. The model $\mathcal{G}$ over G is depicted on the left-hand side of the figure. In the example, we use for

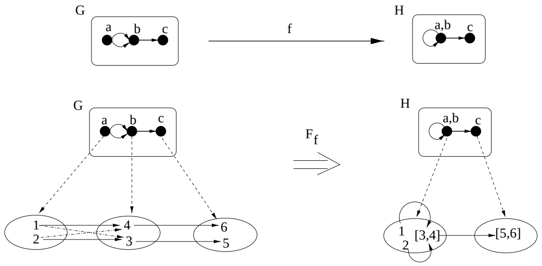


Figure 4.23: *The free construction for* **UAlg**

UAlg the category $\mathbf{Set}^P$ of sets with partial mappings. Sets are unary algebras, where the signature contains only one sort but no operations. The node a of G is refined to a set with the elements 1 and 2, the node b has the elements 3, 4 and node c has the nodes 5 and 6. The two edges in G are refined to two partial mappings (that are total in this case): The one mapping maps 1 onto 4 and 2 onto 3, the other one maps 1 onto 3 and 2 onto 4. The different way of mapping elements causes an identification of the elements 3 and 4 in the refined set of node a, b in H: Since the loop in H is refined according to the two mappings for the pre-image edges in G, we have to map 1 onto 3 but also 1 onto 4. Therefore, the elements 3 and 4 must be identified. Please notice, that the mapping is undefined on $3, 4$. Since the elements 3 and 4 are identified, also the elements 5 and 6 have to be identified in order to achieve a well-defined partial mapping for the edge between a, b and c in H. △

4.4.2.2 Left adjoints for translation functors in PAlg

The existence of the left adjoint for translation functors between model categories in the category **PAlg** can be lifted to the existence of model categories in category **Set**. This is possible since total homomorphisms between algebras are required.

Proposition 4.4.8 (left adjoints for $\mathbf{PAlg}(\Sigma)$)
The translation functor $V_f : \mathbf{Mod}^T_{\mathbf{PAlg}(\Sigma)}(H) \to \mathbf{Mod}^T_{\mathbf{PAlg}}(\Sigma)(G)$ induced by the total graph morphism $f : G \to H$ has a left adjoint $F_f : \mathbf{Mod}^T_{\mathbf{PAlg}(\Sigma)}(G) \to \mathbf{Mod}^T_{\mathbf{PAlg}(\Sigma)}(H)$. △

Proof
The category $\mathbf{Mod}^T_{\mathbf{PAlg}(\Sigma)}(G)$ is isomorphic to the functor category $[Th(G) \to \mathbf{PAlg}(\Sigma)]$ where $Th(G)$ is the theory of G. Furthermore, $\mathbf{PAlg}(\Sigma)$ is isomorphic to the functor category $[Th(\Sigma) \to \mathbf{Set}]$ where $Th(\Sigma)$ is the theory of Σ. Then, $\mathbf{Mod}^T_{\mathbf{PAlg}(\Sigma)}(G)$ is isomorphic to the functor category $[Th(G) \to (Th(\Sigma) \to \mathbf{Set})]$ that is isomorphic to $M(G, \Sigma) = [(Th(G) \times Th(\Sigma)) \to \mathbf{Set}]$. The existence of a left adjoint for the functor $V : M(H, \Sigma) \to M(G, \Sigma)$ is given in [Cor90], that is also a left adjoint for V_f. □

Example 4.21 (left adjoint w.r.t. PAlg)
We use the category **Set** with total mappings for the category **PAlg** in the example depicted in Figure 4.24. We consider the situation already given in Figure 4.23, i.e. the same total graph morphism f and the same model $\mathcal{G}$ over G, where all edges are refined to total graph

morphisms. The elements 3 and 4 resp. 5 and 6 have to be identified again for the same

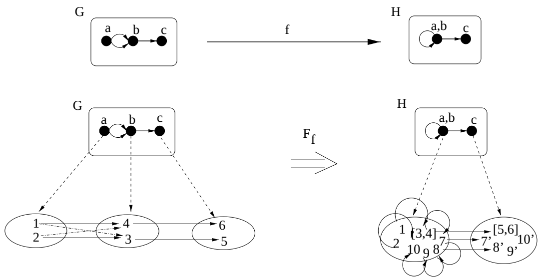


Figure 4.24: *The free construction for* **PAlg**

reasons as explained in the Example 4.20. However, since the edges have to be refined now by total mappings, we have to define the mapping for the loop also for the element $[3, 4]$. This was not necessary in the Example 4.20 because of the partiality of mappings. To define the mapping on $[3, 4]$ a new element is generated. To define the mapping for this newly generated element, a new element is generated again etc. To define the mapping for the edge between a, b and c new elements have to be generated in the set for node c as well. Due to this free generation of elements, we get infinite sets for the nodes a, b and c. △

4.4.2.3 Left adjoints for translation functors in AGr

An attributed graph signature $\Sigma_{ags} = GS + DS + \Omega_{attr}$ is made up of a signature GS with unary operation symbols only, an arbitrary signature DS and a set of unary operations Ω_{attr} (cf. Definition 4.4.1). An attributed graph consists of a unary total algebra over GS and a partial algebra over DS and total mappings for Ω_{attr}. Attributed graph morphisms are partial between the total unary algebras and total between the partial algebras. Due to this combination of totality and partiality resp. unary and non-unary signatures, we can define the left adjoint for translation functors in **AGr** on the basis of the left adjoints for translation functors in **UAlg** and **PAlg**.
In the sequel, we denote the category $\mathbf{AGr}(\Sigma_{ags})$ shortly by **AGr**, the category $\mathbf{UAlg}(GS)$ by **UAlg** and the category $\mathbf{PAlg}(DS)$ shortly by **PAlg**.

Proposition 4.4.9 (left adjoints for AGr)
The translation functor $V_f : \mathbf{Mod}^T_{\mathbf{AGr}}(H) \to \mathbf{Mod}^T_{\mathbf{AGr}}(G)$ induced by the total graph morphism $f : G \to H$ has a left adjoint $F_f : \mathbf{Mod}^T_{\mathbf{AGr}}(G) \to \mathbf{Mod}^T_{\mathbf{AGr}}(H)$. △

Proof
Let $\mathcal{G} \in \mathbf{Mod}^T_{\mathbf{AGr}}(G)$, we define $\mathcal{G}_{GS} \in \mathbf{Mod}^T_{\mathbf{UAlg}}(G)$ by $\mathcal{G}_{GS}(v) = \mathcal{G}(v)^{GS}$ for each $v \in G_V$ and $\mathcal{G}_{GS}(e) = \mathcal{G}(e)^{GS}$ for each $e \in G_E$. Analog, $\mathcal{G}_{DS} \in \mathbf{Mod}^T_{\mathbf{PAlg}}(G)$ is defined by $\mathcal{G}_{DS}(v) = \mathcal{G}(v)^{DS}$ for each $v \in G_V$ and $\mathcal{G}_{DS}(e) = \mathcal{G}(e)^{DS}$ for each $e \in G_E$. Let $\langle F_{f_{GS}}(\mathcal{G}GS), \eta^f_{\mathcal{G}_{GS}}\rangle$ and $\langle F_{f_{DS}}(\mathcal{G}DS), \eta^f_{\mathcal{G}_{DS}}\rangle$ be the free constructions over $\mathcal{G}_{GS}$ resp. $\mathcal{G}_{DS}$ w.r.t. $V^{GS}_f : \mathbf{Mod}_{\mathbf{UAlg}}(H) \to \mathbf{Mod}^T_{\mathbf{UAlg}}(G)$ resp. $V^{DS}_f : \mathbf{Mod}^T_{\mathbf{PAlg}}(H) \to \mathbf{Mod}^T_{\mathbf{PAlg}}(G)$.
Then, the free construction $\langle F_f(\mathcal{G}), \eta_{\mathcal{G}}\rangle$ over $\mathcal{G}$ w.r.t V_f is defined as follows:

1. For each $v \in H_V$, $F_f(\mathcal{G})(v)^{GS} = F_{f_{GS}}(\mathcal{G}_{GS})(v)$ and for each edge $e \in H_E$, $F_f(\mathcal{G})(e)^{GS} = F_{f_{GS}}(\mathcal{G}_{GS})(e)$.

   For each $v \in H_V$, $F_f(\mathcal{G})(v)^{DS} = F_{f_{DS}}(\mathcal{G}_{DS})(v)$ and for each edge $e \in H_E$, $F_f(\mathcal{G})(e)^{DS} = F_{f_{DS}}(\mathcal{G}_{DS})(e)$.

   For each $v \in H_V$ and $op : s \to s' \in \Omega_{attr}$ we define $op^{F_f(\mathcal{G})(v)}$ as follows: By construction of the free construction $\langle F_f(\mathcal{G}_{GS}), \eta^f_{\mathcal{G}_{GS}}, \alpha^{\eta^f_{\mathcal{G}_{GS}}}(x) = nat_{f(x)} \circ in_x$ for each $x \in G_V$, where $in_x : \mathcal{G}(x) \to \mathcal{H}(f(x))$ is a coproduct morphism and $nat_{f(x)} : \mathcal{H}(x) \to F_f(\mathcal{G}_{GS})$ is the natural homomorphism for a congruence relation (cf. the proof of Prop. 4.4.7). Then, there exists a total mapping $c_s : \mathcal{H}(v)_s \to F_f(\mathcal{G})(v)_{s'}$ as the universal coproduct morphism induced by the total mappings $(\alpha^{\eta^f_{\mathcal{G}_{GS}}}(x)_{s'} \circ op^{\mathcal{G}(x)})_{x \in f^{-1}(v)}$. Due to the homomorphism theorem, there is a unique mapping $d_s : F_f(\mathcal{G})(v)_s \to F_f(\mathcal{G})(v)_{s'}$ such that $d_s \circ nat_{x_s} = c_s$. We define then $op^{F_f(\mathcal{G})(v)} = d_s$.

2. We define the universal morphism $\eta^f_{\mathcal{G}} = \langle id_G, \alpha^{\eta^f_{\mathcal{G}}} \rangle : \mathcal{G} \to V_f(F_f(\mathcal{H}))$ for each $x \in G_V$ by $\alpha^{\eta^f_{\mathcal{G}}}(x)^{GS} := \alpha^{\eta^f_{\mathcal{G}_{GS}}}(x)$ and $\alpha^{\eta^f_{\mathcal{G}}}(x)^{DS} := \alpha^{\eta^f_{\mathcal{G}_{DS}}}(x)$.

The universal property of $\langle F_f(\mathcal{G}), \eta^f_{\mathcal{G}} \rangle$ follows from the universal property of $\langle F_f(\mathcal{G}_{GS}), \eta^f_{\mathcal{G}_{GS}} \rangle$ and $\langle \mathcal{H}^{\mathcal{G}}_{DS}, \hat{g}^{\mathcal{G}}_{DS} \rangle$. □

From the existence of left adjoints and the co-completeness of the categories **UAlg**, **PAlg** and **AGr** follows the co-completeness of the model categories.

Corollary 4.4.10 (co-completness)
The categories $\mathbf{Mod}^T_{\mathbf{UAlg}}$, $\mathbf{Mod}^T_{\mathbf{PAlg}}$ *and* $\mathbf{Mod}^T_{\mathbf{AGr}}$ *are co-complete.* △

Proof
The claim follows from Theorem 4.2.7, since the Propositions 4.4.7, 4.4.8 resp. 4.4.9 provide left adjoints and the categories **UAlg**, **PAlg** and **AGr** are co-complete. □

4.4.3 Distributed Graph Transformation with Attributed Graphs as Local Data

Distributed graph transformation with attributed graphs as local states is an instance of distributed graph transformation with local **C**-states (cf. Section 4.3.2), where **C** is the category **AGr** of attributed graphs. However, I do not consider all distributed productions and distributed matches that are possible according to Definition 4.3.7 and Definition 4.3.8. I restrict the distributed productions to those one, where the local morphisms of the production morphism are attributed productions according to Definition 4.4.5. In the same way, I restrict matches to total distributed morphisms, where the local morphisms of the match are matches according to Definition 4.4.6.

Definition 4.4.11 (distributed attributed production and match)
A *distributed attributed production* is a conditional distributed production $p : (\hat{L} \xrightarrow{\hat{r}} \hat{R}, A(p))$ according to Definition 4.3.11 in category $\mathbf{DGr_{AGr}}(NTG)$, such that the network morphism r is injective and $\alpha^r(x)$ is an attributed production according to Definition 4.4.5 for each $x \in dom(r)_V$.

A *distributed attributed match* for a distributed attributed production p is a distributed match $\hat{m} : \hat{L} \to \hat{G}$ according to Definition 4.3.8 in the category $\mathbf{DGr_{AGr}}(NTG)$, such that the network morphism m is injective and $\alpha^m(x)$ is an attributed match according to Definition 4.4.6 for each $x \in L_V$. △

Since I restricted the components of a distributed production to be attributed productions and the components of a distributed match to be attributed matches, I can always construct the pushout of a given distributed production $(p : \hat{r})$ and a total distributed morphism $\hat{m}$. This is based on the fact, that the pushout of the restricted morphisms of the distributed production and the match can be constructed pointwise. In particular, I do not have to check the locality conditions for a match.

Proposition 4.4.12 (pointwise pushout construction)
Let $\hat{r} : \mathcal{L} \to \mathcal{R}$ and $\hat{m} : \mathcal{L} \to \mathcal{G}$ be total injective model morphisms in $\mathbf{Mod}^T_{\mathbf{AGr}}$ such that $\alpha^r(x)$ is an attributed production according to Definition 4.4.5 for each $x \in dom(r)_V$ and $\alpha^m(x)$ is an attributed match according to Definition 4.4.6 for each $x \in L_V$. Then, the pointwise pushout construction in Construction 4.2.9 yields a pushout of $\hat{r}$ and $\hat{m}$ in $\mathbf{Mod}^T_{\mathbf{AGr}}$. △

Proof
Let $(\mathcal{D}, \hat{c}, \hat{d})$ be constructed for $\hat{r}$ and $\hat{m}$ according to Construction 4.2.9. We have to show that $\mathcal{D}$ is a model in $\mathbf{Mod}^T_{\mathbf{AGr}}$. The rest of the proof is then analogously to the proof in 4.2.11. The only problem for the construction of $\mathcal{D}$ are the attributed graph morphisms for edges $e' \in D_E$ such that there exists a $e \in G_E - m_E(L_E)$ with $c_E(e) = e'$ or there exists a $e \in R_E - r_E(L_E)$ with $d_E(e) = e'$. We have to show then, that $\alpha^c(t(e)) \circ \mathcal{G}(e) \circ \alpha^c(s(e))^{-1}$ (resp. $\alpha^d(t(e)) \circ \mathcal{R}(e) \circ \alpha^d(s(e))^{-1}$) is an attributed graph morphism. If $s(e) \in G_V - m_V(L_V)$ then $\alpha^c(s(e))$ is the identity and $\mathcal{D}(e')$ is an attributed graph morphism. If $s(e) = m(y)$ for a $y \in L_V$ then $\alpha^c(s(e))$ is a pushout morphism. By construction of pushouts in the category **AGr** (see Proposition 4.4.4), $\alpha^c(s(e))^{DS}$ is the identity since $\alpha^r(y)^{DS}$ is the identity. Since $\alpha^r(y)^{GS}$ is injective, also $\alpha^c(s(e))$ is injective ([Löw93]). Therefore, $\alpha^c(s(e))^{-1}$ is an attributed graph morphism and so $\mathcal{D}(e')$ is an attributed graph morphism. Analogously can be shown that $\alpha^d(t(e)) \circ \mathcal{R}(e) \circ \alpha^d(s(e))^{-1}$ is an attributed graph morphism. □

This property of distributed productions and distributed matches allows the application of distributed productions without checking of locality conditions. The direct derivation is constructed pointwise as explained in Chapter 2, Section 2.4.1.

Theorem 4.4.13 (direct derivation)
The direct derivation of a given distributed production $p : (\hat{r} : \hat{L} \to \hat{R})$ and a match $\hat{m} : \hat{L} \to \hat{G}$ for p exists always and is characterized by the pushout of $\hat{r}$ and $\hat{m}$ in category $\mathbf{DGr_{AGr}}(NTG)$. △

Proof
The pushout of $\hat{m}$ and $\hat{r}$ exists in the category $\mathbf{DGr_{AGr}}(NTG)$ if the pushout of the underlying model morphisms $\hat{m} : \mathcal{L} \to \mathcal{G}$ and $\hat{r} : \mathcal{L} \to \mathcal{R}$ in the category $\mathbf{Mod_{AGr}}$ exist. Their pushout exist in $\mathbf{Mod_{AGr}}$, if the pointwise pushout construction in 4.2.9 for the restricted morphisms $\hat{r}|_{\mathcal{K}}$ and $\hat{m}|_{\mathcal{K}}$ yields a pushout in $\mathbf{Mod}^T_{\mathbf{AGr}}$ (cf. Theorem 4.2.17 point 1). This is

the case according to Proposition 4.4.12. □

Chapter 5

Formal Description of Graph-Interpreted Temporal Formulas in Distributed Systems and their Checking

This chapter integrates distributed graph transformation and a propositional temporal logic by combining pre-orders with propositional temporal logic. By means of a pre-order over distributed graph morphisms I define *graph-interpreted temporal formulas* and *temporal graph models*. Graph-interpreted formulas are temporal formulas, where the variables are interpreted by distributed morphisms, that form the states of temporal graph models as well. The main ideas of this combination were developed in [GHK97,GHK98].

By means of the pre-order and the existence of least upper bounds I construct a *typical temporal model* that is typical for a class of temporal models in the sense that it satisfies exactly those formulas that are satisfied by all temporal models of this class. The typical model is in general finite and enables to make use of the variety of (automatic) verification techniques developed in the area of temporal logic, e.g. model-checking.

I define a concrete pre-order over distributed graphs in the category of distributed graphs with attributed graphs as local states and show the existence of least upper bounds for this pre-order.

A main part of this chapter is concerned with the compositional verification of temporal formulas. I provide concepts that support the checking of global properties in a distributed system. The idea is the division of a global property into local properties, that can be checked in a distinguished sub-system of the distributed system. The local satisfaction of the local formulas in these sub-systems and the typical model are used to reason to global satisfaction.

The chapter is organized as follows: I start with a brief review in propositional temporal logic and integrate this logic with pre-orders. This integration includes the construction of the typical model for temporal models with pre-orders. In the following section, I define a concrete pre-order over distributed graphs with attributed graphs as local states and construct its least upper bounds. I define temporal graph models and their generation by graph grammars. The last section of this chapter investigates the verification of local and global properties in distributed systems.

5.1 Propositional Temporal Logic

In this section I review the classical syntax and semantics for the propositional fragment of linear temporal logic, according to [Sti92]. The classical semantics is based on transition systems. I introduce temporal models by defining an evaluation of variables and morphisms between this models forming a category. Finally I define when a formula is satisfied.

Definition 5.1.1 (temporal formula)
Let Q be a (countable) set of *propositional variables*. A *temporal formula* is a term generated by the following syntax

$$\Phi ::= Q|\neg\Phi|\Phi_1 \wedge \Phi_2|\Diamond\Phi| \bigcirc \Phi$$

I let $\Phi, \Phi_1 \ldots$ range over the set TF of formulas. △

The operators $\neg$ and $\wedge$ are the usual ones for negation and conjunction. The operators $\bigcirc$ and $\Diamond$ constitute instead the temporal part of the logic. Roughly, since the semantics is given in terms of an evaluation over (sequences of) states of a *transition system*, the *next-time*-operator $\bigcirc$ demands that a formula holds in the immediate successor state, while the *eventually*-operator $\Diamond$ requires that a formula becomes true sooner or later. In the following, I apply the usual abbreviations for implication $\rightarrow$ and disjunction $\vee$. The boolean constant *true* can be defined as a logical tautology, for instance as $q \vee \neg q$ for propositional variables q in Q. The temporal *always*-operator is defined by $\Box\Phi := \neg\Diamond\neg\Phi$.
As an example the temporal formula $q_1 \rightarrow \Diamond(\neg q_2 \vee \neg q_3)$ (containing the variables q_1, q_2, and q_3) has the following intended meaning: if q_1 holds then eventually either q_2 or q_3 is false.
As I mentioned, the classical semantics is based on transition systems. A transition system consists of a set of states, possibly infinite, and a relation on these states representing state transitions. The validity of a formula refers then to *runs* generated by such a transition system intended as infinite sequences of states. The set R of runs for examining validity is a designated subset of all possible runs through the transition system. The following definition of a transition system is taken from [Sti92].

Definition 5.1.2 (transition system)
A *transition system* is a triple $T = \langle S, \rightarrow, R\rangle$ where S is a non-empty set of *states*, $\rightarrow \subseteq S \times S$ is a *transition relation*, and R is a suffix closed set of *runs*. A *run* $\sigma = (\sigma(0)\sigma(1)...)$ with $\sigma(i) \in S$ is a maximal length path through the transition system. Its i-th suffix is given by $\sigma|_i = (\sigma(i)\sigma(i+1)...)$.
A *transition system morphism* $h : \langle S, \rightarrow, R\rangle \rightarrow \langle S', \rightarrow', R'\rangle$ is a function $h : S \rightarrow S'$ such that $h(R) \subseteq R'$, where $h(\sigma(0)\sigma(1)...) = h(\sigma(0))h(\sigma(1))...$ for all $\sigma = \sigma(0)\sigma(1)... \in R$. △

I always assume that every state occurs in some run. Transition system morphisms allow to simulate the runs of the source system within the target system, in order to preserve, as shown later, the notion of *validity* of a formula. Together with an evaluation of the variables, the transition system constitutes a temporal model. The evaluation assigns to each variable a subset of S, where the variable is deemed to be true at each element of this subset.

Definition 5.1.3 (temporal model)
Given a transition system $T = \langle S, \rightarrow, R\rangle$ and a set of propositional variables Q, an *evaluation* $\mathcal{V} : Q \rightarrow \mathcal{P}(S)$ assigns to each variable a set of states $\mathcal{V}(q) \subseteq S$. The pair $\mathcal{M} = \langle T, \mathcal{V}\rangle$ constitutes a *temporal model*.
A *morphism* $h : \mathcal{M} \rightarrow \mathcal{M}'$ *of temporal models* $\mathcal{M} = \langle T, \mathcal{V}\rangle$ and $\mathcal{M}' = \langle T', \mathcal{V}'\rangle$ is a transition system morphism $h : T \rightarrow T'$ such that $s \in \mathcal{V}(q) \Leftrightarrow h(s) \in \mathcal{V}'(q)$ for all $q \in Q$, $s \in S$.
The category of temporal models and their morphisms, with the obvious composition and identities, is denoted by **TM**. △

An evaluation denotes for each variable those states where the variable is deemed to be true. We have now all the components needed to inductively introduce the notion of *satisfaction* of temporal formulas Φ for a given run σ.

Definition 5.1.4 (satisfaction of temporal formula)
Let $\mathcal{M} = \langle T, \mathcal{V}\rangle$ be a temporal model over Q, where $T = \langle S, \rightarrow, R\rangle$. A formula Φ is satisfied by a run $\sigma \in R$, denoted as $\sigma \models \Phi$, if one of the following cases is verified:

- $\sigma \models q \Leftrightarrow \sigma(0) \in \mathcal{V}(q)$ for all $q \in Q$
- $\sigma \models \neg\Phi \Leftrightarrow \sigma \not\models \Phi$
- $\sigma \models \Phi_1 \wedge \Phi_2 \Leftrightarrow \sigma \models \Phi_1$ and $\sigma \models \Phi_2$
- $\sigma \models \bigcirc\Phi \Leftrightarrow \sigma|_1 \models \Phi$
- $\sigma \models \Diamond\Phi \Leftrightarrow \exists i \in \mathbb{N}$ such that $\sigma|_i \models \Phi$

The formula Φ is $\mathcal{M}$-*true*, short $\models_{\mathcal{M}} \Phi$, if $\sigma \models \Phi$ for all $\sigma \in R$. It is *valid*, denoted by $\models \Phi$, if $\models_{\mathcal{M}} \Phi$ for all models $\mathcal{M}$. △

The semantical idea behind morphisms of temporal models is invariance of satisfaction: For each temporal formula Φ and every run σ in $\mathcal{M}$, $\sigma \models \Phi \Leftrightarrow h(\sigma) \models \Phi$.
Given an at least countable set S (i.e., such that there exists an injective function $c : \mathbb{N} \rightarrow S$), it is easy to understand that we could restrict our attention to the full sub-category $\mathbf{TM}^S$ of **TM**, containing only those temporal models whose set of states is contained in S, without losing expressive power. In other words, a formula Φ is valid, that is $\models \Phi$, if and only if $\models_{\mathcal{M}} \Phi$ for all models $\mathcal{M} \in \mathbf{TM}^S$, denoted as $\models_S \Phi$.
Of course, such a result simply says that we could restrict our attention to that family of models, whose set of elements is contained in $\mathbb{N}$ (see again [Sti92]). Nevertheless, such a formulation allows us to investigate the class of temporal models over a given interpretation; and to ask for the characterization of the *minimal model* satisfying a given formula under that interpretation. Such concerns will be the basis for our notion of *graph interpretation* of temporal logic, given in Section 5.2 and Section 5.3.

5.2 Propositional Temporal Logic Enriched by Pre-Orders

In this part, I investigate transition systems as introduced in the previous section, where its set of states is additionally equipped by a *pre-order*. A pre-order $\lhd \subseteq M \times M$ over a set M is a reflexive and transitive relation. Instead of interpreting propositional variables by a set of states where the variable is deemed to be true, I interpret propositional variables in a temporal model equipped by a pre-order by a single state only. Due to the additional structure on the states, I define satisfaction by the pre-order relation in the following sense: A propositional variable is deemed to be true in the state assigned by the interpretation and all states to which the state of the interpretation is related by the pre-order. I construct for an interpretation I a pre-order transition system that is *typical for* I in the sense that its induced model satisfies exactly those formulas that are satisfied by all models induced by the transitions systems with this interpretation. The typical transition system is fully abstract in the sense that any two states which are not distinguishable by constraints from the interpretation I are equal in the typical transition system. The typical model is finite if the set of states used for interpreting propositional variables is finite. This allows to use the concepts introduced in the theory of temporal logic for checking temporal formulas that require mostly a finite state space.

In the following I fix a set M and a pre-order $\lhd \subseteq M \times M$ over M and assume that the pre-order has *least upper bounds*. A least upper bound for a set $A \subseteq M$ is an element $lub(A)$ of M such that $a \lhd lub(A)$ for all $a \in A$ and for each $x \in M$ with $a \lhd x$ for all $a \in A$ follows $lub(A) \lhd x$.

Definition 5.2.1 (temporal pre-order model)
A *pre-order transition system* w.r.t. a pre-order $\lhd \subseteq M \times M$ is a transition system $T_\lhd = \langle S, \rightarrow, R\rangle$ where $S \subseteq M$. A *(temporal) pre-order model* w.r.t. $\lhd$ is a pair $\mathcal{M}_\lhd = \langle T_\lhd, I\rangle$, where $I : Q \rightarrow M$ is an *interpretation* of the propositional variables.
A *morphism* $h : \mathcal{M}_\lhd \rightarrow \mathcal{M}'_\lhd$ between pre-order models $\mathcal{M}_\lhd = \langle T_\lhd, I\rangle$ and $\mathcal{M}'_\lhd = \langle T'_\lhd, I'\rangle$ is a transition system morphism $h : T_\lhd \rightarrow T'_\lhd$ such that $I(q) \lhd s$ if and only if $I'(q) \lhd h(s)$ for each q in Q and s in S. △

Since pre-order transition systems are only a special case of transition systems, morphisms between pre-order transition systems can be defined analogously and lifted up to pre-order models. Pre-order models and morphisms form a category $\mathbf{M}_\lhd$, since composition of morphisms between pre-order models can be defined by the composition of the transition system morphisms. The composed morphism preserves and reflects the pre-order because of the preservation and reflection given by the components. The identities are given by the identities in **Set** that preserve and reflect the pre-order obviously.

Each pre-order model $\mathcal{M}_\lhd = \langle T_\lhd, I\rangle$ with $T_\lhd = \langle S, \rightarrow, R\rangle$ induces a temporal model $F(\mathcal{M}_\lhd) = \langle T_\lhd, \mathcal{V}_I\rangle$, where $\mathcal{V}_I(q) = \{s \in S | I(q) \lhd s\}$ for each $q \in Q$. This shows that the pre-order enables to present a whole set by a single state. I define a functor $F : \mathbf{M}_\lhd \rightarrow \mathbf{TM}$ where $F(\mathcal{M}_\lhd)$ is the temporal model induced by $\mathcal{M}_\lhd$ in $\mathbf{M}_\lhd$. The mapping is identical on the morphisms. The morphism $F(h) : F(\mathcal{M}_\lhd) \rightarrow F(\mathcal{M}'_\lhd)$ with $\mathcal{M}_\lhd = \langle T_\lhd, I\rangle$, $\mathcal{M}'_\lhd = \langle T'_\lhd, I'\rangle$ and $h : T_\lhd \rightarrow T'_\lhd$ is a morphism in $\mathbf{TM}$, since for each state in the transition system $T_\lhd$, $s \in \mathcal{V}_I(q)$ iff $I(q) \lhd s$ iff $I(q) \lhd h(s)$ iff $h(s) \in \mathcal{V}_{I'}(q)$. I use the functor F for lifting the notions of satisfiability over pre-order models.

Definition 5.2.2 (satisfaction in pre-order models)
Given a pre-order model $\mathcal{M}_\lhd = \langle T_\lhd, I\rangle$, a run σ of $T_\lhd$ *satisfies* a temporal formula Φ, written $\sigma \models_{\mathcal{M}_\lhd} \Phi$, if the formula is satisfied in the induced temporal model $F(\mathcal{M}_\lhd)$, that is, $\sigma \models_{F(\mathcal{M}_\lhd)} \Phi$. Similarly for the notion of $\mathcal{M}_\lhd$-truth, $\models_{\mathcal{M}_\lhd} \Phi$, or P-*validity*, $\models_P \Phi$. △

5.2.1 The Typical Pre-order Model

The idea of the *typical pre-order model* or short *typical model* is to build for every possible configuration of truth values of propositional variables a state which implements this configuration, i.e., where only such constraints are satisfied that interpret a variable which is true in the given configuration. Such states are constructed as least upper bounds of the set of states with respect to the pre-order $\lhd$.

Construction 5.2.3 (typical temporal model)
For an interpretation $I : Q \rightarrow M$, let $\mathcal{P}_I(Q)$ be the set of all subsets Q' of Q which are closed under entailment, that is, where for all $q \in Q$, $I(q) \lhd lub(I(Q'))$ implies $q \in Q'$. Then, the typical transition system $T_\lhd(I) = \langle S_I, \rightarrow_I, R_I\rangle$ *has as states all least upper bounds $lub(I(Q'))$ for $Q' \in \mathcal{P}_I(Q)$, the transition relation $\rightarrow_I = S_I \times S_I$ is the full cartesian product and R_I is the set of all paths through $\rightarrow_I$.* △

An example of the construction of the typical model is shown in Chapter 3 in Subsection 3.3.1. The construction is based on a pre-order over distributed graph morphisms that I will introduce in the next section.

Proposition 5.2.4 (finite typical model)
The typical transition system for an interpretation $I : Q \rightarrow M$ is finite if the set $I(Q)$ is finite. △

Proof
If $I(Q)$ is finite, the set $S = \{A | A \subseteq I(Q)\}$ of all subsets of $I(Q)$ is finite as well. Therefore, also the set of least upper bounds of S, $lub(S) = \{lub(A) | A \in S\}$, is finite. Since $S_I \subseteq lub(S)$, S_I is finite. □

Due to the construction of the states of the typical transition system as least upper bounds, they satisfy exactly those constraints from which they are constructed, i.e. $I(q) \lhd lub(I(Q'))$ if and only if $q \in Q'$ for all $q \in Q$ and Q' in $\mathcal{P}_I(Q)$.
The following theorem characterizes the typical graph transition system for I as a final object in $\mathbf{M}_\lhd$.

Theorem 5.2.5 (fully abstract transition system)
The typical transition system $T_\lhd(I)$ for an interpretation I is final in $\mathbf{M}_\lhd$, that is, for each pre-order transition system $T_\lhd$ there exists a unique morphism $!_{T_\lhd} : T_\lhd \rightarrow T_\lhd(I)$. △

Proof
For each pre-order transition system $T_\lhd = \langle S, \rightarrow, R\rangle$ in $\mathbf{M}_\lhd$ we define a morphism $!_{T_\lhd}$: $T_\lhd \rightarrow T_\lhd(I)$ as follows: Let for a state $s \in S$ the set $Q(s) = \{q \in Q | I(q) \lhd s\}$. Then, $Q(s)$ is closed under entailment, i.e., $Q(s) \in \mathcal{P}_I(Q)$, and we define $!_{T_\lhd}(s) = lub(I(Q(s)))$. The mapping $!_{T_\lhd}$ is a morphism from $\langle T_\lhd, I\rangle$ to $\langle T_\lhd(I), I\rangle$ since, by construction, $I(q) \lhd s$ if and only if $I(q) \lhd lub(I(Q(s)))$. Thus, $!_{T_\lhd}$ is a morphism of pre-order transition systems.
Its uniqueness follows from the fact that for a set of propositional variables Q', $lub(I(Q'))$ is the only state in $T_\lhd(I)$ that satisfies exactly the constraints of this set. Since morphisms of pre-order transition systems have to preserve and reflect the satisfaction of propositional variables, the mapping $s \mapsto lub(I(Q(s)))$ is forced by this condition. □

The unique morphism $!_{T_\lhd}$ collapses a potentially infinite pre-order transition system $T_\lhd$ to a finite one. Finality is a categorical way of saying that $T_\lhd(I)$ is both typical and fully abstract: The existence of the morphisms $!_{T_\lhd} : T_\lhd \rightarrow T_\lhd(I)$ for all pre-order transition systems $T_\lhd$ expresses the fact that $T_\lhd(I)$ is typical. This is made precise in the corollary below. The uniqueness of these morphisms implies that $T_\lhd(I)$ is fully abstract: If there are two different states indistinguishable by the interpretation, there can be two different candidates for the definition of $!_{T_\lhd}$.

Corollary 5.2.6 ($T_\lhd(I)$ is typical for I)
The pre-order transition system $T_\lhd(I)$ is typical among the pre-order transition systems with interpretation I in the sense that for all temporal formulas Φ, $\models_{\langle T_\lhd(I),I\rangle} \Phi$ if and only if $\models_{\langle T_\lhd,I\rangle} \Phi$ for all pre-order transition systems $T_\lhd$. $\triangle$

Proof
"$\Longleftarrow$": Since $T_\lhd(I)$ is in $\mathbf{M}_\lhd$, the claim holds.
"$\Longrightarrow$": We show that a pre-order transition system $T_\lhd = \langle S, \rightarrow, R\rangle$ in $\mathbf{M}_\lhd$ with $\not\models_{\langle T_\lhd,I\rangle} \Phi$ implies $\not\models_{\langle T_\lhd(I),I\rangle} \Phi$. If $\not\models_{\langle T_\lhd,I\rangle} \Phi$ there is a run $\sigma \in R$ such that $\sigma \not\models_I \Phi$. Since Theorem 5.2.5 provides a pre-order transition morphism $!_{T_\lhd} : T_\lhd \rightarrow T_\lhd(I)$, and such morphisms preserve and reflect satisfaction of temporal formulas, $!_{T_\lhd}(\sigma) \not\models_I \Phi$ implying $\not\models_{\langle T_\lhd(I),I\rangle} \Phi$. $\square$

5.3 Graph-Interpreted Propositional Temporal Logic

In the previous section, I defined transition systems enriched by a pre-order. In this section, I define a concrete pre-order over distributed graphs in the category $\mathbf{DGr_{AGr}}(NTG)$ of distributed graphs with attributed graphs as local states. This category was introduced in Section 4.4. In the following, I fix a category $\mathbf{DGr_{AGr}}(NTG)$ for a network type graph NTG, where local states are represented by attributed graphs. All distributed graphs and distributed morphisms occurring in the sequel are supposed to be taken from the category $\mathbf{DGr_{AGr}}(NTG)$, abbreviated by $\mathbf{DGr}$. In the sequel, I call a distributed morphism $\hat{f} = \langle f, \alpha^f\rangle : \hat{G} \rightarrow \hat{H}$ in $\mathbf{DGr_{AGr}}(NTG)$ *total* if f is total as well as $\alpha^f(v)$ for each $v \in G_V$ is total.

I provide a distributed graph $\hat{X}$ for the graph-interpreted temporal logic that represents the graphical variables of the logic. By means of a distributed graph morphism starting at $\hat{X}$ the variables are assigned by actual values. The distributed morphism is called an *assignment for* $\hat{X}$.

Definition 5.3.1 (assignment)
Let $\hat{X}$ be a distributed graph, an *assignment* for $\hat{X}$ is each distributed morphism $\hat{a} : \hat{X} \rightarrow \hat{A}$ given in the category $\mathbf{DGr}$ with source $\hat{X}$. I denote the set of all assignments for $\hat{X}$ by $\mathcal{DM}or(\hat{X})$. $\triangle$

The Example 3.1 of our case study in Chapter 3 shows examples of assignments. I fix now a distributed graph $\hat{X}$ and assume that all assignments occurring in the sequel are assignments for $\hat{X}$.

An assignment $\hat{a} = \langle a, \alpha^a\rangle : \hat{X} \rightarrow \hat{A}$ in the category $\mathbf{DGr}$ is not only partial on the network level, but also the components $\alpha^a(v)$ for each $v \in X_V$ are partial, since they are attributed graph morphisms. Therefore, I define the *domain* of an assignment $\hat{a}$ in $\mathbf{DGr}$ by the distributed graph, where the network graph is the domain of the partial network morphism a and the refined graph for each $v \in X_V$ is the domain of the attributed graph morphism $\alpha^a(v)$. The domain of assignments will then be used to define the pre-order over assignments.

Definition 5.3.2 (domain of an assignment)
Let $\hat{a} = \langle a, \alpha^a\rangle : \hat{X} \rightarrow \hat{A}$ be an assignment for $\hat{X}$, the *domain* of $\hat{a}$ is the distributed graph $\hat{X}_a$ over $dom(a)$, where $\mathcal{X}_a(v) = dom(\alpha^a(v))$ for each $v \in dom(a)_V$ and for each edge $e : v \rightarrow w \in dom(a)_E$, I define for each $x \in \mathcal{X}_a(v)$, $\mathcal{X}_a(e)(x) = \mathcal{X}(e)(x)$ if $x \in dom(\mathcal{X}(e))$ and $\mathcal{X}(e)(x) \in \mathcal{X}(w)$, undefined otherwise. $\triangle$

Example 5.1 (domain of an assignment)
Figure 5.1 depicts two assignments $\hat{a} : \hat{X} \rightarrow \hat{A}$ and $\hat{b} : \hat{X} \rightarrow \hat{B}$. The network graph of the distributed graph $\hat{X}$ consists of two nodes connected by an edge. The figure does not show the network nodes explicitly, but only its refinement by attributed graphs. An attributed

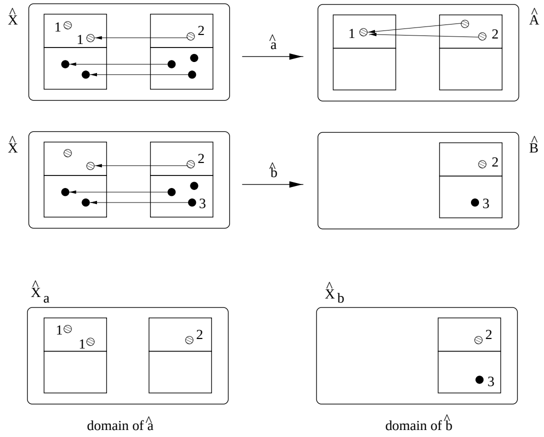


Figure 5.1: *Domains of assignments.*

graph consists of two parts, the data and the graph part, such that I present an attributed graph by two boxes. The black objects are graphical objects, those one with the pattern are objects of the data part. The mappings for the assignments $\hat{a}$ and $\hat{b}$ are indicated by the numbers next to the objects. Objects (graphical and data objects) are mapped to objects with the same number. If there does not exist a number at a node, the mapping is undefined for this object. Please notice, that the mapping for the data part has to be total, whereas the mapping for the graphical part may be partial. The domain $\hat{X}_a$ of $\hat{a}$ contains both network nodes, the whole data part, since the data morphism has to be total and no object in the graph part, since $\hat{a}$ is undefined on all graphical objects. The domain $\hat{X}_b$ of $\hat{b}$ contains only one network node with the complete data part and one graphical object.
The Example 3.2 shows additional examples of the domain of assignments. $\triangle$

The domain $\hat{X}_a$ of an assignment $\hat{a}$ is a distributed graph, since the attributed morphism $\alpha^a(v)^{DS}$ is total for each $v \in dom(a)_V$ due to the definition of an attributed graph morphism in 4.4.3. Therefore, for each $e \in dom(a)_E$ the attributed graph morphism $\mathcal{X}_a(e)^{DS}$ is total as well. It is easy to see that the domain $\hat{X}_a$ is a subgraph of $\hat{X}$ and there is a total distributed morphism $\hat{a}_t : \hat{X}_a \rightarrow \hat{A}$. In this way, we get an equivalent representation of assignments by means of a span.

Definition 5.3.3 (span representation of assignments)
An *assignment for $\hat{X}$ in span representation* is each pair

$$\hat{a} = (\hat{X} \hookleftarrow \hat{X}_a \overset{\hat{a}_t}{\to} \hat{A}),$$

where $\hat{X}_a \subseteq \hat{X}$ is the domain of $\hat{a}$ and $\hat{a}_t$ is the restriction of $\hat{a}$ to $\hat{X}_a$. I denote the set of all assignments for $\hat{X}$ in span representation by $\mathcal{SDM}or(\hat{X})$. △

Proposition 5.3.4 (span representation is equivalent)
The sets $\mathcal{DM}or(\hat{X})$ and $\mathcal{SDM}or(\hat{X})$ are isomorphic. △

Proof
We define the mapping $f : \mathcal{DM}or(\hat{X}) \to \mathcal{SDM}or(\hat{X})$ for each $\hat{a} : \hat{X} \to \hat{A} \in \mathcal{DM}or(\hat{X})$ by $f(a) = (\hat{X} \hookleftarrow \hat{X}_a \overset{\hat{a}_t}{\to} \hat{A})$, where $\hat{X}_a$ is the domain of the assignment $\hat{a}$ and $\hat{a}_t$ is the corresponding total distributed morphism. Vice versa, we define the mapping $f^{-1} : \mathcal{DM}or(\hat{X}) \to \mathcal{SDM}or(\hat{X})$ for each $\hat{a} = (\hat{X} \hookleftarrow \hat{X}_a \overset{\hat{a}_t}{\to} \hat{A}) \in \mathcal{SDM}or(\hat{X})$ by $f^{-1}(a) = \langle f^{-1}(a), \alpha^{f^{-1}(a)} \rangle : \hat{X} \to \hat{A}$, where $dom(f^{-1}(a)) = X_a$ and $f^{-1}(a)|_{X_a} = a_t$, and for each $v \in X_a$, $dom(\alpha^{f^{-1}(a)}(v)) = \mathcal{X}_a(v)$ and $\alpha^{f^{-1}(a)}(v)|_{\mathcal{X}_a(v)} = \alpha^{a_t}(v)$. It can be easily checked, that f^{-1} is the inverse mapping to f. □

I use for the following explanations the most useful representation of an assignment. Examples of assignments in span representation are shown in Example 3.2 of Chapter 3.

5.3.1 Embedding Relation

I define now an *embedding relation* over assignments for a distributed graph $\hat{X}$. An assignment $\hat{a} : \hat{X} \to \hat{A}$ is related by the embedding relation to an assignment $\hat{b} : \hat{X} \to \hat{B}$ if $\hat{a}$ can be embedded into $\hat{b}$ in the sense that the domain of $\hat{a}$ can be embedded into the domain of $\hat{b}$ and $\hat{A}$ provides at least the structure of $\hat{B}$ and possibly some more while respecting the assignments for $\hat{X}$.

Definition 5.3.5 (embedding relation)
Let $\hat{X}$ be a distributed graph, the *embedding relation* $\lhd_{\hat{X}} \subseteq \mathcal{SDM}or(\hat{X}) \times \mathcal{SDM}or(\hat{X})$ is defined as follows: Let $\hat{a} = (\hat{X} \hookleftarrow \hat{X}_a \to \hat{A})$ and $\hat{b} = (\hat{X} \hookleftarrow \hat{X}_b \to \hat{B})$ be two assignments for $\hat{X}$, then $\hat{a} \lhd_{\hat{X}} \hat{b}$ if and only if $\hat{X}_a \subseteq \hat{X}_b$ and there is a total distributed morphism $\hat{f} : \hat{A} \to \hat{B}$ such that the diagram (1) below commutes.

$$\begin{tikzcd}
 & \hat{X}_a \arrow[dl, hook] \arrow[d, hook] \arrow[r, "\hat{a}_t"] \arrow[dr, phantom, "(1)"] & \hat{A} \arrow[d, "\hat{f}"] \\
\hat{X} & \hat{X}_b \arrow[l, hook'] \arrow[r, "\hat{b}_t"'] & \hat{B}
\end{tikzcd}$$

△

Example 5.2 (embedding relation)
Consider the two assignments $\hat{a} = (\hat{X} \hookleftarrow \hat{X}_a \to \hat{A})$ and $\hat{b} = (\hat{X} \hookleftarrow \hat{X}_b \to \hat{B})$ given in the Figure 5.2. I show only the graphical part of the attributed graphs and attributed graph morphisms; the attribute part is treated analogously. The assignment $\hat{a}$ can be embedded

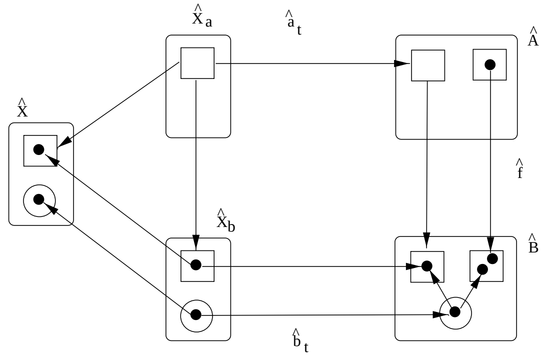


Figure 5.2: *The assignment $\hat{a}$ can be embedded in $\hat{b}$, but not vice versa.*

into $\hat{b}$ since the domain $\hat{X}_a$ of $\hat{a}$ can be embedded into the domain $\hat{X}_b$ of $\hat{b}$ and there exists a total distributed morphism $\hat{f} : \hat{A} \to \hat{B}$ such that the diagram commutes. In fact, there is more than one total distributed morphism commuting the diagram. However, the other way around, the assignment $\hat{b}$ cannot be embedded into $\hat{a}$ since neither the domain $\hat{X}_b$ can be embedded into $\hat{X}_a$ nor there exists a total distributed morphism from $\hat{B}$ to $\hat{A}$. Therefore, $\hat{a} \lhd_{\hat{X}} \hat{b}$, but $\hat{b} \not\lhd_{\hat{X}} \hat{a}$.
The Example 3.2 in Chapter 3 shows an example of the embedding relation of our case study. △

I show now that the embedding relation $\lhd_{\hat{X}}$ on the set of assignments for $\hat{X}$ defines a pre-order, i.e. the embedding relation is reflexive and transitive.

Proposition 5.3.6 ($\lhd_{\hat{X}}$ is pre-order)
Let $\lhd_{\hat{X}}$ be the embedding relation induced by a distributed graph $\hat{X}$ as introduced in Definition 5.3.5, then, $\lhd_{\hat{X}}$ is a pre-order. △

Proof
The relation $\lhd_{\hat{X}}$ is reflexive, since assignments can be embedded in itself by the identities in **DGr**. The relation is transitive, since the composition of total distributed morphisms is again a total distributed morphism that satisfies the conditions of the embedding relation. □

Before I provide the construction of least upper bounds with respect to the embedding relation, I introduce the *intersection* of the domains of a set of assignments.

Lemma 5.3.7 (intersection of domains)
Let $M \subseteq \mathcal{DM}or(\hat{X})$ be a set of assignments $\hat{a} = \langle a, \alpha^a \rangle : \hat{X} \to \hat{A}$ for $\hat{X}$, the intersection of the domains *with respect to M is given by the distributed graph $\hat{X}_{\cap M}$ over $X_{\cap M}$, where*

- *$X_{\cap M} = \bigcap_{\hat{a} \in M} dom(a)$ is the intersection of the network domain graphs $dom(a)$ for each $\hat{a}$ in M and*
- *$\mathcal{X}_{\cap M}(v) = \bigcap_{\hat{a} \in M} \mathcal{X}_a(v)$ for each node $v \in X_{\cap M}$ and $\mathcal{X}_{\cap M}(e) = \bigcap_{\hat{a} \in M} \mathcal{X}_a(e)$.*

△

Proof
The distributed graph $\hat{X}_{\cap M}$ is indeed a distributed graph, since the intersection of subgraphs of X is again a subgraph of X and the intersection $\bigcap_{\hat{a}\in M} \mathcal{X}_a(v)$ of the closed sub-algebras $\mathcal{X}_a(v)$ of $\mathcal{X}(v)$ is again a closed sub-algebra of $\mathcal{X}(v)$. □

Example 5.3 (intersection of domains)
Figure 5.3 shows the intersection of the two assignments $\hat{a}$ and $\hat{b}$ shown in Figure 5.1. At the top of the figure their domains are shown, at the bottom the intersection of their domains. The intersection of the domains contains only one network node, since only one network

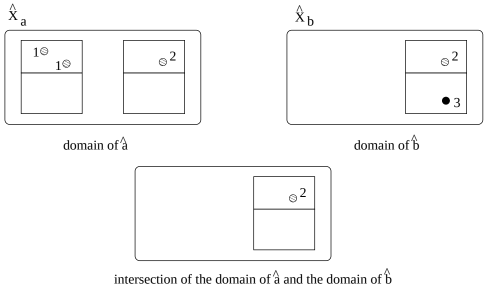


Figure 5.3: *The intersection of the domains $\hat{X}_a$ and $\hat{X}_b$.*

node is in both domains. The refined attributed graph in the intersection is empty for the graphical part, since $\hat{X}_a$ is empty on the graphical part for this refined node. The data part contains the data node with number 2 since this node appears in both domains. △

The least upper bound for a set of assignments M is given by the assignment, where the domain is the colimit of the embeddings from the intersection into the domains of the assignments and the total distributed morphism of the least upper bound is the morphism induced by the colimit property.

Proposition 5.3.8 ($\lhd_{\hat{X}}$ has least upper bounds)
Let $\hat{X}$ be a distributed graph, the embedding relation $\lhd_{\hat{X}}$ induced by $\hat{X}$ as introduced in Definition 5.3.5 has least upper bounds. △

Proof
Given a set $M \subseteq \mathcal{DM}or(\hat{X})$ of assignments $\hat{a} : \hat{X} \to \hat{A}$ over $\hat{X}$. Let $\hat{X}_{\cup M}$ be the colimit object of the family of embeddings $(\hat{X}_{\cap M} \hookrightarrow \hat{X}_a)_{\hat{a}\in M}$ and $\hat{Y}$ be the colimit object of the family of total distributed morphisms $(\hat{X}_{\cap A} \hookrightarrow \hat{X}_a \to \hat{A})_{a\in M}$ induced by the underlying total model morphisms in $\mathbf{Mod}^T_{\mathbf{AGr}}$. Then, there exists a total distributed morphism $\hat{v} = \langle v, \alpha^v\rangle : \hat{X}_{\cup M} \to \hat{Y}$ induced by the universal property of the colimit object $\hat{X}_{\cup M}$ (see the

diagram below for the case of $M = \{\hat{a}, \hat{b}\}$).

$$\begin{tikzcd}
\hat{X}_{\cap M} \arrow[r, hook] \arrow[d] & \hat{X}_b \arrow[r, "\hat{b}_t"] \arrow[d] & \hat{B} \arrow[dd, "\hat{y}_b"] \\
\hat{X}_a \arrow[r, hook] \arrow[d, "\hat{a}_t"'] & \hat{X}_{\cup M} \arrow[r, phantom, "="] \arrow[d, phantom, "="] \arrow[dr, "\hat{v}"] & {} \\
\hat{A} \arrow[rr, "\hat{y}_a"'] & {} & \hat{Y}
\end{tikzcd}$$

Then, the *least upper bound* of M is the assignment $lub(M) = \langle u, \alpha^u\rangle : \hat{X} \to \hat{Y}$, where

- $u = (X \hookleftarrow X_{\cup M} \stackrel{v}{\to} \hat{Y})$ and
- $\alpha^u(x) = (\mathcal{X}(v) \hookleftarrow \mathcal{X}_{\cup M}(v) \stackrel{\alpha^v(x)}{\to} \mathcal{Y}(v(x))$ for each node $x \in X_{\cup M}$.

In order to see that $lub(M)$ is indeed an upper bound, observe that there is an inclusion $\hat{X}_a \hookrightarrow \hat{X}_{\cup M}$ for each $\hat{a} \in M$ and a total distributed morphism $\hat{y}_a : \hat{A} \to \hat{Y}$ for each $\hat{a} \in M$ commuting the newly formed diagrams obtained as colimit injection. That $lub(M)$ is least upper bound follows by the fact that $\hat{X}_{\cup M}$ and $\hat{Y}$ are colimit objects and the universal property of colimits. □

Notice that for the case of two assignments $\hat{a} : \hat{X} \to \hat{A}$ and $\hat{b} : \hat{X} \to \hat{B}$, if $\hat{X}_a = \hat{X}_b$ the distributed graph $\hat{Y}$ is just the pushout of $\hat{a}_t$ and $\hat{b}_t$. On the other hand, if $\hat{X}_a$ and $\hat{X}_b$ are disjoint, then $\hat{Y} = \hat{A} + \hat{B}$ is the disjoint union.

Example 5.4 (least upper bound)
I construct the least upper bound of the two assignments $\hat{a}$ and $\hat{b}$ introduced in Figure 5.1. Their intersection was already calculated in Example 5.3, their domains $\hat{X}_a$ and $\hat{X}_b$ in Example 5.1. The reader can check that for each assignment, where $\hat{a}$ and $\hat{b}$ can be embedded also the least upper bound of $\hat{a}$ and $\hat{b}$ can be embedded.
An additional example is given in Chapter 3 in 3.7. △

5.3.2 Temporal Graph Models

Since the embedding relation is a pre-order, it induces a temporal pre-order model as defined in Definition 5.2.1, where the states of the pre-order transition system are assignments for $\hat{X}$. Therefore, I call the assignments for the states of a transition system *states*. A pre-order transition system for the embedding relation $\lhd_{\hat{X}}$ is called *graph transition system* and a pre-order model with respect to the embedding relation is a *temporal graph model*. The assignments used for the interpretation of the propositional variables are called *graphical constraints*. The idea of graphical constraints is originally developed in [HW95,Koc96,Wag97] to graphically express properties of a certain class of graphs.

Definition 5.3.9 (graph transition system and graph model over $\hat{X}$)
Given the embedding relation $\lhd_{\hat{X}}$ over $\hat{X}$ defined in Definition 5.3.5, a *graph transition system* over $\hat{X}$ is a pre-order transition system GT as defined in Def. 5.2.1 with respect to

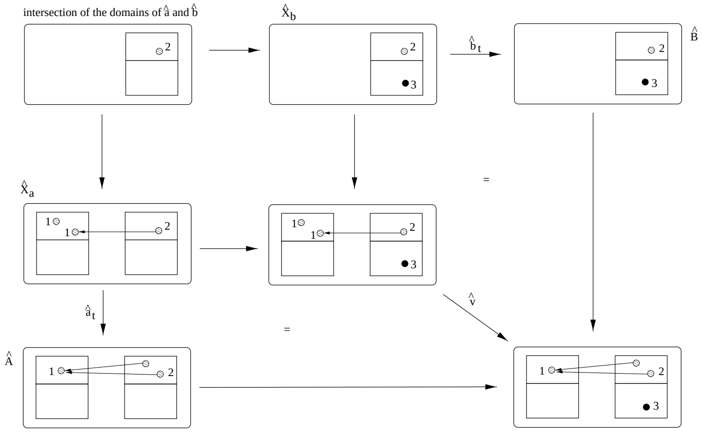


Figure 5.4: *Construction of the least upper bound.*

$\lhd_{\hat{X}}$. A *temporal graph model* over $\hat{X}$ is a pre-order model where the pre-order transition system is a graph transition system. The category of graph models over $\hat{X}$ and graph model morphisms is denoted by $\mathbf{M}_{\lhd_{\hat{X}}}$. △

Since the propositional variables in temporal graph models are interpreted only by a single assignment instead of a set of states, I can represent temporal formulas with respect to temporal graph models in a special representation. This representation shows not only the syntactical temporal formula, but the temporal formula together with its interpretation of propositional variables by assignments. The temporal formula together with the interpretation of the occurring propositional variables by assignments is called *graph-interpreted temporal formula*. Examples of graph-interpreted formulas can be found throughout Chapter 3 of our case study.

Example 5.5 (graph-interpreted temporal formula)
Let $q_1 \wedge \bigcirc q_2$ be a temporal formula with the propositional variables q_1 and q_2. The interpretation of these two variables by assignments $I(q_1)$ and $I(q_2)$ is shown in the Figure 5.5 at the top. The presentation of the temporal formula $q_1 \wedge \bigcirc q_2$ as graph-interpreted temporal formula is shown at the bottom of the figure. This representation integrates directly the assignments for the interpretation of the propositional variables. This allows a graphical specification of distributed system properties as it is shown in Chapter 3. In this way, the specification technique for the properties of a distributed system and the specification technique of the distributed system itself coincide. △

I introduced in Section 5.2 the construction of a typical pre-order model with respect to a pre-order that has least upper bounds and a given interpretation of the propositional variables. Since the embedding relation $\lhd_{\hat{X}}$ has least upper bounds we are able to construct the typical temporal graph model according to Construction 5.2.3.

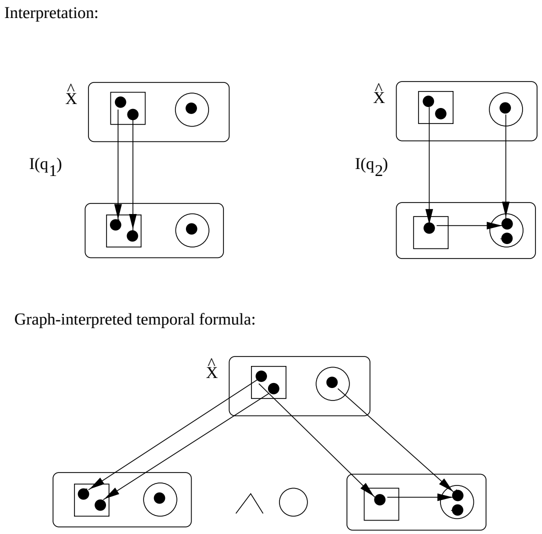


Figure 5.5: *Graph-interpreted temporal formula.*

Theorem 5.3.10 (typical temporal graph model)
Let $\hat{X}$ be a distributed graph and $\lhd_{\hat{X}}$ be the embedding relation as defined in 5.3.5, then the typical model with respect to $\lhd_{\hat{X}}$ exists. △

Proof
A direct consequence of Proposition 5.3.8 and the construction of the typical model given in 5.2.3. □

An example for the construction of the typical model for a set of assignments is shown in Subsection 3.3.1.

5.3.3 Compositional Verification

Checking properties in distributed systems is especially a problem, since there does not exist a global state. I tackle this problem in this section by making use of the typical model introduced in Section 5.2 to reason from the satisfaction of *local temporal formulas* in the temporal graph model induced by the open process semantics to global satisfaction of formulas in the temporal graph model induced by the distributed semantics for a distributed system specification. A local formula is a temporal formula that refers only to one process instance, such that the information given in the open process semantics of this process instance is sufficient to check the local formula. I call the temporal graph model induced by the open process semantics a *process model* and the temporal graph model induced by the distributed semantics the *distributed system model*.

5.3.3.1 Process Models and Distributed System Model

The open process semantics as well as the distributed semantics are given by transformation systems in $\mathbf{DGr_C}(NTG)$. I show how a transformation system in $\mathbf{DGr_C}(NTG)$ generates a graph transition system. All notions and results of this section are fairly independent of the particular graph transformation approach, since the results are based on transformation systems. For the following considerations, it doesn't matter how the transformation system was generated.

A transformation system generates a graph transition system inductively by means of a given assignment for $\hat{X}$ in the initial state of the transformation system. This assignment is the initial state of the graph transition system. The remaining states of the generated graph transition system are the assignments generated by the inductive *continuation* of the initial state along all edges of the control flow graph of the transformation system. The continuation of a state along an edge in the control flow graph is an assignment for $\hat{X}$ that is compatible with the direct derivation used to interpret the edge of the control flow graph in the following sense: Let $\hat{G} \Rightarrow \hat{H}$ be the direct derivation for interpreting the edge of the control flow graph and $\hat{a} : \hat{X} \to \hat{G}$ be a state in $\hat{G}$. I *continue* the state $\hat{a}$ along the direct derivation $\hat{G} \Rightarrow \hat{H}$ by a state $\hat{b} : \hat{X} \to \hat{H}$, such that $\hat{a}$ and $\hat{b}$ agree on all elements that are preserved from $\hat{G}$ to $\hat{H}$. Only elements of $\hat{G}$ that are deleted by the direct derivation may be forgotten from $\hat{a}$ to $\hat{b}$ and only those elements that are new in $\hat{H}$ may be used to extend the state $\hat{b}$ with respect to $\hat{a}$. Due to the fact, that we may have a choice for the extension of the state $\hat{b}$ in the case of several newly generated elements, several states $\hat{b}$ for the continuation are possible for one given direct derivation and one state $\hat{a}$.

Definition 5.3.11 (continuation of states)
Given a transformation system $A = \langle CG, a\rangle$ in $\mathbf{DGr_C}(NTG)$ with $CG = \langle G, L, l, init\rangle$. Let $a(e) : a(v) \to a(w)$ be the distributed morphism for $e : v \to w \in G_E$ and $\hat{a} : \hat{X} \to a(v)$ be a state, the *continuation of* $\hat{a}$ *on* e is given by the largest set $C(\hat{a}, e)$ of states $\hat{b} : \hat{X} \to a(w)$ such that $a(e) \circ \hat{a} \sqsubseteq \hat{b}$ for each $\hat{b} \in C(\hat{a}, e)$. △

Example 5.6 (continuation of states)
Figure 5.6 shows the continuation of the state $\hat{a}$ along the direct derivation from $\hat{G}$ to $\hat{H}$. The derivation deletes the entire round network node from $\hat{G}$ and an element from the square network node. The derivation creates a new round network node in $\hat{H}$ that contains two new elements. The two states $\hat{b}_1$ and $\hat{b}_2$ are two possible continuations of $\hat{a}$ since we have

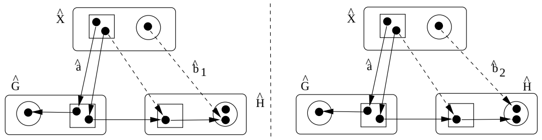


Figure 5.6: *The continuation of* $\hat{a}$ *along* $\hat{G}$ *to* $\hat{H}$.

two possibilities to assign the one element of the round network node in $\hat{X}$ to the two newly generated elements in the round network node of $\hat{H}$. This is possible, since the assignment of

the element in the round network node in $\hat{X}$ is undefined in $\hat{a}$ and the assignment of elements in $\hat{X}$ on newly generated objects is not restricted. The assignment of the two elements in the square network node in $\hat{X}$ is given by $\hat{a}$. The state $\hat{b}$ has to coincide on the preserved elements and has to be undefined on the deleted elements. Therefore, $\hat{b}_1$ and $\hat{b}_2$ have to be undefined on these elements, too. △

The generated graph transition system for a transformation system and an initial state has as states the continuations of the initial state along the paths in the control flow graph. There exists a transition between two states if one state is the continuation of the other one. The set of runs are formed by all the possible paths through the transition relation. Since I require infinite paths, I extend finite paths by infinitely repeating the last state.

Definition 5.3.12 (generated graph transition system)
Let $A = \langle CG, a\rangle$ be a transformation system in $\mathbf{DGr_C}(NTG)$ with $CG = \langle G, L, l, init\rangle$ and $\hat{a}_0 : \hat{X} \to a(init)$ be an assignment in the initial state of A. The *graph transition system generated by* A is given by $GT(A) = \langle S_A, \to_A, R_A\rangle$, where S_A is defined inductively:

- $\hat{a}_0 \in S_A$
- if $\hat{a} : \hat{X} \to a(v) \in S_A$ and there is an edge $e : v \to w \in G_E$ then $\hat{b} : \hat{X} \to a(w) \in S_A$ for each $\hat{b} \in C(\hat{a}, e)$.

The transition relation $\to_A \subseteq S_A \times S_A$ contains a pair $(\hat{a}, \hat{b})$ with $\hat{a} : \hat{X} \to a(v)$ and $\hat{b} : \hat{X} \to a(w)$ if and only if there is an edge $e : v \to w$ in G_E and $\hat{b} \in C(\hat{a}, e)$. The set R_A is the suffix-closed set of all paths through $\to_A$, where we replace each finite path $\sigma = (\sigma(0)\sigma(1)...\sigma(n))$ by the infinite path $\sigma = (\sigma(0)\sigma(1)...\sigma(n)\sigma(n)\sigma(n)...)$. △

Since the open process semantics and the distributed semantics of a distributed system specification are transformation systems, we can generate graph transition systems. Since the initial state of each open process semantics used for the composition is the empty distributed graph, also the initial state of the distributed system semantics is the empty distributed graph. Therefore, the only initial assignment for $\hat{X}$ in the initial state is a totally undefined assignment.

Definition 5.3.13 (distributed system model, process model)
Let $DS(DSP)$ be the distributed semantics for a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ with $\mathcal{GG} = (\mathcal{GG}(k), Ind(k))_{k \in PType}$ as defined in Definition 4.3.27 and $I : Q \to \mathcal{DM}or(\hat{X})$ be an interpretation. Then, the *distributed system model* is given by the temporal graph model $\mathcal{DM} = \langle GT(DS(DSP)), I\rangle$, where $GT(DS(DSP))$ is the graph transition system generated by $DS(DSP)$ and $\emptyset : \hat{X} \to \mathcal{A}(init)$ is the empty assignment.
The *process model* for the process instance i_k for $i \in Ind(k)$, $k \in PType(NTG)$, is the temporal graph model $\mathcal{M}(x_{i_k}) = \langle GT(OS(x_{i_k})), I\rangle$, where $GT(OS(x_{i_k}))$ is the graph transition system generated by the open process semantics $OS(x_{i_k})$ and the empty assignment $\emptyset : \hat{X} \to a_{i_k}(init_{i_k})$. △

I show next, that each run in the distributed system model can be projected on a run in the process models. This is based on the definition of the composition of the open process semantics to the distributed semantics, where the distributed semantics is, roughly speaking, only a restricted part of the union of the open process semantics such that the runs can be constructed by the resulting inclusions.

Proposition 5.3.14 (projection of runs)
Let $DS(DSP) = \langle CG_A, a\rangle$ be the distributed semantics with respect to a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$ with $\mathcal{GG} = (\mathcal{GG}(k), Ind(k))_{k\in PType}$, $\mathcal{DM}$ be the distributed system model for $DS(DSP)$ and $\mathcal{M}(x_{i_k})$ be the process model for $i \in Ind(k)$, $k \in PType$. Then, for each run σ in $\mathcal{DM}$ with $\sigma(0) : \hat{X} \to a(init)$ there is a run σ_{i_k} in $\mathcal{M}(x_{i_k})$. $\triangle$

Proof
Given a run $\sigma = (\sigma(0)\sigma(1)...)$ in $\mathcal{DM}$ with $\sigma(j) : \hat{X} \to a(a_{\sigma(j)})$ for $j \in \mathbb{N}$. Then, we define the sequence $\sigma_{i_k} = (\sigma_{i_k}(0)\sigma_{i_k}(1)...)$ for $i \in Ind(k), k \in PType$ by

$$\sigma_{i_k}(j) = \alpha^{h'_{i_k}}(h'^{G}_{i_k}(a_{\sigma(j)}))^{-1} \circ \sigma(j),$$

where $j \in \mathbb{N}$, $h'_{i_k} = \langle h'^{CG}_{i_k}, \alpha^{h'_{i_k}}\rangle : OS(x_{i_k}) \to DS(DSP)$ with $h'^{CG}_{i_k} = \langle h'^{G}_{i_k}, h'^{L}_{i_k}\rangle : CG_A \to CG_{A_{i_k}}$.
The assignments $\sigma_{i_k}(j)$ for $j \in \mathbb{N}$ are well defined since $\alpha^{h'_{i_k}}(h'^{G}_{i_k}(a_{\sigma(j)}))$ is injective (cf. Proposition 4.3.29). We show that σ_{i_k} is a run in $\mathcal{M}(x_{i_k})$ by induction over the length of the sequence.

start of induction : If $j = 0$, $a(init)$ and $a_{i_k}(init_{i_k})$ are the empty distributed graph $\emptyset$ by construction. Therefore, $\sigma_{i_k}(0) : \hat{X} \to \emptyset$.

induction step : We show $\sigma(n+1) \in C(\sigma(n), e) \Rightarrow \sigma_{i_k}(n+1) \in C(\sigma_{i_k}(n), h'^{G}_{i_k}(e))$. Consider the diagram below for the following considerations, where $e' = h'^{G}_{i_k}(e)$, $a_{\sigma_{i_k}(n)} = h'^{G}_{i_k}(a_{\sigma(n)})$ and $a_{\sigma_{i_k}(n+1)} = h'^{G}_{i_k}(a_{\sigma(n+1)})$.

$$\begin{tikzcd}
a_{i_k}(a_{\sigma_{i_k}(n)}) \arrow[rrr, "\alpha^{h'_{i_k}}(a_{\sigma_{i_k}(n)})"] \arrow[dr, "a_{i_k}(e')"] & & & a(a_{\sigma(n)}) \arrow[dr, "a(e)"] & \\
& a_{i_k}(a_{\sigma_{i_k}(n+1)}) \arrow[rrr, "\alpha^{h'_{i_k}}(a_{\sigma_{i_k}(n+1)})"] & & & a(a_{\sigma(n+1)}) \\
& & \hat{X} \arrow[uull, bend left, "\sigma_{i_k}(n)"] \arrow[ul, "\sigma_{i_k}(n+1)"] \arrow[uur, "\sigma(n)"'] \arrow[urr, "\sigma(n+1)"'] & &
\end{tikzcd}$$

According to Definition 5.3.11 we have to show that $a(e') \circ \sigma_{i_k}(n) \sqsubseteq \sigma_{i_k}(n+1)$:
$a_{i_k}(e') \circ \sigma_{i_k}(n) = a_{i_k}(e') \circ \alpha^{h'_{i_k}}(a_{\sigma_{i_k}(n)})^{-1} \circ \sigma(n) = \alpha^{h'_{i_k}}(a_{\sigma_{i_k}(n+1)})^{-1} \circ a(e) \circ \sigma(n) \sqsubseteq \alpha^{h'_{i_k}}(a_{\sigma_{i_k}(n+1)})^{-1} \circ \sigma(n+1) = \sigma_{i_k}(n+1)$

$\square$

Definition 5.3.15 (projection of runs)
Under the notations of Proposition 5.3.14, I call the run $\sigma_{i_k} = (\sigma_{i_k}(0)\sigma_{i_k}(1)...)$ the *projection of* $\sigma = (\sigma(0)\sigma(1)...)$ on $\mathcal{M}(x_{i_k})$ and denote σ_{i_k} by $pr_{i_k}(\sigma)$. $\triangle$

Since the set of runs is suffix-closed and all runs are generated from runs starting at the initial state, I can generalize the previous proposition to runs that do not necessarily start in the initial state.

Corollary 5.3.16
Lemma 5.3.14 is also true for arbitrary runs σ in $\mathcal{DM}$ with $\sigma(0) : \hat{X} \to a(a_{\sigma(0)})$ and $a_{\sigma(0)} \neq init$. $\triangle$

Proof
There exists for each σ in $\mathcal{DM}$ a σ' in $\mathcal{DM}$ such that $\sigma'(0) = \hat{X} \to a(init)$ and σ is a suffix of σ' by construction of the set of runs R_A of $\mathcal{DM}$ (cf. Def. 5.3.12). The corollary follows then directly from Lemma 5.3.14 and that the set of runs has to be suffix-closed. $\square$

5.3.3.2 Local and Global Satisfaction

The remainder of this section is concerned with the preservation of the satisfaction of temporal formulas by the composition of the open process semantics to the distributed semantics, i.e. the question is: If a temporal formula is satisfied in a process model, is the formula then satisfied in the distributed system model? Unfortunately, this question cannot be answered positively in general, as the following example shows.

Example 5.7 (satisfaction is not preserved by composition)
The graphs G_1 and G_2 in Figure 5.7 shall be data states of the open process semantics $OS(x_1)$ and $OS(x_2)$ for process instances x_1 and x_2, respectively. Actually, data states are distributed graphs in this chapter, but to consider only their network graphs is sufficient in order to clarify the problem. Both graphs are composed to the graph G, that is a data state of the distributed semantics. Let σ be a state of the distributed system model $\mathcal{DM}$. I have shown in the figure only its network graph morphism, that is defined for both boxes in X. The states σ_1 and σ_2 show the projection of σ onto G_1 resp. G_2. They are defined only for

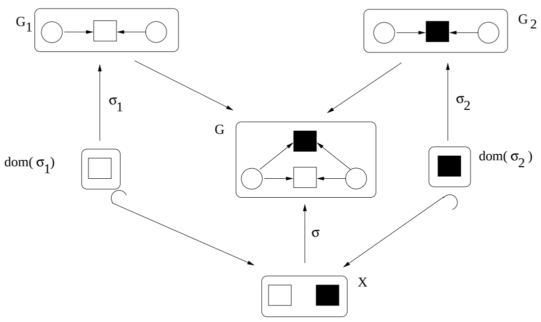


Figure 5.7: *Satisfaction is nor reflected by composition.*

one square node due to the structure of G_1 resp. G_2. If we assume a temporal formula that forbid the occurrence of a white and a black square node at the same time, the states σ_1 and σ_2 satisfy the formula. However, the state σ of the composition does not satisfy this formula, since a white and a black square node occur in G. Therefore, satisfaction of this formula in the process models does not imply satisfaction of the formula in the distributed system model. Vice versa, satisfaction of a formula in the distributed system model is not reflected in the process models. For example, the formula that requires a black and a white square node is satisfied in the distributed system model, but unsatisfied in the process models. $\triangle$

The example has shown that satisfaction is not preserved in general. If we restrict the temporal formulas to so-called *local formulas*, however, we can ensure that for each run in the distributed system model there is a projection on a process model such that the run satisfies a local formula in the distributed system model if and only if the local formula is satisfied in the process model. A local formula is a temporal formula, that refers only to one process instance.

Definition 5.3.17 (local temporal formula)
A temporal formula Φ, where the propositional variables $\{q_1, ..., q_n\} \subseteq Q$ occur, is a *local temporal formula* of type $x \in PType$ with respect to the interpretation $I : Q \to \mathcal{DM}or(\hat{X})$ if for each $i = 1...n$ the graphical constraint $I(q_i) : \hat{X} \to \hat{Y}_i$ satisfies the following conditions:

- there is one and only one process node $v \in dom(I(q_i))$,
- the process node v is of type x and
- $I(q_i)(v)$ is the only process node in Y_i

and for $i = 1...n$, the node $v \in X$ is the same. We call v the *process node of the local formula*. $\triangle$

Example 5.8 (local temporal formula)
Figure 5.8 provides two examples of graph-interpreted temporal formulas. The distributed graph $\hat{X}$ for the graphical variables is shown at the top of the figure. It consists of a network node of type RA and one of type $PortRA$, both containing a document. The temporal formulas are shown below. For the interpretation of the propositional variables I do not show the entire assignment for $\hat{X}$, but only its co-domain distributed graphs. The assignment for $\hat{X}$ can be reconstructed from this representation by the index at the documents: If there is a d at a document in a distributed graph of the formula, the assignment maps the revision archive and the document of $\hat{X}$ to these objects. Analogously, the assignment of the $PortRA$ node and its document is indicated by the p-index. The first graph-interpreted formula is not a local formula, since the co-domain graph of the second assignment refers to two revision archives. The satisfaction of this formula can only be checked by a global view on the distributed system. The lower formula is a local formula, since it contains only one process node in the co-domain graphs for the assignments. The local formula is of type RA since the corresponding process node in the assignment has this type. $\triangle$

For each run in the distributed system model there is a projection on a process model such that the run satisfies a local formula in the distributed system model if and only if the local formula is satisfied in the process model.

Theorem 5.3.18 (preservation of satisfaction)
Let $DSP = (NTG, \mathcal{GG}, \sim_S)$ be a distributed system specification and Φ be a local formula of type $k \in PType$, $\mathcal{DM} = \langle GT(DS(DSP)), I\rangle$ be the distributed system model for DSP and $\lhd_{\hat{X}}$ the embedding relation defined in 5.3.5. Then, for each run σ in $\mathcal{DM}$ there is a $i_k \in Ind(k)$ such that $pr_{i_k}(\sigma) \models_{\mathcal{M}(i_k)} \Phi$ if and only if $\sigma \models_{\mathcal{DM}} \Phi$. $\triangle$

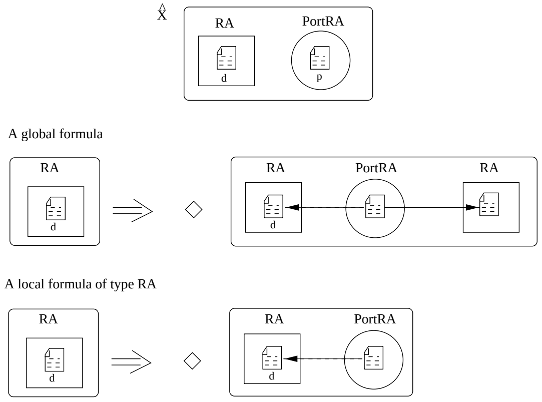


Figure 5.8: *Global and local formulas.*

Proof
Let $v \in X$ be the process node of the local temporal formula Φ and $\sigma = (\sigma(0)\sigma(1)...)$ with $\sigma(j) = \langle f_{\sigma(j)}, \alpha^{f_j}\rangle : \hat{X} \to \hat{Y}_{\sigma(j)}$ a run in $\mathcal{DM}$. Then, choose $i_k \in Ind(k)$ in such a way, such that $f_{\sigma(j)}(v)$ is defined if and only if $f_{pr_{i_k}(\sigma)(j)}(v)$ is defined for all $j \in \mathbb{N}$.
If $f_{\sigma(j)}(v)$ is undefined for all $j \in \mathbb{N}$, each $i_k \in Ind(k)$ can be chosen. If $f_{\sigma(j)}(v)$ is defined for at least one $j \in \mathbb{N}$, $i_k \in Ind(k)$ is uniquely determined, since the process node $f_{\sigma(j)}(v)$ comes from a unique process instance i_k. Moreover, by definition of the continuation holds that whenever $f_{\sigma(j)}(v)$ and $f_{\sigma(j')}(v)$ are defined then $f_{\sigma(j)}(v) = f_{\sigma(j')}(v)$.
If $f_{\sigma(j)}(v)$ is undefined for all $j \in \mathbb{N}$ then $\sigma \not\models_{\mathcal{DM}} \Phi$ by definition of the satisfaction of graph-interpreted temporal formulas. Since, $f_{pr_{i_k}(\sigma)(j)}(v)$ is then undefined for all $j \in \mathbb{N}$ as well, we get $pr_{i_k}(\sigma) \not\models_{\mathcal{M}(x_{i_k})} \Phi$.
Let us assume now a $j \in \mathbb{N}$ such that $f_{\sigma(j)}(v)$ is defined. We show the claim by induction over the structure of the local formula Φ.

start of induction: $q \in Q$

$\Longrightarrow$: $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} q \Leftrightarrow I(q) \lhd_{\hat{X}} pr_{i_k}(\sigma)(0) \Leftrightarrow \hat{X}_{I(q)} \subseteq \hat{X}_{pr_{i_k}(\sigma)(0)}$ and there is a total distributed morphism $\hat{f}$ such that the diagram (1) below commutes.

$$\begin{tikzcd}
 & \hat{X}_{I(q)} \arrow[r] \arrow[d] \arrow[dl, hook] \arrow[dr, phantom, "(1)"] & \hat{Y}_{I(q)} \arrow[d, "\hat{f}"] \\
\hat{X} & \hat{X}_{pr_{i_k}(\sigma)(0)} \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & \hat{Y}_{pr_{i_k}(\sigma)(0)} \arrow[d, "\hat{h}'_{i_k}"] \\
 & \hat{X}_{\sigma(0)} \arrow[r] \arrow[ul] & \hat{Y}_{\sigma(0)}
\end{tikzcd}$$

By definition of $pr_{i_k}(\sigma)$, we have $\hat{X}_{pr_{i_k}(\sigma)(0)} \subseteq \hat{X}_{\sigma(0)}$ and a total distributed morphism $\hat{h}'_{i_k}$ got by the composition of the open process semantics to the distributed semantics such that the diagram (2) commutes. It can be easily checked that $I(q) \lhd_{\hat{X}} \sigma(0)$, i.e. $\sigma \models_{\mathcal{DM}} \Phi$.

$\Longleftarrow$: $\sigma \models_{\mathcal{DM}} \Phi \Leftrightarrow I(q) \lhd_{\hat{X}} \sigma(0) \Leftrightarrow \hat{X}_{I(q)} \subseteq \hat{X}_{\sigma(0)}$ and there is a total distributed morphism $\hat{g}$ such that the diagram (3) below commutes.

$$\begin{tikzcd}
& \hat{X}_{I(q)} \arrow[r] \arrow[dl, hook] \arrow[d, hook] & \hat{Y}_{I(q)} \arrow[d, "\hat{g}"] \\
\hat{X} & \hat{X}_{\sigma(0)} \arrow[l, hook] \arrow[r] \arrow[ur, phantom, "(3)"] & \hat{Y}_{\sigma(0)} \\
& \hat{X}_{pr_{i_k}(\sigma)(0)} \arrow[ul, hook] \arrow[u, hook] \arrow[r] \arrow[ur, phantom, "(4)"] & \hat{Y}_{pr_{i_k}(\sigma)(0)} \arrow[u, "\hat{h}'_{i_k}"']
\end{tikzcd}$$

By definition of $pr_{i_k}(\sigma)$ and Proposition 4.3.29 follows $\hat{X}_{I(q)} \subseteq \hat{X}_{pr_{i_k}(\sigma)(0)}$ and that the distributed morphism $\hat{h'}^{-1}_{i_k} \circ \hat{g}$ is well-defined and total. The commutativity of the resulting diagrams follows from the definition of the morphisms.

induction step $\neg$: $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \neg\Phi \Leftrightarrow pr_{i_k}(\sigma) \not\models_{\mathcal{M}(x_{i_k})} \Phi \Leftrightarrow \sigma \not\models_{\mathcal{DM}} \Phi \Leftrightarrow \sigma \models_{\mathcal{DM}} \neg\Phi$

induction step $\wedge$: $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \Phi_1 \wedge \Phi_2 \Leftrightarrow pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \Phi_1$ and $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \Phi_2 \Leftrightarrow \sigma \models_{\mathcal{DM}} \Phi_1$ and $\sigma \models_{\mathcal{DM}} \Phi_2 \Leftrightarrow \sigma \models_{\mathcal{DM}} \Phi_1 \wedge \Phi_2$.

induction step $\bigcirc$: $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \bigcirc\Phi \Leftrightarrow pr_{i_k}(\sigma)|_1 \models_{\mathcal{M}(x_{i_k})} \Phi \Leftrightarrow \sigma|_1 \models_{\mathcal{DM}} \Phi \Leftrightarrow \sigma \models_{\mathcal{DM}} \bigcirc\Phi$.

induction step $\Diamond$: $pr_{i_k}(\sigma) \models_{\mathcal{M}(x_{i_k})} \Diamond\Phi \Leftrightarrow \exists j \in \mathbb{N}.pr_{i_k}(\sigma)|_j \models_{\mathcal{M}(x_{i_k})} \Phi \Leftrightarrow \exists j \in \mathbb{N}.\sigma|_j \models_{\mathcal{DM}} \Phi \Leftrightarrow \sigma \models_{\mathcal{DM}} \Diamond\Phi$.

□

To check a global property in a distributed system is a problem due to the lack of a global state. Due to Theo. 5.3.18 the satisfaction of a local formula in the distributed system can be checked locally in a process model. The idea is now to divide a global formula in a set of local formulas, check the local formulas in the process models and the typical model checks whether the satisfaction of the local formulas in the process models implies the satisfaction of the global formula in the distributed system model.

Theorem 5.3.19 (typical model implies global satisfaction)
Given a distributed system specification $DSP = (NTG, \mathcal{GG}, \sim_S)$. *Let* $(\Phi_j)_{j \in J}$ *be a family of local formulas of type* $k_j \in PType$, Φ *be a temporal formula,* $\mathcal{DM}$ *be the distributed system model for* DSP *and* $\mathcal{M}_T = \langle T_{\lhd_{\hat{X}}}(I), I \rangle$ *be the typical model for the interpretation* $I : Q \to \mathcal{DM}or(\hat{X})$. *If for each* $j \in J$, $\models_{\mathcal{M}(x_i)} \Phi_j$ *for each* $i \in Ind(k_j)$ *and* $\models_{\mathcal{M}_T} (\bigwedge_{j \in J} \Phi_j \to \Phi)$ *then* $\models_{\mathcal{DM}} \Phi$. △

Proof
Let σ be a run in $\mathcal{DM}$, then $pr_i(\sigma)$ is a run in the process model $\mathcal{M}(x_i)$ for each $i \in Ind(k_j)$. By premise, for each $j \in J$, $pr_i(\sigma) \models_{\mathcal{M}(x_i)} \Phi_j$ for all $i \in Ind(k_j)$. By Theo. 5.3.18 follows $\sigma \models_{\mathcal{DM}} \Phi_j$ for each $j \in J$. If $\models_{\mathcal{M}_T} (\bigwedge_{j \in J} \Phi_j \to \Phi)$ then $\models_{\mathcal{DM}} (\bigwedge_{j \in J} \Phi_j \to \Phi)$ according to Corollary 5.2.6. By definition of satisfaction of temporal formulas follows $\sigma \models_{\mathcal{DM}} \Phi$. □

Chapter 6

Related Work

This chapter relates the approach introduced in this work to existing specification techniques for distributed systems. The comparison is guided by the list of requirements to a specification technique already mentioned in the introduction, that I recall here again. I add the tool support, since a specification technique has more chances to be accepted or to become industrial relevant if "easy-to-use" tools for the specification technique exist.

Structural Aspects: These aspects are concerned with the description of a distributed system's state. In particular, this includes the current topology of the distributed system, i.e. its sites, processes, components, and their connection. This aspect is expressed preferably graphically. Local states of sites, processes, and components together with their relations via shared data form the second part of a distributed state.

Transformational Aspects: A distributed system is an evolving, dynamic structure. Evolution takes place in each part of the system. Topological reconfiguration, i.e. the creation and removal of sites, processes, and components as well as connection changes are a crucial property of systems incorporated in an evolving environment. Besides the necessary state transformations of the processes resp. local components themselves, communication and interconnection aspects, as replication, migration, and remote interactions, have to be described as well.

Compositional Aspects: A distributed system consists of several interacting components. Therefore, a notion of a component and their composition has to be defined in the specification technique in order to support the composition of distributed systems by their components.

Analysis Aspects: Whereas analysis techniques for non-distributed systems, preferably tool-supported, are already a highly desirable feature and focus of research in the last decades, the natural increased complexity of distributed systems make such feature even more desirable in the area of distributed systems.

Tool Support: The tool support is crucial to the acceptance and the industrial relevance of a specification technique.

I firstly consider how far our approach satisfies the following aspects and discuss then related approaches based on graph transformation. Afterwards I consider approaches, that do not consider graph transformations, that are temporal logics, process algebras and object-oriented modeling techniques, in particular UML. The graph transformation approaches and UML are data-state driven approaches, where the central concept is the data state and its transformation. These approaches are well suited for the the structural and transformational aspects. Process algebras and temporal logics are process-driven approaches mainly concerned with the specification of the behavior. Data type specifications are usually not supported by the process-driven approaches, data states are just abstract control states. The process-driven approaches focus more on compositional and analysis aspects.

6.1 The Integrated Approach of Graph Transformation and Temporal Logic

The approach of this work combines a state-driven approach, that is distributed graph transformation, and a process-driven approach, that is temporal logic. The integrated approach benefits from the advantages of these two specification techniques.
My approach especially addresses the structural and the transformational aspects that are the classical domains of (distributed) graph transformation. The topology and the local states of a distributed system are explicitly modeled and are manipulated rule-based.
The compositional aspect is addressed in this work by the restriction to process instances and their open process semantics and the results for the composition of the open process semantics for processes based on the composition of transformation systems provided in [GR98,GR99]. The work shows that the combination of distributed graph transformation with transformation systems is meaningful, since it allows to operate only on local states that can be composed to a distributed system state without operating with a global state. I use the basic concepts for the composition of transformation systems provided in [GR98,GR99]. However, it is only a first step. Große-Rhode provides in [GR98,GR99] more sophisticated concepts for the composition, that allow to specify the compositional operational semantics of a distributed system even more suitable.
The combination of distributed graph transformation with temporal logic allows the graphical and intuitive specification of temporal system properties by graph-interpreted temporal logic formulas. I provide a typical model for a set of graph-interpreted formulas that supports the verification of temporal formulas in infinite temporal models. However, there are graph-interpreted temporal formulas that cannot be checked in the typical model and where additional verification techniques have to be considered. Hence, the typical model is a first promising step in the automatic verification of graph-interpreted formulas in a distributed system model specified by distributed graph transformation, but there has to be done future work to provide complete verification techniques for graph-interpreted formulas.
The same is true for the tool support. The combination with the temporal logic allows to use the tools developed in the field of temporal logic, but there is currently no tool customized for distributed graph transformation and graph-interpreted temporal formulas that support the verification of graphically specified system properties. With respect to tools, approaches like temporal logics or UML are better supported at the moment.

6.2 Graph Transformation Based Approaches

6.2.1 The DPO Approach to Distributed Graph Transformation

The roots of the single pushout approach to distributed graph transformation presented in this work lie in the work of Taentzer, who developed in [Tae96] *distributed graph transformation* in the double-pushout (DPO) approach. In her work, the idea of the two-layered structure of topology and local states is introduced, where local states are specified by labeled graphs. The main focus of her approach is to provide a specification technique for the design

phase of a distributed system, where a global view can be assumed. The distributed system is specified by global productions, that specify also the synchronization of local components. The operational semantics is global as well, given by the set of derivations using the global productions. Due to this global view, this approach is not well suited for supporting the implementation of a distributed system, since a global view in a distributed system does not exist.
To get rid of the global view, the double-pushout approach was extended in [TFKV99] and [FKT98] by concepts for expressing locality in a distributed system. A type of distributed system consisting of local systems connected by export and import interfaces was assumed and a local view on this system is defined. Local view productions are introduced, that correspond to the process productions in this work. Synchronization was expressed by the amalgamation of productions resulting again in global productions. However, the problem of a global operational semantics was left open in both works. Just as in [Tae96], the operational semantics is given by the derivation sequences using the global productions got by the amalgamation. A projection of the global semantics on the local systems provides the operational semantics of the local systems.
With respect to [Tae96] the two approaches [TFKV99,FKT98] generalize the local states to attributed graphs. This generalization raises the power of the specification of local states.
The main advantage of the single pushout approach in comparison with the double pushout approach affects the application of productions and the preservation of locality boundaries in a distributed system. In the double pushout approach a set of conditions has to be checked before a production can be applied. To check these conditions, the boundaries of locality have to be crossed, i.e. whether a production is applicable or not is not decidable in a real distributed environment. Since the conditions for checking the application of a production in the single pushout approach can be reduced or even omitted the single pushout approach is not only useful for the design phase, but also for the execution in a real distributed environment.
All DPO approaches to distributed graph transformation mentioned above are well suited for the description of the statical as well as the transformational aspects. The topology of the distributed system as well as local states are considered. The transformation of both topology and local states is described in an intuitive and graphical way. However, the approaches are neither compositional, since they operate all on a global distributed system state, nor they provide verification concepts. For non-distributed graph transformations several tools are currently developed (see [EEKR99]). For distributed graph transformation, a tool based on AGG [ERT99] is implemented at the Technical University of Berlin. This tool represents another advantage of the SPO approach to distributed graph transformation, since this tool is based on the SPO approach.

6.2.2 Grammars for Distributed Systems

Degano and Montanari introduced in [DM87] a special graph rewriting formalism, called *grammars for distributed systems* (GDS), to specify distributed systems and their evolution with synchronization requirements. The model of a distributed system corresponds to that one given in this work, namely a system consisting of concurrent processes communicating via ports. In the GDS approach, graphs represent not only the current state of the system but also its history. In [MPR99] the history was given up and the use of constraint solving is introduced to cope with the rule matching problem. Both approaches, however, consider only the topological evolution of the distributed system, but do not treat the local data states of processes and ports. Processes are represented by hyperarcs and ports by nodes. Context-free productions, that correspond to the process productions of our approach, equipped by synchronization requests specify the process moves. To apply a context-free production with synchronization requirements means to make all such processes evolve that are necessary to satisfy the synchronization requirements. The operational semantics is global in the sense that there is a global graph where the context-free productions are applied. This is only possible if there is for each context-free production a match, such that the synchronization requests are satisfied. Therefore, the operational semantics is not compositional, since the semantics is given for the whole system at once.

The reconfiguration of the distributed system is not considered in [DM87,MPR99] in the sense that neither ports nor processes are deleted. In [CDM85] a variation of GDS is proposed which uses a more general rewriting mechanism allowing also for system reconfiguration.

Altogether, the GDS approach addresses the statical and the transformational aspects with respect to the topology of a distributed systems, but local states of processes and ports are not considered. The productions are applied on a global state and are not compositional in the sense that the global state is composed from the local states. The specification of distributed system properties and their verification is not considered. The combination of the GDS approach with the constraint solving is an interesting step and enables to make use of the algorithms and tools in this area. These algorithms are mainly used to solve the production matching problem (see also [Rud97]).

6.2.3 Open Graph Transformation Systems

Heckel proposes in [Hec98] a new approach to compositional modeling of concurrent and reactive systems, called *open graph transformation systems* (OGS). In the OGS approach graph transformation is modeled by a new loose semantics. The idea of the loose semantics is that of under-specified transformations. These semantics is an important concept for the compositional verification of behavioral properties of systems specified by temporal logic. The idea of the loose semantics for a system is reflected in our approach in the open process semantics. In both approaches, the semantics describes the possible effects of the environment. The more the environment becomes known the more the semantic, that means possible effects of the environment, is restricted. Therefore, temporal properties satisfied in a single component are also satisfied in a composed component with restricted behavior.

The approach of Heckel is not especially developed for distributed systems, but for the view-based modeling of concurrent and reactive systems. Therefore, an explicit separation between topology and local states is not given. The OGS approach rather considers several views on a specification of a system and its integration to one system model. The specification of system properties is provided by graphical temporal formulas as well. The OGS approach considers the compositional verification of graphical temporal formulas. Satisfaction of formulas in one view can be lifted up to the satisfaction of the formula in the integrated system model. Unfortunately, there exists currently no tool for this new approach.

6.3 Temporal Logics

Temporal Logics are used to formulate the system requirements in an abstract way. Temporal logics are in general not especially developed for distributed systems. They provide special operators to formulate statements on state and event sequences allowed. The specification of the states itself and how states are transformed is not in the viewpoint of temporal logics. Temporal logics are better used to formulate requirements than for design. One main focus in the field of temporal logic is the verification of temporal properties for systems. These verification techniques are often supported by tools [MS96,Man94]. There exist also approaches for the compositional verification of temporal properties (see [MP92]).
The verification techniques and therefore the corresponding tools require in general finite models to check a temporal formula. To cope with infinite models is a current problem in the research of temporal logics. Stirling, Walker, Bradfield and others propose *local model checking* to solve the problem of infinite models [SW89,BS90,SW90,Bra92]. The difference to the usual model-checking approach is that formulas are checked by reference to properties of adjacent states. Roughly speaking, whereas model-checking asks 'does this model satisfy the formula?', local model-checking asks 'does this state in the model satisfy the formula?'. For checking a formula in an infinite model one considers only some finite sub-parts of the model and tries to check the formula in these finite parts. However, local model checking is not fully automatic and requires some intelligence for its use, that is in general considerable knowledge of the model. In this way it is similar to the typical model proposed in this work, that do not solve the problem of infinite models completely as well.

6.4 Process algebras

Process calculi model the behavior of concurrent processes in form of constraints on observable events. Events are able to describe synchronous interactions between processes. Processes are coupled by ports and complex processes can be composed from simpler ones by a number of composition operators. It is possible to model finite data structures by means of the calculus, but this approach is not usable in practice. Thus, e.g. the formalism CCS [Mil89] was combined with algebraic specification concepts for abstract data types leading to the language LOTOS [ISO89].
For the specification of dynamic reconfiguration, the π-calculus [MPW92] has been developed. The difference between data and channel names existing in CCS, is removed in this calculus. It is usual to visualize the process structure by graphs. The textual description of such a reconfiguration becomes unreadable very quickly. The correctness of a specification is proven by the equality of a process behavior with an already known process.
Process algebras are also used to describe the operational semantics of languages for describing distributed systems. An example is the architecture description language DARWIN [MK96] for describing distributed software architectures.

6.5 Object-Oriented Modeling Techniques for Distributed Systems

The object-oriented modeling methods and languages, like OMT [RBP+91], OOD [Boo94], and UML [Rat97], nicely support the development of static system structures, like class and object structures, as well as the description of sequential and concurrent behavior. For the description of allocation of objects and tasks, object replication and migration, remote interactions and dynamic network topologies which are important issues in distributed object systems these techniques promise support, but, however, seem to be insufficient. In the following, we consider the most promising language, UML.

UML is a recent approach to strengthen the efforts of designing an object-oriented modeling language where all main issues of system analysis and design are taken into account. UML provides deployment diagrams to show a system's network topology and the software components, processes and nodes that live on the network nodes. Such a node represents any kind of processing unit. For the description of network topologies nodes may be connected by associations indicating communication paths between nodes. It is possible to describe the kind of association more exactly by the use of stereotypes.
Instances of components as well as processes and objects that live on a node are located within this node. If one component uses the services of another component this is indicated by a dependency between those components. Objects which move during an interaction are modeled by two or more instances on several nodes connected by "becomes" dependencies.
The description of control flow in UML is supported by sequence and collaboration diagrams. Multi threads of control can be expressed by collaboration diagrams containing several active objects. There are different types of control flow containing synchronous and asynchronous communication between processes. Sequence and collaboration diagrams neglect the distributed object allocation, thus it not possible to explicitly address the distribution issues of object systems during the behavior design.
The existing notation of deployment diagrams captures the description of migrating objects but does not provide notational elements to express replication of an object. This restriction seems to be easily removed by inventing a further stereotype which indicates this dependency. But there is no feature to describe the behavior on replicated object structures. Dynamic networking is not addressed by UML. Maybe that is the case because there is no established technique to describe an object-oriented system's behavior without any sequential ordering. The tool support of UML is one of its big advantages, that helped UML to become industrial relevant. However, the tools mainly support the design of a distributed system and are less verification tools. This is based on the fact, that UML has not a sufficient theoretical basis yet.

Chapter 7

Conclusions

The specification and verification of distributed systems is a task of great importance and becomes even more important as distributed systems become more ordinary. In this work, I presented a technique for the specification and verification of distributed systems based on graph transformation and temporal logic. I provided concepts for the specification of distributed systems by distributed graph transformation in the single pushout approach and the specification of system properties by graph-interpreted temporal logic formulas. The integration of graph transformation and temporal logic took place over a special notion of a transition system, where the semantics of temporal logic is based on. Since graph-interpreted temporal logic is only a special kind of propositional temporal logic, we are able to use the checking techniques for temporal formulas developed in the field of temporal logic. To cope with infinite models I provided the construction of a typical model with respect to a set of graph-interpreted formulas that supports, but does not solve completely, the automatic checking of temporal formulas in infinite models. The property of the typical model indicates already that my approach does not solve all problems occurring in the specification of distributed systems. Especially the work with respect to the graph-interpreted temporal logic and the automatic verification of graph-interpreted temporal formulas is in the beginning. In fact, my approach is an interesting step on the way to a specification technique for distributed systems based on graph transformation, but there are still many open problems. I will consider now a few of them.

7.1 A SPO High Level Approach to distributed graph transformation

I developed the single pushout approach to distributed graph transformation, where local states are represented by objects of a category **C** that has to satisfy certain conditions in order to describe a transformation step by a pushout in the corresponding category. This kind of definition gives raise to the question whether it is possible to integrate my approach in the framework of *High-Level Replacement Systems* [EPPG99,EL93], shortly HLR systems. HLR systems provide an abstract framework in which objects of an arbitrary category are transformed. Not every category leads to the same results; they depend very much on the properties of the category, similar to the properties of our category **C**. The focus of HLR systems is on the properties of its instance categories. Minimal conditions are extracted which allow for definitions and theorems concerning, for example, parallelism, embedding or amalgamation. In this way properties can be proven in the biggest class of instance categories. Checking the HLR conditions for our approach may provide us with the results obtained in HLR systems, for example parallelism. Which HLR conditions can be checked successfully for our approach depends on the one hand on the category **C** used for local states and the structure of distributed graphs and distributed morphisms. Further work has to be done to investigate how much these components influence the satisfaction of the HLR conditions.

7.2 Incorporating Distributed Temporal Logics

I combined graph transformation with a propositional temporal logic, that is not especially customized for the requirements of distributed systems. Since the combination is slightly independent of the temporal logic, further work has to be done to find a temporal logic more suitable for distributed systems. There are ongoing efforts to customize temporal logics for distributed systems, see e.g. [Nie97]. The idea is to divide the distributed system in local components and to define component-indexed temporal operators. A component-indexed formula can refer to only one component or to several components. For each single component a transition relation is defined and the composition of transition systems of several components takes place over common actions. For the satisfaction of formulas the composed transition system of those components is taken, that occur in the formula. The integration of such a distributed temporal logic and graph transformation gives the hope to specify distributed systems even more appropriate.

7.3 Incorporating the Grammar in the Typical Model

Until now, the construction of the typical model does not consider the grammar. From a semantical point of view, this can be solved since the typical model is the final object in the category of (a certain class of) transition systems. Therefore, there exists a unique morphism from the (possibly infinite) transition system generated from the grammar to the typical one. This morphism marks a sub-system of the typical graph transition system by its image. This sub-system is still finite, since the typical model is finite. Since transition morphisms preserve and reflect satisfaction (cf. Def. 5.1.3), a formula satisfied resp. not satisfied in the sub-system of the typical model is also satisfied resp. not satisfied in the infinite system. The problem is to find the image of an infinite transition system in the typical model. Further work has to be done to solve this problem.

7.4 Tool Support

This work is a contribution to the ongoing effort to establish (distributed) graph transformation as a specification technique for concurrent, reactive and (in my case) distributed systems. The establishment is simplified if the specification technique is supported by a tool. In particular, if the technique shall become industrial relevance a tool is mandatory. Therefore, the implementation of a tool is one of the most challenging aspects of future work. Tools shall support the design of a distributed system by distributed graph transformation as well as the specification of graph-interpreted temporal formulas and their verification. The implementation of a tool for the design of distributed systems by distributed graph transformation is addressed at the Technical University of Berlin, but the work is still in its beginnings. An analysis tool would give the approach additional acceptance.

Bibliography

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the 5th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1990.

[BGT89] R.M. Burstall, J.A. Goguen, and A. Tarlecki. Some Fundamental Algebraic Tools for the Semantics of Computation – Part 3: Indexed Categories. LFCS Report Series ECS-LFCS-89-90, 1989.

[Boe87] B. Boehm. Industrial software metrics top ten list. *IEEE Software*, pages 84–85, Sept. 1987.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[Bra92] Julian Charles Bradfield. *Verifying Temporal Properties of Systems*. Birkhaeuser, 1992.

[BS90] Julian Bradfield and Colin Stirling. Local Model Checking For Infinite State Spaces. LFCS Report Series ECS-LFCS-90-115, 1990.

[Bur86] P. Burmeister. A model theoretic oriented approach to partial algebras. *Mathematical Research – Mathematische Forschung*, 32:257–274, 1986.

[Bur93] P. Burmeister. Partial algebras – an introductory survey, 1993.

[BW95] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall Int., second edition, 1995.

[CDM85] A. Corradini, P. Degano, and U. Montanari. Specifying highly concurrent data structure manipulation. In G.Bucci and G.Valle, editors, *COMPUTING 85:A Broad Perspective of Current Developments*. Elsevier Science, 1985.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, volume 131 of *LNCS*. Springer, May 1981.

[CELP96] A. Corradini, H. Ehrig, M. Löwe, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science*, number 1073 in LNCS, pages 56–74. Springer, 1996.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems*, number 8(2), pages 244–263, 1986.

[CL95] Ingo Classen and Michael Löwe. Scheme evolution in object oriented models:A graph transformation approach. In *Proc. Workshop on Formal Methods at the ISCE'95*, 1995.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, chapter Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. World Scientific, 1997.

[Con97] Reidar Conradi, editor. *Proceedings of the 7th Int. Workshop on Software Configuration Management*, volume 1235 of *LNCS*, Berlin u. a., 1997. Springer.

[Cor90] Andrea Corradini. *An Algebraic Semantics for Transition Systems and Logic Programming*. PhD thesis, University of Pisa, 1990.

[CW97] Reidar Conradi and Bernhard Westfechtel. Towards a uniform version model for software configuration management. In *[Con97]*, pages 1–17, 1997.

[DM87] P. Degano and U. Montanari. A model for distributed systems based on graph rewritings. *Journal of ACM, 34(2)*, 1987.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. II: Applications, Languages, and Tools*. World Scientific, 1999.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, chapter Algebraic Approaches to Graph Transformation Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [Roz97], 1997.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. III: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.

[EKR91] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *4th International Workshop on Graph Grammars and Their Application to Computer Science*. Springer, 1991. Lecture Notes in Computer Science 532.

[EL93] H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, 1993.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In *Formal models and semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.

[EPPG99] H. Ehrig, F. Parisi-Presice, and M. Gajewski. High-level replacement systems with application to algebraic specification and petri nets. In G. Rozenberg, editor, *[EKMR99]*. World Scientific, 1999.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual Symposium on Switching ans Automata Theory*, pages 167–180, 1973.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. *[EEKR99]*, chapter The AGG approach: Language and environment. World Scientific, 1999.

[Est95] Jacky Estublier, editor. *Proceedings of the 4th and 5th Int. Workshop on Software Configuration Management*, volume 1005 of *LNCS*, Berlin u. a., 1995. Springer.

[Fis98] Ingrid Fischer. *Modelling Neural Networks with Graph Transformation Systems*. PhD thesis, Universität Erlangen-Nürnberg, 1998. Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), 31/11.

[FKT98] I. Fischer, M. Koch, and G. Taentzer. Local views on distributed systems and their communication. In G.Engels and G. Rozenberg, editors, *Proc. of 6th International Workshop on Theory and Apllication of Graph Transaformation*, number tr-ri-98-201, pages 40–47, 98.

[GHK97] Fabio Gaducci, Reiko Heckel, and Manuel Koch. Combining Graph Transformations with Temporal Logic. 1997. Extended abstract accepted for Workshop on General Theory of Graph Transformation Systems.

[GHK98] F. Gadducci, R. Heckel, and M. Koch. Model Checking Graph-Interpreted Temporal Formula. In G. Engels and G. Rozenberg, editors, *Proc. of 6th Int. Workshop on Theory and Graph Transformation*, number tr-ri-98-201, pages 292–299, Paderborn, 1998.

[GR98] Martin Große-Rhode. Algebra transformation systems and their composition. In E. Astesiano, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'98)*, number 1382 in LNCS, pages 107–122. Springer, April 1998.

[GR99] Martin Große-Rhode. A compositional comparison of specifications of the alternating bit protocol in CCS and UNITY based on algebra transformation systems. In *Proc. of the 1st International Workshop on Integrated Formal Methods (IFM'99)*, York, UK, June 1999. to appear in Springer LNCS.

[HCEL96] R. Heckel, A. Corradini, H. Ehrig, and M. Loewe. Horizontal and vertical structuring of typed graph transformation systems. In *Math. Struc. in Comp. Science*, pages 6(6):613–648, 1996.

[Hec95] Reiko Heckel. Algebraic Graph Transformations with Application Conditions. Master's thesis, TU Berlin, 1995.

[Hec98] Reiko Heckel. *Open Graph Transformation Systems - A New Approach to Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, TU Berlin, 1998.

[HEWC97] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Integrating the specification techniques of graph transformation and temporal logic. In *Proc. of MFCS'97, Bratislava*, 1997.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[HMTW95] Reiko Heckel, Jürgen Müller, Gabriele Taentzer, and Annika Wagner. Attributed Graph Transformations with Controlled Application of Rules. In G. Valiente and F. Rossello, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*, 1995. Technical Report B - 19, Universitat de les Illes Balears.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditionsl graph grammars - a constructive approach. In *Proc. of SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[ISO89] ISO. Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807, ISO, 1989.

[JR91] D. Janssens and G. Rozenberg. Structured transformations and computation graphs for actor grammars. In Ehrig et al. [EKR91], pages 446–460. Lecture Notes in Computer Science 532.

[KLG91] S.M. Kaplan, J.P. Loyall, and S.K. Goering. Specifying concurrent languages and systems with Δ-grammars. In Ehrig et al. [EKR91], pages 475–489. Lecture Notes in Computer Science 532.

[Koc96] Manuel Koch. Modellierung und Nachweis der Konsistenz von verteilten Transaktionsmodellen für Datenbanksysteme mit algebraischen Graphgrammatiken. Technical Report 96-36, TU Berlin, 1996.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In *Term Graph Rewriting - Theory and Practice*, pages 185–199. Wiley, 1993.

[Löw93] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, (109):181–224, 1993. Short version of Tech. Report 90/05, TU Berlin, 1990.

[Löw98] Michael Löwe. Ein formaler Ansatz zur Entwicklung langlebiger Softwaresysteme auf der Basis von Graphgrammatiken. Technical Report 98-4, TU Berlin, 1998.

[LR98] M. Llabrés and F. Rosselló. Pushout complements for arbitrary partial algebras. In G.Engels and G. Rozenberg, editors, *Proc. of 6th International Workshop on Theory and Apllication of Graph Transaformation*, number tr-ri-98-201, pages 80–87, 98.

[Man76] E. Manes. *Algebraic Theories.* Springer Verlag, 1976.

[Man94] Z. Manna. Beyond model checking. In *6th Int. Conference on Computer Aided Verification*, volume 818 of *LNCS*, pages 220–221. Springer, 1994.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In David Garlan, editor, *Proc. of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[MP92] Zohhar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer Verlag, 1992.

[MPR99] U. Montanari, M. Pistore, and F. Rossi. *[EKMR99]*, chapter Modeling Concurrent, Mobile and Coordinated Systems via Graph Transformations . World Scientific, 1999.

[MPW92] R.A. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i & ii. *Information and Computation*, 100(1):1–46 & 47 – 77, 1992.

[MS96] F. Moller and P. Stevens. The Edinburgh Concurrency Workbench user manual. 1996. http://www.dcs.ed.ac.uk/home/cwb.

[Nie97] Peter Niebert. *A Temporal Logic for the Specification and Verification of Distributed Behaviour.* PhD thesis, Universität Hildesheim, Fachbereich für Mathematik, Informatik und Naturwissenschaften, 1997.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[Pnu77] A. Pnueli. The Temporal Logic of programs. In *Proc. 18th Ann. IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. of the Fifth International Symposium in Programming*, 1981.

[Rat97] Rational software cooperation. Unified modeling language. available via http://www.rational.com, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[Rei84] Horst Reichel. *Structural Induction on Partial Algebras*, volume 18. Mathematical Research, Akademie-Verlag Berlin, 1984.

[Rei87] Wolfgang Reisig. Place/transition systems. In *Petri Nets : Central Models and Their Properties.*, volume 254 of *Lecture Notes in Computer Science*, pages 117 – 141. Springer Verlag, 1987.

[Rom96] D. Rombach, editor. *Proc. 18th International Conference on Software Engineering*, Los Alamitos, CA, 1996. IEEE.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations.* World Scientific, 1997.

[Rud97] M. Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Master's thesis, TU Berlin, 1997. http://tfs.cs.tu.berlin.de/publikationen/public1997.html.

[Sch92] Georg Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme.* PhD thesis, Universität Erlangen-Nürnberg, 1992. Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), 25/2.

[Sch93] Hans-Jürgen Schneider. On categorical graph grammars: Integrating structural transformations and operations on labels. *Theoretical Computer Science*, 109:257–274, 1993.

[Som96] Ian Sommerville, editor. *Proceedings of the 6th Int. Workshop on Software Configuration Management*, volume 1167 of *LNCS*, Berlin u. a., 1996. Springer.

[Sti92] Colin Stirling. Modal and Temporal Logics. In *Background: Computational structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 477–563. Clarendon Press, OXFORD, 1992.

[SW89] C.P. Stirling and D.J. Walker. Local model checking in the model mu-calculus. In *Proc. International Joint Conference on Theory and Practice of Software Development*, number 351 in LNCS, pages 369–382, 1989.

[SW90] C.P. Stirling and D.J. Walker. A general tableau technique for verifying temporal properties of concurrent programs. In *Proc. International BCS FACS Workshop on Semantics for Concurrency*, Workshops in Computing, pages 1–15, 1990.

[Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996. Shaker Verlag.

[Tae97] G. Taentzer. Parallel high-level replacement systems. *TCS*, (186), 1997.

[TFKV99] G. Taentzer, I. Fischer, M. Koch, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 3: Concurrency and Distribution.* World Scientific, 1999.

[Tic94] Walter Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software.* New York, 1994.

[vdHHW96] André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *[Rom96]*, pages 308–317, 1996.

[Vol97] Karsten Victor Volle. Verteilte Konfigurationsverwaltung: *COMAND*. Technical report, Basys GmbH, Am Weichselgarten 4, 91058 Erlangen, 1997.

[VW83] I. Vessey and R. Weber. Some factors affecting program repair maintenance – an empirical study. *CACM*, pages 128–134, Feb. 1983.

[Wag97] Annika Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras.* PhD thesis, TU Berlin, 1997.