Behavioral Congruences and Verification of Graph Transformation Systems with Applications to Model Refactoring

vorgelegt von
Diplom-Informatiker
Guilherme Salum Rangel

Fakultät IV – Informatik und Elektrotechnik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Uwe Nestmann
Berichter:	Prof. Dr. Hartmut Ehrig
Berichterin:	Prof. Dr. Barbara König

Tag der wissenschaftlichen Aussprache: 14.11.2008

Berlin 2008
D 83

Abstract

The concept of borrowed contexts has opened up graph transformations to the notion of an external "observer" where graphs (specifying systems) may interact with an environment in order to evolve. This leads to open systems in which a clear line delimits internal (non-observable) and external (observable) behavior. The observable interactions of a graph build up labeled transition systems such that bisimulations are automatically congruences, which means that whenever one graph is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This result turns out to be very useful for model refactoring, since one part of the model can be replaced by another bisimilar one.

The main goal of this thesis is twofold, namely to further develop the borrowed context framework and to explore its suitability as an instrument to reason about behavior preservation in model refactoring.

First we extend the borrowed context framework to handle rules with negative application conditions, which are often a required feature of nontrivial system specifications. That is, a rule may only be applied if certain patterns are absent in the vicinity of a left-hand side. This extension, which is carried out for adhesive categories, requires an enrichment of the transition labels which now do not only indicate the context that is provided by the observer, but also constrain further additional contexts that may (not) satisfy the negative application condition. We have shown that bisimilarity is still a congruence when rules have negative application conditions.

Experience shows that bisimulation proofs easily become tedious tasks and very prone to error when done by hand. In order to overcome this problem we have extended an existing on-the-fly bisimulation checking algorithm to the borrowed context setting and defined additional procedures to mechanize the verification of graphs for bisimilarity. This algorithm forms the core of a tool support which will enable us to come up with behavior analysis tools based on the borrowed context machinery.

Finally, techniques based on borrowed contexts are defined to check model refactorings for behavior preservation, which is always a crucial aspect of every refactoring transformation. One technique checks instances of a metamodel for bisimilarity w.r.t. to a set of productions defining the operational semantics of the metamodel. Bisimilarity implies preservation of behavior. A more elaborate technique shifts the behavior-preservation focus from instances of a metamodel to refactoring rules, where rules are checked for behavior preservation. One of the advantages of these techniques is that they are not tied up to specific metamodels, but rather can be applied to any metamodel whose operational semantics can be described by finite graph productions.

Zusammenfassung

Das Konzept von Borrowed-Contexts eröffnete für Graphtransformationen die Möglichkeit eines externen "Beobachters", wobei Graphen (als Spezifikationen von Systemen) mit einer Umgebung kommunizieren können, um sich zu entwickeln. Dies führt zu offenen Systemen, in denen internes (nicht beobachtbares) und externes (beobachtbares) Verhalten unterschieden werden. Die beobachtbaren Interaktionen bilden ein Transitionssystem derart, dass Bisimilaritäten automatisch Kongruenzen sind. Das heißt, wenn ein Graph bisimilar zu einem anderen ist, können beide Graphen im Kontext eines größeren Graphen ohne Auswirkung auf das beobachtbare Verhalten ausgetauscht werden. Dieses Ergebnis ist sehr nützlich für Modell-Refactoring, weil ein Teil des Modells durch ein bisimilares Teil ersetzt werden kann.

Das Hauptziel dieser Arbeit besteht darin, den Ansatz von Graphtransformationen mit Borrowed-Contexts weiter zu entwickeln und auch seine Eignung zur Analyse der Bewahrung des Verhaltens im Anwendungsgebiet Modell-Refactoring zu erforschen.

Erstens erweitern wir den Borrowed-Context Ansatz auf Regeln mit negativen Anwendungsbedingungen, die oft in komplexen Spezifikationen verwendet werden. Dies bedeutet, dass eine Regel nur angewendet werden darf, wenn bestimmte Muster außerhalb einer linken Seite abwesend sind. Diese Erweiterung, die im Rahmen von Adhesiven Kategorien durchgeführt wird, erfordert auch eine Erweiterung der Label des Transitionssystems um negative Anwendungsbedingungen. Als wichtiges Ergebnis zeigen wir, dass die Bisimilarität immer noch eine Kongruenz ist, wenn Regeln negative Anwendungsbedingungen haben.

Die Erfahrung zeigt, dass Bisimulationsbeweise langwierig und sehr fehleranfällig werden, wenn sie von Hand durchgeführt werden. Um dieses Problem zu lösen, haben wir einen bestehenden "on-the-fly" Bisimulations-Algorithmus um zusätzliche Prozeduren zur Mechanisierung der Überprüfung der Bisimilarität von Graphen im Rahmen des Borrowed Context Ansatzes definiert. Dieser Algorithmus bildet den Kern einer Werkzeugunterstützung für die Entwicklung von Techniken zur Verhaltensanalyse basierend auf Graphtransformationen mit Borrowed Contexts.

Darauf aufbauend werden Techniken definiert, um die Verhaltensbewahrung im Anwendungsbereich Modell-Refactoring zu untersuchen. Verhaltensbewahrung ist immer ein wesentlicher Aspekt jeder Refactoring-Transformation. Eine Technik überprüft die Bisimilarität von Instanzen eines Metamodells in Bezug auf Regeln, die die operationelle Semantik des Metamodells beschreiben, wobei Bisimilarität Verhaltensbewahrung impliziert. Eine zweite Technik bezieht sich auf Verhaltensbewahrung von Refactoring-Regeln. Einer der Vorteile dieser Techniken ist, dass sie nicht an spezifische Metamodelle gebunden sind, sondern auf jedes Metamodell angewendet werden können, deren operationelle Semantik durch endliche Graphregeln beschrieben werden kann.

Acknowledgments

Completing a PhD is truly a long journey which I would not have been able to complete without the aid and support of many people. First of all, I must express my gratitude towards my supervisor, Prof. Hartmut Ehrig, who gave me not only the opportunity but also freedom to pursue my own research interests. His support, attention to detail, scholarship and hard work have set an example I hope to match some day.

I am also greatly indebted to Prof. Barbara König. She spent a lot of time with our discussions over the past years. Her background, ideas and tremendous support had a major influence on this thesis. From her I learned that *playing* with several perspectives may have a huge impact on finding sensible solutions for problems. Her hard work has showed me that there is always something left to be done in order to improve our work.

Over the years, I have enjoyed the aid of DAAD (German Academic Exchange Service) which has supported me while I completed my PhD. Special thanks go to Ms. Roswitha Paul Walz for the outstanding service and attention offered by her and her team for the international students at TU Berlin.

I would like to thank Prof. Daltro Nunes, from whom I learned how research should be done and how to present our ideas in a clear way. Within the context of his research group at UFRGS I met Heribert Schlebbe, whom I worked with during a research stay in Stuttgart. I had such a great time working with Heribert that I ended up deciding to pursue my PhD in Germany. Thanks Heribert!

I owe a lot to Leen Lambers and Tobias Heindel for all their support on category theory. I will never forget Tobias bringing me a pile of books from the library. Thank you both for the many helpful discussions. Thanks Paolo Baldan, Gabi Taentzer and Claudia Ermel for several interesting discussions on model refactoring. Thanks also go to my fellow TFS colleges who helped me out in one way or another during this long journey, specially my office mates Enrico Biermann and Olga Runge for the nice working environment. Thanks go to Käte Schlicht and Margit Russ for their help in any situation.

I thank my family, specially Susana, Nelson, Susaninha and Juliana, for instilling in me confidence and a drive for pursuing my PhD. And for the one who has made the many hours spent in this work seem worthwhile after all, Paula Silva. Thanks for enjoying life together with me ;-)

Last but not least, I wish to thank my friends who have always been present and supported me: Rita Krauser and her wonderful family, Geraldo, Ivan, Gabi, Julia, Eliecer, Galina, Juan and Carlos.

Contents

Chapter 1

Introduction

1.1 Motivation

Engineering is a practical application of science in order to fulfill the needs of society. Its main goal is to produce high quality products at the lowest possible cost. Despite many advances in the last years Software Engineering still lacks the rigor associated with other engineering areas which enable them to build quality products effectively. While most engineering projects do not fail, 70% of software projects fail in some way [Kik05].

The development of large software systems is always a challenging task [Gib94]. What makes quality software hard to build is its intrinsic essence which is mainly governed by *freedom* of choice. Unlike other branches of Engineering, Software Engineering is not governed by physical laws. Whenever a bridge has to be built construction engineers must unconditionally obey physical laws, which are models of the real world in terms of mathematics. Therefore, a bridge is constructed within many physical world constraints. Physical laws and mathematics form a framework within which engineers can test their own designs. Construction engineers design a particular solution to a given problem and physical statements can be tested, i.e., they can basically "ask" if a certain solution will work without actually building it.

On the other hand, software development is an engineering discipline which builds on mathematics itself. No physical laws are applicable to software. Thus, software development is inherently complex due to the immense degree of freedom engineers have in their hands to come up with software solutions. Moreover, software is composed of thousands of unique (and often difficult to analyze) parts rather than repeated well-known parts as in other engineering disciplines.

The software development life cycle begins with user requirements elicitation and progresses into the build, test, and acceptance phases. Experience has shown that requirement errors detected at late stages of the development process lead to a dramatic increase of the software costs [Wes02]. Hence, the most important work in software development happens before a single line of code is written. In his essay "No Silver Bullet" [Bro86], Frederick Brooks wrote:

> *"The essence of a software entity is a construct of interlocking concepts... I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."*

A very useful tool to be applied in the development of complex software systems is formal methods, which makes possible the specification (description) of software in terms of mathematical entities. Formal methods equip software engineers with the necessary means to reason about critical aspects of software designs without having to build them first. While simulation and testing explore some of the possible behaviors of the system, formal verification, which is based upon formal methods, conducts exhaustive exploration of all possible behaviors of the system. Formal verification consists in proving the correctness of a design with respect to mathematical properties. The main approaches to formal verification are:

- *Model checking* [HR04, JGP99] can be seen as an extension of testing, where given a description of a system and a desired property of the system expressed as a formula in some temporal logic, then automated tools check whether the system satisfies the desired property;

- *Theorem proving* [New01] is a formalization of mathematics in which logic is applied to characterize mathematical reasoning, and automated formal support is used to aid the creation and checking of proofs;

- *Equivalence checking* [HC98] is a verification method to prove that two representations of a system, e.g. the specification and its implementation, exhibit exactly the same behavior w.r.t. some notion of equivalence.

Our focus in this thesis is the equivalence-checking approach, which originally stems from the field of process calculi, where the foundations of concurrent and mobile processes are investigated. Process calculi are designed in order to express some fundamental aspect of computation, and then research is done to investigate the calculus' behavioral theory and its expressive power. Examples of process calculi are: CCS [Mil89], π-calculus [MP92, MPW92, SW01], Ambients [CG98], Fusion [PV98], Join [FG96] and Spi [AG97], just to cite a few. The underlying theory of such calculi is often complicated, perhaps because of the various design decisions involved in their definitions. Process calculi are in essence textual, and therefore must be equipped with a structural congruence $\equiv$ to determine processes that are equal even though they are not syntactically identical. In CCS, for example, $P|Q$ and $Q|P$ should represent the same process.

Notions of behavioral equivalence are of fundamental importance to compare processes. A plethora of process equivalences has been proposed and studied in the literature, ranging from the coarsest (trace equivalence) to the finest (bisimulation). The latter is the most widespread notion of behavioral equivalence and makes less identifications than any of the others. In [Gla01] van Glabbeek shows a lattice containing several equivalences notions ordered by their distinguishing power. In this thesis we are mainly interested in bisimulation.

Congruence is a very desirable property a behavioral equivalence may have. It allows one to replace a subsystem with an equivalent one without changing the behavior of the overall system, and furthermore helps to make bisimilarity proofs modular. However, proving that an equivalence is a congruence is by far not an easy task.

A behavioral equivalence can be defined on either reactions rules (also called unlabeled transitions) or labeled transitions. The main advantage of reaction rules is that it is often relatively easy to justify their correctness and appropriateness as notions of equivalence. The main problem is that bisimilarity defined on unlabeled reduction rules is in general not a congruence. Previous solutions have been to either require that two processes are related if and only if they are bisimilar under all possible contexts [MS92] or to derive a labeled transition system manually. The first solution needs quantification over all possible contexts, so proofs of bisimilarity may quickly become very complicated. In the second solution, proofs tend to be much easier, but it is still necessary to show that the labeled transition system is equivalent to the unlabeled variant.

The idea which was formulated in the papers of Sewell [Sew98, Sew02], Leifer/Milner [Lei01, LM00] (*relative pushouts*), Sassone/Sobociński [SS03a] (*groupoidal relative pushouts*) and Ehrig/König [EK04, EK06] (*borrowed contexts* for graph rewriting) is to automatically derive a labeled transition system (from unlabeled rules) such that the resulting bisimilarity is a congruence. A central concept of this approach is to formalize the notion of minimal context which enables a process to reduce. The first three techniques were developed in the setting of process calculi, whereas the borrowed context technique was originally defined for graph rewriting.

Graphs, due to their simple but yet powerful visual notation, are a natural way to explain complex situations on an intuitive level. Graph transformation [Roz97, EEKR99, EKMR99, EEPT06] is concerned with the rule-based modification of graphs according to graph transformation rules. Graph transformations are useful to define the operational semantics of visual models in analogy to the operational semantics of programming languages defined by term rewriting systems [SPvE93]. Among the formal approaches to graph rewriting, the *double pushout* (DPO) can be considered a standard due to its large amount of theoretical results and applications [Roz97, EEKR99, EKMR99, EEPT06] in several branches of computing, such as programming, specification, concurrency, distribution, visual modeling and model transformation. Furthermore, its implementation called AGG [AGG] provides the practical means to execute and reason about systems defined as graph transformations.

In the recent years an area which has consistently profited from the DPO approach to graph rewriting is model transformation [MG06]. Model transformation concerns the automatic generation of models from other models according to a transformation definition, which describes how a model in the source language can be transformed into a model in the target language. Graph transformation systems (GTS) are well-suited to model not only model transformation but also model refactoring, where source and target languages are the same. A GTS specifies model transformation by defining graph transformation rules to translate one model into another. A crucial question that must be asked is whether a given refactoring (or model transformation) is behavior-preserving, which means that transforming one model into another model does not change the original behavior. In practice, the proof of behavior-preserving transformations is not an easy task, and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved.

In this thesis we focus on the DPO approach to graph transformation and its extension to borrowed context for behavioral analysis of graph systems. The main advantages of the borrowed context (BC) technique over the others based on relative pushouts are:

- Process calculi with complex structural equivalences often do not yield the intended results with Leifer/Milner's relative pushouts. For these cases, groupoidal relative pushouts are more suitable but at the cost of a more complex underlying theory. Whenever a process calculus is encoded as graphs with interfaces in the BC framework, these issues on structural equivalence are automatically handled by graph isomorphisms. Many process calculi such as π-calculus [MP92, MPW92, SW01] and ambient calculus [CG98] can be translated into graphs and analyzed via borrowed contexts;

- Many complex data structures and also specification languages can be easily understood by humans via graph-based notations;

- BC machinery to label derivation is based on very simple categorical concepts, namely pushouts and pullbacks. The relative-pushout techniques per se and their underlying categorical constructions are by far more complex;

- Experience shows that label derivation and bisimulation proofs easily become tedious tasks and very prone to error when done by hand. Therefore, a tool support is of fundamental importance. Due to its simple constructions the BC technique lends itself better to mechanize these tasks.

We believe that the borrowed context technique can be a useful instrument to reason about the behavior of a wide variety of systems. This thesis proposes new extensions to the BC framework that will help the development of novel behavioral analysis techniques.

1.2 Aims of the Thesis

The main objective of this thesis is twofold:

1. further develop the borrowed context framework;
2. explore the suitability of borrowed contexts in form of analysis techniques to reason about behavior preservation in model refactoring.

In early stages of a software development the key concepts of the software are usually investigated in detail in order to ensure that the future system will work as expected. Describing software systems as graphs turns out to help bridge the gap between engineers and the future users of the system (also known as stakeholders). Due to the intuitive and expressive notation of graphs, system designs given by graphs are easily understood even by non-experts in computing, such as the stakeholders, who can understand the design and hence be more proactive during requirements elicitation.

The DPO approach to graph rewriting already possesses a wide amount of results. Nonetheless, it is well-known that beyond theoretical results, expressive power and ease-of-use of the notation another fundamental feature of a formal technique is tool support. Practitioners of graph transformations may design, execute and reason about properties of their systems with AGG [AGG], which is a graph transformation engine equipped with a graphical interface developed at TU Berlin.

For the behavioral analysis of systems the borrowed context machinery offers an uncomplicated way of deriving transition labels which not only smoothly extends the DPO approach but also has a very constructive nature. Compared to other approaches (discussed in Chapter 2), where the derivation of labels is a somewhat complex task, the borrowed context technique is rather straightforward and simple. However, from applying the BC technique to several examples we have learned that the derivation of labels can be often very time consuming. Therefore, the development of algorithms to automatize the derivation of labels and bisimulation proofs is required in order to implement a tool. Bisimulation checking is in general undecidable which makes such mechanized proofs be in general very difficult. In this thesis, though, we show several interesting examples where bisimulation can indeed be checked automatically.

In contrast to standard DPO to graph rewriting, the borrowed context framework has not yet been applied to a wide range of examples. In this thesis we also aim at the development of several examples in order to demonstrate the suitability of borrowed contexts as a behavior analysis tool. We also develop techniques based on the BC machinery to reason about behavior preservation in model refactoring.

Last, but not least, the borrowed context framework defined in [EK04, EK06] handles graph transformation rules with no application condition other than the gluing condition. Even though the generative power of the DPO approach is sufficient to generate any recursively enumerable set of graphs, very often extra application conditions are a required feature of nontrivial specifications. Negative application conditions (NACs) [HHT96] for a graph production are conditions such as the nonexistence of nodes, edges, or certain subgraphs in the graph G being rewritten, as well as embedding restrictions concerning the match $L \to G$.

Summarizing, the concrete aims of this thesis are to:

- investigate the viability of using rules with negative application conditions in the borrowed context framework;
- determine whether bisimilarity for rules with NACs remains a congruence and under which conditions;
- define up-to techniques to reduce the size of relations needed to define a bisimulation;
- tailor an existing bisimulation checking algorithm to borrowed contexts;
- define the necessary algorithms for the development of a tool support for bisimulation checking;
- investigate and develop techniques based on the BC machinery to reason about behavior preservation in model refactoring;
- demonstrate the suitability of the BC technique as a tool to analyze behavior preservation via several examples.

1.3 Main Results

The main results achieved in this thesis are summarized as follows:

Borrowed Contexts with NACs: we investigated negative application conditions in transformations rules and their consequences to the bisimilarity result. We discussed which problems arise due to the introduction of NACs and how they can be overcome in order to guarantee that the derived bisimilarities are congruences. The extension, which is carried out for adhesive categories [LS04], requires an enrichment of the transition labels which now do not only indicate the context that is provided by the observer, but also constrain further additional contexts that may (not) satisfy the negative application condition. That is, we do not only specify what must be borrowed, but also what should not be borrowed. We prove that the main result of [EK06] (bisimilarity is a congruence) still holds for our extension. Moreover, as a straightforward consequence of a technique based on initial pushouts proposed in [BGK06a] we define the notion of gluing condition for borrowed context rewriting. Two examples are given to illustrate the theory: a simple example based on processing tasks on servers and a more elaborate one in terms of blade server systems.

Up-to Techniques: up-to techniques [San95] relieve the onerous task of bisimulation proofs by reducing the size of the relation needed to define a bisimulation. They also provide the means to check bisimilarity with finite up-to relations in some cases where any bisimulation is infinite. In this thesis we define three up-to techniques for the BC setting with NACs, namely bisimulation up to isomorphism, context and bisimilarity. Because graphs in the DPO approach are defined up to isomorphism the up-to isomorphism technique can be considered the minimal requirement to enable bisimulation checks in the borrowed context framework. Even though this technique turns out to be subsumed by the more powerful up-to context technique it is still very useful in practice since it can be computed much faster with help of graph certificates [Ren06], which are "signatures" that identify certain properties in graphs such as the number of nodes and edges with their respective labels.

Algorithms for Borrowed Contexts: we have defined algorithms which pave the way to the development of a tool to check graphs for bisimilarity. The main algorithms are:

- **partial match finding**: this algorithm finds the partial matches between a graph with interface and left-hand sides of graph rules which lead to borrowed context steps;

- **label matching**: this procedure performs the label matching required by the bisimulation game;
- **up-to techniques**: procedures to decide whether a pair of graphs is contained in relations defining bisimulation up to isomorphism and also up to context;
- **bisimulation checking procedure**: we extended Hirschkoff's on-the-fly bisimulation checking algorithm [Hir01] to the borrowed context setting and provided it with several additional details for the manipulation of the required data structures;

Apart from the algorithms above we have outlined how to make the bisimulation checking procedure more efficient with use of graph certificates to alleviate the burden caused by isomorphism checks. We have also implemented a prototype in a functional programming language called Objective Caml [OCa] for validation purposes. This prototype covers the entire process of label derivation, whereas the complete implementation of the algorithms above is part of our future work.

Behavior Preservation in Model Refactoring: a fundamental question in every refactoring is whether the transformations preserve behavior, i.e., do not change the observable behavior of a model. We define techniques based on the borrowed context machinery to reason about behavior preservation in model refactoring. The first technique allows checking instances of a metamodel for bisimilarity w.r.t. a set of productions defining the operational semantics of the metamodel. Bisimilarity implies behavior preservation. The second, more elaborate, technique exploits the fact that observational equivalence is a congruence and hence we show how to check refactoring rules for behavior preservation. When rules are behavior-preserving, their application will never change behavior, i.e., every model and its refactored version will have the same behavior. However, often there are refactoring rules describing intermediate steps of the transformation, which are not behavior-preserving, although the full refactoring does preserve the behavior. For these cases we present a procedure to combine refactoring rules to behavior-preserving concurrent productions in order to ensure behavior preservation. These techniques are applied to examples of minimization of deterministic finite automata and flattening of hierarchical statecharts. We believe that such a method will help the user gain a better understanding of the refactoring rules since he or she can be told exactly which rules may modify the behavior during a transformation. One of the main advantages of defining behavioral analysis techniques based on borrowed contexts is that they are promptly available for every metamodel whose operational semantics can be specified in terms of finite graph transformation productions.

Further Examples: we have defined many mid-size examples in this thesis, namely blade server systems (to illustrate the use of NACs), two refactorings of deterministic finite automata (minimization of automata by merging equivalent states and deletion of unreachable states) and flattening of hierarchical statecharts.

1.4 Overview of the Chapters

This thesis is structured as follows:

Chapter 2 (Deriving Bisimulation Congruences) gives an overview about the issues emerged in the field of process calculi which led to the idea of deriving labeled transition systems from reaction rules in such a way that the resulting behavioral equivalence is automatically a congruence. Three techniques to label derivation are discussed: Sewell's seminal work, Leifer and Milner's relative pushouts (RPOs) and then Sassone and Sobociński's GRPOs. The double-pushout (DPO) approach to graph transformations is recalled as well as its extension to borrowed contexts, which is a natural way to label derivation in the DPO approach and the basis upon which this thesis is developed.

Chapter 3 (Deriving Bisimulation Congruences in the Presence of NACs) presents an important and useful extension of the borrowed context framework to transformation rules with negative application conditions (NACs). That is, a rule may only be applied if certain patterns are absent in the vicinity of a left-hand side. This extension is carried out in the setting of adhesive categories, and therefore it is also available for other adhesive structures than graphs. The main result of this chapter consists in the proof that the bisimilarity for rules with NACs remains a congruence. The theoretical notions and problems due to NACs are illustrated throughout this chapter by a simple example which is intuitive and easy to understand. We also develop up-to techniques to handle NACs and define the gluing condition for borrowed context rewriting. A more elaborate example in terms of blade server systems is additionally given to illustrate the theory.

Chapter 4 (Bisimulation Verification) algorithmically describes how graphs can be checked for bisimilarity using the borrowed context setting. More specifically, we extend Hirschkoff's on-the-fly algorithm for bisimulation checking, enabling it to verify whether two graphs are bisimilar with respect to a given set of productions (possibly with NACs). We then apply this framework to refactoring problems, where we check instances of a model and their refactored versions for bisimilarity. Two examples are given: the minimization of deterministic finite automata and the flattening of hierarchical statecharts.

Chapter 5 (Behavior Preservation in Model Refactoring) presents novel techniques to reason about behavior preservation in model refactorings. Behavior preservation, namely the fact that the behavior of a model is not altered by the transformations, is a crucial property in refactoring. The most common approaches to behavior preservation rely basically on checking given models and their refactored versions, as we showed in Chapter 4 using the borrowed context technique. In this chapter we move up the abstraction ladder and introduce a more general technique for checking behavior preservation of refactorings defined by graph transformation rules. Exploiting the fact that observational equivalence is a congruence, we show how to check refactoring rules for behavior preservation. An example of refactoring for finite automata is given to illustrate the theory. Moreover, we also apply this technique to the statecharts example of Chapter 4.

Chapter 6 (Towards a Tool Support) describes additional algorithms required by the bisimulation checking procedure of Chapter 4.

Chapter 7 (Conclusion) summarizes the main achievements of this thesis and outlines open problems and directions for future work.

Appendices briefly recall some basic categorical concepts required in this thesis. Furthermore, all proofs of Chapter 3 and additional information about the extension to NACs can be found here.

1.5 How to Read the Thesis

Finally, we give some suggestions for reading this thesis. The main chapters, namely 3, 4 and 5, are self-contained and can be read independently. The only exception is the statecharts example of Chapter 5 which requires jumping back to its counterpart in Chapter 4.

We have in mind three main kinds of readers. Those who are mostly interested in behavior equivalences theory may profit reading Chapter 3 on the extension to rules with negative application conditions and also Chapter 5 for an interesting application of the borrowed context machinery to model refactorings.

For model-refactoring oriented readers the first sections of Chapter 5 give an overall idea of how to use the borrowed context technique for behavior-preservation purposes in model refactoring. Then we suggest the refactoring examples (finite automata and statecharts) of Chapter 4 and finally, the second part of Chapter 5, where the technique to check refactoring rules for behavior preservation is presented and applied to the finite automata and statecharts examples.

Chapters 4 and 6 are for those who are interested in implementing a tool for bisimulation checking.

Chapter 2

Deriving Bisimulation Congruences

2.1 Motivation

Process calculi have been developed to describe and analyze concurrent systems. A process calculus is a concise pseudo-programming language which focuses on small language features for concurrent aspects, such as non-determinism, synchronization, and communication. The goal is to concentrate on a few basic principles and reasoning techniques which provide the means to study and investigate the concurrent phenomena of systems without the additional burden (e.g. syntactic sugar) of a full programming language.

The syntax of a process calculus is based upon a language defining a small set of operators and a few syntactic rules for constructing larger processes from simpler components. Additionally, a process calculus may be equipped with unlabeled transitions (also called reaction rules), labeled transitions and notions of equivalences.

Unlabeled transitions specify the internal state changes of a process by focusing on interactions between different parts. They do not require any form of interaction with the environment, and therefore correspond more closely to the executions conceived by the calculus designer's. An example of a reaction rule in CCS is $a.P \to P$, where the process performs a and becomes P. On the other hand, labeled transitions characterize the state changes of a process by interacting with the environment. Each transition has a label specifying the exact interaction with the environment that enables the transition. Hence, a label can be seen as an observation about the behavior of a process. An example of labeled transition in CCS is $a.P \xrightarrow{a} P$, where a can be observed. Labeled transitions describe observable behaviors in a compositional way and their labels are often not so intuitive in operational terms.

Another ingredient a process calculus may possess consists of notions of preorders and equivalences to determine when two processes have the same behavior. The perspective usually assumed by a behavioral equivalence is that an external observer compares two open processes by interacting with them over a restricted interface. The processes are considered to be equivalent when they cannot be distinguished by the observer. Behavioral equivalences can be classified with respect to the power that is given to the observer. In Figure 2.1 we show van Glabbeek's lattice [Gla01] containing several equivalences notions ordered by their distinguishing power.

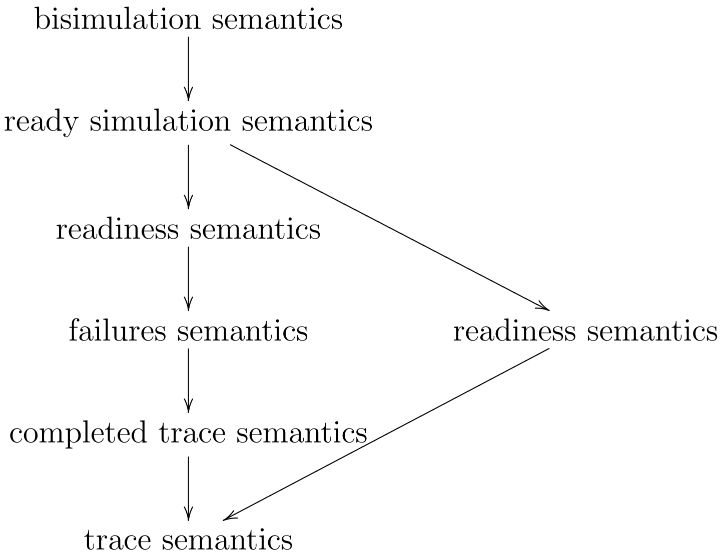


Figure 2.1: The linear time – branching time spectrum.

The behavioral notions in Figure 2.1 range from linear time to branching time equivalences. In the former, a process is determined by its possible executions, whereas in the latter the branching structure of processes is also taken into account. In other words, a linear time equivalence concerns a single computation path, whereas a branching time equivalence considers the paths that are possible. Trace semantics is the coarsest equivalence and makes the most identifications. The finest equivalence is bisimulation, which makes less identifications than any of the others.

Congruence is an important desirable property a behavioral equivalence may have. It allows us to replace a subsystem with an equivalent one without changing the behavior of the overall system and furthermore helps to make bisimilarity proofs modular. The same intuition also holds for preorders. However, proving that a preorder or equivalence is a congruence is by far not a trivial task.

An equivalence semantics can be defined on either reactions rules or labeled transitions. However, reaction rules may lack compositionality which usually leads to complex semantic theories. In these cases a bisimulation congruence based on labeled transition systems may provide useful proof techniques: bisimulations have the benefits inherited from coinduction and the closure properties of congruences allow compositional reasoning.

For complex process calculi reaction rules are usually easier to define and more intuitive than labeled transitions. Though the main question is: given a set of reaction rules is it possible to automatically derive a labeled transitions system such that the resulting bisimilarity is a congruence? In the following sections we briefly review three answers to this question. Thereafter we recall the DPO approach to graph rewriting in Section 2.3 and its extension to borrowed contexts in Section 2.3.2.

2.2 Deriving Bisimulation Congruences

Originally, the operational semantics of simple process calculi (e.g. CCS) was given by labeled transition systems, where labels describe possible interactions with the environment, and which forms the basis for the definition of behavioral equivalences. However, for a more complex calculus, even though its semantics is well understood it may be difficult to define its labeled transition system (LTS). For example, the π-calculus [MS92] has two alternative LTS, the early and the late version, each giving a different bisimulation equivalence.

For complex calculi (e.g. ambient calculus [CG98]) it is usually easier and more natural to define their operational semantics in terms of unlabeled transitions which specifies the possible reductions of a process without considering the environment. The main problem is that a bisimulation defined on unlabeled reduction rules is usually not a congruence, that is, it is not closed under the operators of the process calculus. The first solutions to tackle this problem have been to either require that two processes are related if and only if they are bisimilar under all possible contexts [MS92] or to derive a labeled transition system by hand. The former needs quantification over all possible contexts, and hence proofs of bisimilarity can become very complex. In the latter proofs are usually easier, but it is still necessary to prove that the labeled transition system is equivalent to the unlabeled variant, which is often an ad hoc task for each calculus.

So the idea formulated in the papers of Sewell [Sew98, Sew02], Leifer/Milner [Lei01, LM00], Sassone/Sobociński [SS03a] and Ehrig/König [EK04, EK06] is to automatically derive a labeled transition system (from unlabeled rules) such that the resulting bisimilarity is a congruence. A key concept of this approach is the formalization of a minimal context which enables a process to reduce. For example, given the CCS process $a.P$, it reduces when inserted into the contexts $\_ \mid \bar{a}.Q$ and $\_ \mid \bar{a}.Q \mid b.R$. However, the second context provides too much information which is not required to trigger the reduction. Hence, the first context is in some sense more adequate and then yields the labeled transition

$$a.P \xrightarrow{- \mid \bar{a}.Q} P \mid Q$$

saying that $a.P$ inserted into this context reacts and reduces to $P \mid Q$.

2.2.1 Sewell's Dissection Lemmas

In his seminal work [Sew98, Sew02] Sewell's goal was to derive a labeled transition system directly from reaction rules such that useful LTS based equivalences, including bisimilarity, are automatically congruences. He proposed several ways of doing this for restricted classes of term rewriting systems. The fundamental idea is that terms give rise to labeled transitions, where each label is a context which allows the term to react. In other words, whenever a term receives from the environment a particular context then the rewrite of the term inside this context should be possible in the underlying rewriting semantics. Additionally, these labels should be the smallest contexts that trigger the rewriting of a particular term. The notion of "smallest" was later on elegantly expressed in categorical terms by Leifer and Milner.

Sewell defined a series of dissection lemmas to analyze a term's structure and determine the missing triggers, if any. The proofs that bisimulation is a congruence on the resulting LTS is simple in the case of free syntax. However, they easily become very complicated for non-trivial structural congruences. Already for rules defining parallel composition Sewell's method becomes quite complex.

2.2.2 Leifer and Milner's Relative Pushouts

A generalized approach to Sewell's method was developed by Leifer and Milner [LM00], where the notion of smallest context is formalized as the categorical concept of relative pushout.

Leifer and Milner consider categories in which arrows are terms and composition is substitution. In such a framework a process a evolves to a' (depicted as $a \to a'$) when inserted into a context F which provides the missing parts to trigger a reaction rule (l, r), as shown in the commuting square below. In this case there also exists an arrow D such that $a = Dl$ and $a' = Dr$.

$$\begin{tikzcd} {} \arrow[r, "a"] \arrow[d, "l"'] & {} \arrow[d, "F"] \\ {} \arrow[r, "D"'] & {} \end{tikzcd}$$

Hence, given a process a and the left side l of a reaction rule (l, r) the idea is to find arrows D and F such that D, F are the minimal arrows that make the diagram above commute, i.e., $Dl = Fa$. In this case a labeled transition $a \xrightarrow{F} a'$ is derived. The notion of minimal context is captured by relative pushouts as in Definition 2.2.1, where h_1, h_0 in Diagram (2.2) are minimal with respect to the original square in Diagram (2.1).

Definition 2.2.1 (Relative Pushout). *In any category* $\mathbf{C}$*, consider a commuting square as shown in Diagram* (2.1) *such that* $g_0; f_0 = g_1; f_1$*. A* relative pushout *for this commuting square is a triple* h_0, h_1, h *satisfying the following two properties:*

(i) *commutation:* $h_0; f_0 = h_1; f_1$ *and* $h; h_i = g_i$ *for* $i = 0, 1$ *(see Diagram* (2.2)*);*

(ii) *universality: for any* h'_0, h'_1, h' *satisfying* $h'_0; f_0 = h'_1; f_1$ *and* $h'; h'_i = g_i$ *for* $i = 0, 1$*, there exists a unique mediating arrow* k *such that* $h'; k = h$ *and* $k; h_i = h'_i$ *(see Diagram* (2.3)*).*

$$\begin{tikzcd}
{} \arrow[r, "f_0"] \arrow[d, "f_1"'] & {} \arrow[d, "g_0"] \\
{} \arrow[r, "g_1"'] & {}
\end{tikzcd} \qquad (2.1)$$

$$\begin{tikzcd}
{} \arrow[rr, "f_0"] \arrow[dd, "f_1"'] & & {} \arrow[dd, "g_0"] \arrow[dl, "h_0"'] \\
& {} \arrow[dr, "h"] & \\
{} \arrow[rr, "g_1"'] \arrow[ur, "h_1"] & & {}
\end{tikzcd} \qquad (2.2)$$

$$\begin{tikzcd}
{} \arrow[rrr, "f_0"] \arrow[dd, "f_1"'] & & & {} \arrow[dd, "g_0"] \arrow[dll, "h_0"'] \arrow[dl, "h'_0"] \\
& {} \arrow[drr, "h"] & {} \arrow[l, dotted, "k"'] \arrow[dr, "h'"] & \\
{} \arrow[rrr, "g_1"'] \arrow[ur, "h_1"] \arrow[urr, "h'_1"'] & & & {}
\end{tikzcd} \qquad (2.3)$$

Leifer and Milner [LM00] showed that labeled transition systems with labels obtained via the relative pushout technique lead to observational equivalences that are congruences. They showed in [LM00, Lei01] that bisimilarity, trace and failures equivalences are congruences.

One of the greatest advantages of the relative pushout technique is that its underlying theory is defined in terms of categorical concepts. By using category theory constructions and proofs are performed on an abstract level and so can be “reused” across a variety of models, where each model forms a category possessing relative pushouts.

2.2.3 Sassone and Sobociński's G-Relative Pushouts

An important feature a process calculus may be equipped with is some notion of structural congruence $\equiv$ to determine equivalence of processes that are not syntactically identical.

Sassone and Sobociński observed that the technique to label derivation based on relative pushouts fails for process calculi with even simple structural congruences [SS03b]. In Leifer and Milner's approach if arrows are quotiented by a structural congruence then too much information is lost and the derivation of labels via relative pushouts does no longer yield the expected results. Sassone and Sobociński depict this problem by showing a very simple calculus with an associative and commutative parallel operator [SS03b]. In this example the relative pushout technique is not able to "remember" the place within a term where a reaction takes place, which is an essential information to the derivation of a sensible labeled transition system.

A similar problem also occurs with algebraic structures such as action graphs [Mil96] and bigraphs [Mil01]. For these structures due to the problem of locating reactions, sufficient relative pushouts do not exist [Lei01, Mil01]. Sewell [Sew02] tackles this problem for syntactic terms by using a notion of coloring, as Leifer [Lei02] proposes an abstract approach by adding support to the category (via precategories).

Sassone and Sobociński's approach extends the relative-pushout technique to groupoidal relative-pushouts (**G**-RPO for short), which are basically relative pushouts in 2-categories. This richer underlying category makes possible keep track of the application of structural congruence rules (as 2-cells). Hence, more sensible labeled transition systems can be obtained via **G**-RPOs than by applying other approaches which forget where the reactions take place.

Sassone and Sobociński also lifted the **G**-RPO technique to a class of cospan bicategories over adhesive categories, which allows the derivation of labeled transition systems for every model based on such bicategories. Interesting examples of cospan categories are Milner's bigraphs [Mil01] and the extension of the double-pushout approach to borrowed contexts [EK04, EK06].

2.3 Graph Transformations

Graph transformation is concerned with the rule-based modification of graphs according to graph transformation rules. Graph grammars, which consist of graph rules and a start graph, are very useful to generate graph languages by Chomsky grammars in formal language theory. Furthermore, graphs can be used to model states of systems and graph transformations to describe state changes of these systems. Especially, graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation.

Best engineering practices have distinctly demonstrated that visual notations take a huge advantage over textual descriptions as they are more succinct and easily understood by humans. Graphs are a natural way to explain complex situations on an intuitive level. For this reason graphs are widely used almost everywhere in computer science, e.g. as data and control flow diagrams, entity relationship and UML diagrams, Petri nets, visualization of software and hardware architectures, evolution diagrams of nondeterministic processes, SADT diagrams and many more.

The research area of graph transformations dates back to the early seventies and since then many approaches, methods, techniques and results have been developed. In volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97] one can find a detailed presentation of different graph transformation approaches. A state-of-the-art report for applications, languages and tools for graph transformation as well as results for concurrency, parallelism and distribution can be found in volumes 2 [EEKR99] and 3 [EKMR99].

A graph transformation rule (also called production) $p = (L, R)$ is a pair of graphs (L, R), called left-hand side L and right-hand side R. A graph transformation occurs when for a given source graph G and a graph rule p we find a match of L in G which leads to the replacement of L by R in the graph G giving rise to the target graph H. This notion of transformation usually leads to technical problems on how to connect R with the context in the target graph, which gives rise to many approaches to handle them. The graph transformation approaches are summarized below.

1. The *node label replacement approach*, mainly developed by Rozenberg, Engelfriet and Janssens, allows a single node, as the left-hand side L, to be replaced by an arbitrary graph R. The connection of R with the context is determined by an embedding relation depending on node labels.

2. The *hyperedge replacement approach*, mainly developed by Habel, Kreowski and Drewes, has as left-hand side L a labeled hyperedge, which is replaced by an arbitrary hypergraph R with designated attachment nodes corresponding to the nodes of L. The gluing of R with the context at the corresponding attachment nodes leads to the target graph without using an additional embedding relation.

3. The *algebraic approach* is based on pushout constructions, where pushouts model the gluing of graphs. In fact, there are two main variants: the double and the single pushout approach. The double pushout approach, mainly developed by Ehrig, Schneider and the Berlin and Pisa groups, is the formal basis for the work presented in this thesis.

4. The *logical approach*, mainly developed by Courcelle and Bouderon, allows expressing graph transformation and graph properties in monadic second-order logic.

5. The *theory of 2-structures* was initiated by Rozenberg and Ehrenfeucht as a framework for decomposition and transformation of graphs.

6. The *programmed graph replacement approach* of Schürr combines the gluing and embedding aspects of graph transformation. Moreover, it uses programs in order to control the nondeterministic choice of rule applications.

2.3.1 Double-Pushout Approach

Here we briefly recall the double-pushout (DPO) approach which is one of the standards for graph transformations. The DPO approach was originally introduced in the seventies to formalize a way of performing rewriting on graphs. Today it already has a great amount of theoretical results and applications in several areas [Roz97, EEKR99, EKMR99, EEPT06].

The DPO approach is not restricted to simple graphs, but has been generalized to a large variety of different types of graphs and other kinds of high-level structures such as labeled graphs, typed graphs, hypergraphs, attributed graphs, petri nets [Rei85] and algebraic specifications [EM85, EM90]. This extension from graphs to high-level structures led to the theory of high-level replacement (HLR) systems [EEPT06]. Later on the concept of high-level replacement systems was joined with that of adhesive categories, introduced by Lack and Sobociński in [LS04], leading to the concept of adhesive HLR categories and systems, which enables reusing constructions and results across a wide range of structures.

We begin by defining graphs and graph morphisms that will be used throughout this thesis. The categorical notions required by this section are given in Appendix A. A more detailed exposition and the proofs of the results stated in this section can be found in [EEPT06].

Definition 2.3.1 (Graph and Graph morphism). *A* graph $G = (V, E, s, t, l_v, l_e)$ *consists of a set V of nodes, a set E of edges, two functions $s, t\colon E \to V$ (source and target) and two labeling functions for nodes and edges $l_v\colon V \to \Omega_V$, $l_e\colon E \to \Omega_E$, where Ω_V and Ω_E are node and edge labels.*

A graph morphism $f\colon G_1 \to G_2$ *is a pair of functions $f = (f_E\colon E_1 \to E_2,\ f_V\colon V_1 \to V_2)$, which is compatible with source, target and labeling functions of G_1 and G_2, i.e., $f_V \circ s_1 = s_2 \circ f_E$, $f_V \circ t_1 = t_2 \circ f_E$, $l_{e_2} \circ f_E = l_{e_1}$ and $l_{v_2} \circ f_V = l_{v_1}$.*

$$\begin{tikzcd}
\Omega_E & E_1 \arrow[l, "l_{e_1}"'] \arrow[r, shift left, "s_1"] \arrow[r, shift right, "t_1"'] \arrow[d, "f_E"'] \arrow[dr, phantom, "="] & V_1 \arrow[r, "l_{v_1}"] \arrow[d, "f_V"] & \Omega_V \\
 & E_2 \arrow[ul, "l_{e_2}"] \arrow[r, shift left, "s_2"] \arrow[r, shift right, "t_2"'] & V_2 \arrow[ur, "l_{v_2}"'] &
\end{tikzcd}$$

Fact 2.3.2 (Composition of Graph Morphisms). *Given two graph morphisms $f = (f_V, f_E)\colon G_1 \to G_2$ and $g = (g_V, g_E)\colon G_2 \to G_3$, the* composition $g \circ f = $ *$(g_V \circ f_V, g_E \circ f_E)\colon G_1 \to G_3$ is again a graph morphism.*

Proof. See Fact 2.5 in [EEPT06] (page 22). □

Graph transformation is based on graph productions that describe in a general way how graphs can be transformed. The application of such a production to a graph is called a direct graph transformation.

Definition 2.3.3 (**Graph Production**). *A* graph production $p = L \xleftarrow{l} I \xrightarrow{r} R$ *consists of graphs* L, I *and* R, *called left-hand side, gluing (or interface) graph and right-hand side respectively, and two graph morphisms* l *and* r.

We also call a graph production as graph (transformation) rule.

Definition 2.3.4 (**Graph Transformation**). *Given a graph production* p *as in Definition 2.3.3, a graph* G *and a graph morphism* $m\colon L \to G$, *called match. A* direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ *from* G *to a graph* H *is given by the double-pushout (DPO) diagram below, where* (1) *and* (2) *are pushouts.*

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
G & C \arrow[l] \arrow[r] & H
\end{tikzcd}$$

A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ *of direct graph transformations is called* graph transformation *and is denoted by* $G_0 \Rightarrow^* G_n$.

Now we define graph transformation systems, graph grammars and the language derived from a start graph and a set of graph rules.

Definition 2.3.5 (**GT System, Graph Grammar and Language**). *A* graph transformation system $GTS = (P)$ *consists of a set* P *of graph productions. A* graph grammar $GG = (GTS, S)$ *is a graph transformation system* GTS *equipped with a start graph* S. *The* graph language L *of a graph grammar* GG *is defined by* $L = \{G \mid \exists$ *graph transformation* $S \Rightarrow^* G\}$.

2.3.1.1 Construction of Graph Transformations

We recall the conditions under which a graph production $p = L \leftarrow I \rightarrow R$ can be applied to a graph G via a match $m\colon L \to G$. In general, the existence of a context graph C is required, leading to a pushout. This allows the construction of first pushout square in a direct graph transformation $G \overset{p,m}{\Longrightarrow} H$, where in a second step the graph H is built by gluing C and R via I.

Definition 2.3.6 (**Applicability of Productions**). *A graph production* $p = L \xleftarrow{l} I \xrightarrow{r} R$ *is* applicable *to a graph* G *via a match* $m\colon L \to G$ *if there exists a context graph* C *such that* (1) *is a pushout.*

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \\
G & C \arrow[l] &
\end{tikzcd}$$

Definition 2.3.6 gives no criterion for deciding whether p is applicable or not, whereas this can be easily checked via the gluing condition. Both concepts are equivalent, as shown in Fact 2.3.8.

Definition 2.3.7 (**Gluing Condition**). *Given a graph production $p = L \xleftarrow{l} I \xrightarrow{r} R$, a graph G and a match $m\colon L \to G$ with $X = (V_X, E_X, s_X, t_X)$ for all $X \in \{L, I, R, G\}$, we can state the following definitions:*

- *The* gluing points *GP are those nodes and edges in L that are not deleted by p, i.e., $GP = l_V(V_I) \cup l_E(E_I) = l(I)$.*
- *The* identification points *IP are those nodes and edges in L that are identified by m, i.e., $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$.*
- *The* dangling points *DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to $m(L)$, i.e., $DP = \{v \in V_L \mid \exists e \in E_G \setminus m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.*

The production p and the match m satisfy the gluing condition *if all identification and all dangling points are also gluing points, i.e., $IP \cup DP \subseteq GP$.*

Fact 2.3.8 (**Existence and Uniqueness of Context Graph**). *Given a graph production $p = L \leftarrow I \rightarrow R$ (l is injective), a graph G and a match $m\colon L \to G$ then it holds: there exists a context graph C such that* (1) *in the diagram of Definition 2.3.6 is a pushout if and only if the gluing condition is satisfied. If C exists, it is unique up to isomorphism.*

Proof. See Fact 3.11 in [EEPT06] (page 45). □

In the following we show how a direct transformation is constructed.

Fact 2.3.9 (**Construction of Direct Graph Transformation**). *Given a graph production $p = L \xleftarrow{l} I \xrightarrow{r} R$ and a match $m\colon L \to G$ such that p is applicable to a graph G via m, then we construct the direct graph transformation in two steps:*

1. *Delete the nodes and edges from G that are reached by the match m but keep the ones in I, i.e., $C = (G \setminus m(L)) \cup m(l(I))$. More precisely, we construct the context graph C and the pushout* (1) *such that G is the gluing of L and C along I.*

2. Add the nodes and edges that are created in R, i.e., $H = C \dot{\cup} (R \setminus r(I))$, where the disjoint union $\dot{\cup}$ makes sure that the elements of $R \setminus r(I)$ are added as new elements. More precisely, we build the pushout (2) *such that H is the gluing of R and C along I.*

This construction is unique up to isomorphism.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
G & C \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Proof. See Fact 3.13 in [EEPT06] (page 46). □

2.3.2 Double-Pushout with Borrowed Contexts

Finally, we recall the DPO extension to borrowed contexts [EK04, EK06]. In standard DPO [CMR$^+$97] productions rewrite graphs with no interaction with any other entity than the graph itself and the production. In the DPO with borrowed contexts [EK06] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

The borrowed context framework was originally defined for the category of graph structures, but, as already stated in [EK04, EK06], its results can be automatically lifted to adhesive categories [LS05] since the corresponding proofs only use pushout and pullback constructions which are compliant with adhesive categories. In this section we present the borrowed context setting for the category of labeled graphs (as defined in Appendix A).

Now define the notion of graphs with interfaces and contexts, followed by the definition of a rewriting step with borrowed contexts as defined in [EK06] and extended in [Sob04].

Definition 2.3.10 (Graphs with Interfaces and Contexts). *A* graph G with interface J *is a morphism* $J \to G$ *and a context consists of two morphisms* $J \to E \leftarrow \overline{J}$*. The* embedding *of* $J \to G$ *into a context* $J \to E \leftarrow \overline{J}$ *is a graph with interface* $\overline{J} \to \overline{G}$ *which is obtained by constructing* $\overline{G}$ *as the pushout of* $J \to G$ *and* $J \to E$ *(see diagram below).*

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[dl, dotted] \\
G \arrow[r] & \overline{G} &
\end{tikzcd}$$

The embedding is defined up to isomorphism since the pushout object is unique up to isomorphism. Moreover, embedding/insertion into a context and contextualization are used as synonyms.

Definition 2.3.11 (Rewriting with Borrowed Contexts). *Given a graph with interface $J \to G$ and a production $p\colon L \leftarrow I \to R$, we say that $J \to G$ reduces to $K \to H$ with transition label $J \to F \leftarrow K$ if there are graphs D, G^+, C and additional morphisms such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with injective morphisms. In this case a* rewriting step with borrowed context *(BC step) is called feasible: $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$.*

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

We also call transition labels as (derived) labels.

In the diagram above the upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 2.3.11, namely C and F, may not exist. In this case, the rewriting step is not feasible.

Sassone and Sobociński [SS05] found out that in Definition 2.3.11 some morphisms can also be non-injective, namely the morphisms depicted as → in the diagram below. However, in this thesis we consider all morphisms in Definition 2.3.11 to be injective.

$$\begin{tikzcd}
D \arrow[r, tail] \arrow[d, tail] \arrow[dr, phantom, "PO"] & L \arrow[d, tail] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d, tail] \arrow[dr, phantom, "PO"] & R \arrow[d, tail] \\
G \arrow[r, tail] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r, tail] \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 2.3.12 (**Bisimulation and Bisimilarity**). *Let $\mathcal{P}$ be a set of productions and $\mathcal{R}$ a symmetric relation containing pairs of graphs with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a* bisimulation *if, whenever we have $(J \to G)\, \mathcal{R}\, (J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H),$$

then there exists a graph with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$$

such that $(K \to H)\, \mathcal{R}\, (K \to H')$.

We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two graphs with interface. The relation $\sim$ is called bisimilarity*.*

In order to state Theorem 2.3.14 we have to close a relation under all possible contexts.

Definition 2.3.13 (**Closure under Contexts**). *Let $\mathcal{R}$ be a relation containing pairs of graphs with interfaces as in Definition 2.3.12. By $\hat{\mathcal{R}}$ we denote the closure of $\mathcal{R}$ under contexts, i.e., $\hat{\mathcal{R}}$ is the smallest relation that contains, for every pair $(J \to G, J \to G') \in \mathcal{R}$ and for every context of the form $J \to E \leftarrow \overline{J}$, the pair of graphs with interface $(\overline{J} \to \overline{G}, \overline{J} \to \overline{G}')$ which results from the insertion of $J \to G$ and $J \to G'$ respectively into $J \to E \leftarrow \overline{J}$, as in Definition 2.3.10.*

A relation $\mathcal{R}$ is a congruence, i.e., closed under contexts whenever $\hat{\mathcal{R}} = \mathcal{R}$.

Theorem 2.3.14 (**Bisimilarity is a Congruence**). *Whenever $\mathcal{R}$ is a bisimulation, then $\hat{\mathcal{R}}$ is a bisimulation as well. This implies that the bisimilarity relation $\sim$ is a congruence.*

Proof. See Theorem 4.3 in [EK06]. □

In practice, we can decide whether two graphs with interface $J \to G$ and $J \to G'$ are bisimilar w.r.t. a set $\mathcal{P}$ of graph productions by deriving their corresponding transition labels via Definition 2.3.11 and trying to match them in the bisimulation game of Definition 2.3.12. An algorithm to mechanize this process is presented in Chapter 4. Also the fact that bisimilarity is a congruence will be exploited by the techniques developed in this thesis in order to reason about behavior preservation in model refactoring.

The borrowed context framework was originally defined in [EK04, EK06], but it has been extended and applied to several cases studies in the more recent years. Below we summarize the main results:

1. *Deriving Process Congruences from Reaction Rules* [SS05, Sob04]: Sassone and Sobociński showed that the borrowed context framework to label derivation is an instance of their theory of groupoidal relative-pushouts (**G**-RPOs). This close connection allows the transfer of results to the BC framework, namely the use of certain non-injective morphisms in Definition 2.3.11 (rewriting with borrowed contexts) and also their results on other operational equivalences such as traces and failures equivalences;

2. *Composition and Decomposition of DPO Transformations with Borrowed Context* [BEK06a, BEK06b]: in this work, focusing on the situation in which the states of a global system are built out of local components, it is shown that DPO transformations with borrowed context defined on a global system state can be decomposed into corresponding transformations on the local states and vice versa;

3. *Process Bisimulation via a Graphical Encoding* [BGK06a, BGK06b]: here it is illustrated that the BC framework is a suitable tool to analyze process calculi. Milner's CCS is encoded as graphs with interfaces, labels are derived using the BC machinery and finally it is proved that the bisimilarity on processes obtained via BCs coincides with the standard strong bisimilarity for CCS;

4. *Bisimulation Verification for the DPO Approach with Borrowed Contexts* [RKE07]: Hirschkoff's on-the-fly bisimulation checking algorithm is extended to the BC setting. This algorithm is employed to check refactoring steps for bisimilarity, more specifically to check that a deterministic finite automaton is bisimilar to its refactored version after merging equivalent states. More details can be found in Chapter 4 of this thesis;

5. *Deriving Bisimulation Congruences in the Presence of Negative Application Conditions* [RKE08a, RKE08b]: negative application conditions (NACs) are essential to restrict the applicability of a rule and to model complex systems. In this work the BC framework is extended to handle NACs. The extension, which is carried out for adhesive categories, requires an enrichment of the labels which now do not only indicate the context that is provided by the observer, but also constrain further additional contexts that may (not) satisfy the negative application condition. That is, we do not only specify what must be borrowed, but also what must not be borrowed. It has been shown that the main result of [EK06] (bisimilarity is a congruence) still holds for this extension. The theory was illustrated by an example in terms of blade server systems. More details can be found in Chapter 3 of this thesis;

6. *Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed Contexts* [RLK$^+$08a, RLK$^+$08b]: borrowed contexts are used, and, exploiting the fact that observational equivalence is a congruence, it is shown how to check refactoring rules for behavior preservation. If rules are behavior-preserving, their application will never change behavior, i.e., every model and its refactored version will have the same behavior. However, there might exist refactoring rules that are not behavior-preserving, even though the full refactoring preserves the behavior. For these cases a technique to combine refactoring rules to behavior-preserving concurrent productions is presented in order to ensure behavior preservation. These techniques are applied to an automaton example. More details can be found in Chapter 5 of this thesis as well;

7. *Parallel and Sequential Independence for Borrowed Contexts* [BGH08]: the concepts of parallel and sequential independence of DPO transformations are lifted to the BC setting. Moreover, it is show that the local Church-Rosser and parallelism theorems guarantee local confluence and the parallel execution of independent borrowed context steps.

The items 4, 5 and 6 above form the core of this thesis.

Chapter 3

Deriving Bisimulation Congruences in the Presence of NACs

3.1 Motivation

Bisimilarity is an equivalence relation on states of transition systems, associating states that can match each other's moves. In this sense, bisimilar states can not be distinguished by an external observer. Bisimilarity provides a powerful proof technique to analyze the properties of systems and has been extensively studied in the field of process calculi since the early 80's. Especially for CCS [Mil89] and the π-calculus [MP92, MPW92] an extensive theory of bisimulation is now available.

Congruence is a very desirable property that a bisimilarity may have, since it allows the exchange of bisimilar systems in larger systems without effect on the observable behavior. Unfortunately, a bisimulation defined on unlabeled reaction rules is in general not a congruence. Hence, Leifer and Milner [Lei01, LM00] proposed a method that uses so-called idem pushouts (IPOs) to derive a labeled transition system from unlabeled reaction rules such that the resulting bisimilarity is a congruence. Motivated by this work, Ehrig and König proposed in [EK04, EK06] an extension to the double pushout approach (DPO, for short) called DPO with borrowed contexts (DPO-BC), which provides the means to derive labeled transitions from rewriting rules in such a way that the bisimilarity is automatically a congruence. This has turned out to be equivalent to a technique by Sassone and Sobociński [SS05, Sob04] which derives labels via groupoidal idem pushouts. In all approaches the basic idea is the one suggested by Leifer and Milner: the labels should be the minimal contexts that an observer has to provide in order to trigger a reduction.

The DPO with borrowed contexts works with productions consisting of two arrows $L \leftarrow I \rightarrow R$ where the arrows are either graph morphisms, or—more generally—

arrows in an adhesive category. Even though the generative power of the DPO approach is sufficient to generate any recursively enumerable set of graphs, very often extra application conditions are a required feature of nontrivial specifications. Negative application conditions (NACs) [HHT96] for a graph production are conditions such as the non-existence of nodes, edges, or certain subgraphs in the graph G being rewritten, as well as embedding restrictions concerning the match $L \rightarrow G$. Similar restrictions can also be achieved in Petri nets with inhibitor arcs, where these arcs impose an extra requirement to transition firing, i.e., a transition can only be fired if certain places are currently unmarked.

Graph transformation systems, which are our main focus, are often used for specification purposes, where—in contrast to programming—it is quite convenient and often necessary to constrain the applicability of rules by negative application conditions. We believe that this is a general feature of specification languages, which means that the problem of deriving behavioral equivalences in the presence of NACs may occur in many different settings.

In this chapter we extend the borrowed context framework to handle productions with negative application conditions. The extension, which is carried out for adhesive categories, requires an enrichment of the labels which now do not only indicate the context that is provided by the observer, but also constrain further additional contexts that may (not) satisfy the negative application condition. That is, we do not only specify what must be borrowed, but also what should not be borrowed. We prove that the main result of [EK06] (bisimilarity is a congruence) still holds for our extension. Moreover, we further develop up-to techniques in order to cope with NACs and apply the up-to context to examples.

The work presented in this chapter is based on our paper [RKE08a, RKE08b], and structured as follows. Section 3.2 briefly reviews the DPO approach with borrowed contexts. In Section 3.3 we discuss the problems which arise due to productions with NACs and how they can be overcome in order to guarantee that the derived bisimilarities are congruences. Section 3.4 presents proof techniques for our extension. Finally, we present two examples in terms of graph transformation: a small one in Section 3.5 and a more elaborate one in Section 3.6, where blade server systems are presented. Additional proofs and further information about the examples can be found in Appendix B.

3.2 Double-Pushout with Borrowed Contexts

In this section we lift the DPO approach with borrowed contexts [EK04, EK06] already shown in Section 2.3.2 for the category of graphs to the general framework of adhesive categories [LS05]. Furthermore, we define the gluing condition for borrowed context steps.

In standard DPO [CMR$^+$97], productions rewrite graphs with no interaction with any other entity than the graph itself and the production. In DPO with borrowed contexts [EK06] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

The DPO-BC framework was originally defined for the category of graph structures, but, as already stated in [EK04, EK06], its results can be automatically lifted to adhesive categories since the corresponding proofs only use pushout and pullback constructions which are compliant with adhesive categories. In the following we present the DPO-BC setting for adhesive categories [LS05] to which we first give a short introduction.

Definition 3.2.1 (Adhesive Category). *A category* **C** *is called* adhesive *if*

1. **C** *has pushouts along monos;*
2. **C** *has pullbacks;*
3. *Given a cube diagram as shown on the right with: (i) $A \to C$ mono, (ii) the bottom square a pushout and (iii) the left and back squares pullbacks, we have that the top square is a pushout iff the front and right squares are pullbacks.*

$$\begin{tikzcd}
A' \arrow[rr] \arrow[dr] \arrow[dd] & & C' \arrow[dr] \arrow[dd] & \\
& B' \arrow[rr] \arrow[dd] & & D' \arrow[dd] \\
A \arrow[rr] \arrow[dr] & & C \arrow[dr] & \\
& B \arrow[rr] & & D
\end{tikzcd}$$

Pullbacks preserve monos and pushouts preserve epis in any category. Furthermore, for adhesive categories it is known that monos are preserved by pushouts [LS05]. For the DPO-BC extension to productions with negative application conditions, defined in Section 3.3, we need one further requirement, namely that pullbacks preserve epis. This means that if the square (A', B', A, B) above is a pullback and $A \to B$ is epi, we can conclude that $A' \to B'$ is epi as well.

Our prototypical instance of an adhesive category, which will be used for the examples in the paper, is the category of labeled graphs (see Appendix A), where arrows are graph morphisms. In this category pullbacks preserve epis.

We will now define the notion of objects with interfaces and contexts, followed by the definition of a rewriting step with borrowed contexts as defined in [EK06].

Definition 3.2.2 (Objects with Interfaces and Contexts). *An* object G with interface J *is an arrow $J \to G$ and a context consists of two arrows $J \to E \leftarrow \overline{J}$.*

The embedding *of $J \rightarrow G$ into a context $J \rightarrow E \leftarrow \overline{J}$ is an object with interface $\overline{J} \rightarrow \overline{G}$ which is obtained by constructing $\overline{G}$ as the pushout of $J \rightarrow G$ and $J \rightarrow E$ (see diagram below).*

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[dl, dotted] \\
G \arrow[r] & \overline{G} &
\end{tikzcd}$$

The embedding is defined up to isomorphism since the pushout object is unique up to isomorphism. Moreover, embedding/insertion into a context and contextualization are used as synonyms.

Definition 3.2.3 (Rewriting with Borrowed Contexts). *Given an object with interface $J \rightarrow G$ and a production $p\colon L \leftarrow I \rightarrow R$, we say that $J \rightarrow G$ reduces to $K \rightarrow H$ with transition label $J \rightarrow F \leftarrow K$ if there are objects D, G^+, C and additional arrows such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with monos. In this case a* rewriting step with borrowed context *(BC step) is called feasible: $(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H)$.*

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

We also call transition labels as (derived) labels.

In the diagram above the upper left-hand square merges L and the object G to be rewritten according to a partial match $G \leftarrow D \rightarrow L$. The resulting object G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with an arrow $J \rightarrow F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting object H, which can be obtained by "intersecting" the borrowed context F and the object C via a pullback. Note that the two pushout complements that are needed in Definition 3.2.3, namely C and F, may not exist. In this case, the rewriting step is not feasible.

With the procedure described above we may derive infinitely many labels of the form $J \rightarrow F \leftarrow K$. However, observe that there are only finitely many up to iso and hence they can be represented in a finite way.

In the standard DPO approach given a production $L \leftarrow I \rightarrow R$ and a match $L \rightarrow G$ one can employ the gluing condition (Definition 2.3.7) to check whether there exists the pushout complement $I \rightarrow C \rightarrow G$ of $I \rightarrow L \rightarrow G$. This notion of gluing condition can be elegantly formalized with the categorical concept of initial pushout. In Appendix A.3 one can find the definition of initial pushout, its construction for the category of graphs and the formalization of the gluing condition in terms of initial pushouts.

Similarly, based on a technique defined in [BGK06a] which uses initial pushouts to check whether a partial match $G \leftarrow D \rightarrow L$ leads to the existence of the borrowed context F, we define the notion of gluing condition for borrowed context steps.

Definition 3.2.4 (Gluing Condition of Borrowed Context Steps). *Given an adhesive category with initial pushouts, a production $p\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ and an object with interface $J \overset{j}{\rightarrow} G$, then a partial match $G \overset{dg}{\leftarrow} D \overset{dl}{\rightarrow} L$ satisfies the* gluing condition of a borrowed context step *with respect to p and $J \overset{j}{\rightarrow} G$ if the following conditions hold for the diagram below:*

(i) for the initial pushout (1) *over dl there exists a mono $jj\colon J_D \rightarrow J$ such that $dg \circ jd = j \circ jj$;*

(ii) for the initial pushout (2) *over dg there exists a mono $ii\colon I_D \rightarrow I$ such that $dl \circ id = l \circ ii$.*

$$\begin{tikzcd}
 & J_D \arrow[r] \arrow[d, "jd"] \arrow[dddd, bend right, "jj"'] \arrow[dr, phantom, "(1)"] & F_D \arrow[d] & & \\
I_D \arrow[r, "id"'] \arrow[d] \arrow[rrr, bend left, "ii"] \arrow[dr, phantom, "(2)"] & D \arrow[r, "dl"'] \arrow[d, "dg"] & L & I \arrow[l, "l"] \arrow[r, "r"'] & R \\
C_D \arrow[r] & G & & & \\
 & & & & \\
 & J \arrow[uu, "j"'] & & &
\end{tikzcd}$$

The gluing condition of a borrowed context step can be quickly checked: we only need to build $J_D \overset{jd}{\rightarrow} D$ and $I_D \overset{id}{\rightarrow} D$ (the construction for the category of graphs is given in Appendix A.3) and check whether there exist jj and ii leading to the required commutativity. Note that this is usually easier than building the pushout of dg and dl and checking the existence of F and C by using the gluing condition of standard DPO (Definition 2.3.7).

Theorem 3.2.5 (Existence and Uniqueness of Contexts). *Given an adhesive category with initial pushouts, a production $p\colon L \xleftarrow{l} I \xrightarrow{r} R$ and an object with interface $J \xrightarrow{j} G$, then a partial match $G \xleftarrow{dg} D \xrightarrow{dl} L$, which leads to* (3) *as a pushout, satisfies the gluing condition of a borrowed context step with respect to p and $J \xrightarrow{j} G$ if and only if there exist context objects F and C, i.e., there exist pushout complements* (4) *and* (5) *of $J \xrightarrow{j} G \rightarrow G^+$ and $I \xrightarrow{l} L \rightarrow G^+$, respectively.*

$$\begin{tikzcd}
& J_D \arrow[r] \arrow[d, "jd"] \arrow[ddd, bend right, "jj"'] \arrow[dr, phantom, "(1)"] & F_D \arrow[d] \arrow[ddd, bend left, "fd"] & & \\
I_D \arrow[r, "id"'] \arrow[d] \arrow[rrr, bend left, "ii"] \arrow[dr, phantom, "(2)"] & D \arrow[r, "dl"'] \arrow[d, "dg"] \arrow[dr, phantom, "(3)"] & L \arrow[d] \arrow[dr, phantom, "(5)"] & I \arrow[l, "l"] \arrow[r, "r"'] \arrow[d] & R \\
C_D \arrow[r] \arrow[rrr, bend right, "cd"'] & G \arrow[r] \arrow[dr, phantom, "(4)"] & G^+ & C \arrow[l] & \\
& J \arrow[u, "j"'] \arrow[r] & F \arrow[u] & &
\end{tikzcd}$$

Whenever they exist, the context objects F and C are unique up to isomorphism.

Proof. This proof is split into two parts.

"⇒": if the gluing condition of Definition 3.2.4 is satisfied then we construct the pushout $J \rightarrow F \xleftarrow{fd} F_D$ of $J \xleftarrow{jj} J_D \rightarrow F_D$. Since $J_D \rightarrow F_D$ is mono (see Definition A.20 of initial pushout) then so is $J \rightarrow F$. By Lemma A.25 (composition of initial pushout and pushout) we can infer that $(1) + (3)$ is an initial pushout over $G \rightarrow G^+$. The square (J_D, F_D, J, F) as a pushout, the commutativity of $(1)+(3)$ and $dg \circ jd = j \circ jj$ (assumption) imply that there exists a unique morphism $F \rightarrow G^+$ such that (4) and the morphisms in (F_D, L, G^+, F) commute. Since $(1) + (3)$ and (J_D, F_D, J, F) are pushouts and the vertical morphisms above respectively commute with jj and fd, then we can infer that (4) is a pushout by pushout decomposition. Finally, j mono implies $F \rightarrow G^+$ mono. Analogously we build the context object C, where (5) is a pushout along monos.

"⇐": if the context object F exists with (4) as a pushout, then the initiality of the pushout $(1) + (3)$ implies the existence of $jj\colon J_D \rightarrow J$ mono with $dg \circ jd = j \circ jj$. Analogously for C we find that there exists $ii\colon I_D \rightarrow I$ mono with $dl \circ id = l \circ ii$.

The uniqueness of F and C follows from the uniqueness of pushout complements.

□

A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 3.2.6 (Bisimulation and Bisimilarity). *Let $\mathcal{P}$ be a set of productions and $\mathcal{R}$ a symmetric relation containing pairs of objects with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a* bisimulation *if, whenever we have $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H),$$

then there exists an object with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$$

such that $(K \to H)\,\mathcal{R}\,(K \to H')$.

We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two objects with interface. The relation $\sim$ is called bisimilarity*.*

Theorem 3.2.7 (Bisimilarity is a Congruence). *The bisimilarity relation $\sim$ is a congruence, i.e., it is preserved by contextualization as described in Definition 3.2.2.*

Proof. See Theorem 4.3 in [EK06]. □

3.3 Borrowed Contexts with NACs

Here we will extend the DPO-BC framework of [EK06] to productions with negative application conditions. Prior to the extension we will investigate in Section 3.3.1 why such an extension is not trivial. It is worth emphasizing that the extension will be carried out for adhesive categories with an additional requirement that pullbacks preserve epis, but the examples will be given in the category of labeled directed graphs. First, we define negative application conditions for productions.

Definition 3.3.1 (Negative Application Condition). *A negative application condition $NAC(n)$ on L is a mono $n\colon L \to NAC$. A mono $m\colon L \to G$ satisfies $NAC(n)$ on L if and only if there is no mono $q\colon NAC \to G$ with $q \circ n = m$.*

$$\begin{tikzcd}
NAC \arrow[dr, "q"', "|" marking] & L \arrow[l, tail, "n"'] \arrow[d, "m"] \arrow[dl, phantom, "="] \\
 & G
\end{tikzcd}$$

A rule $L \leftarrow I \to R$ with NACs is equipped with a finite set of negative application conditions $\{L \to NAC_y\}_{y \in Y}$ and is applicable to a match $m\colon L \to G$ only if all NACs are satisfied. If we add NACs to the rules in Definition 3.2.3, we have two ways to check their satisfiability: before (on G) or after the borrowing (on G^+), but the latter is more suitable since the first one does not take into account any borrowed structure.

3.3.1 Bisimulation and NACs – Is Bisimilarity still a Congruence?

Let us assume that borrowed context rewriting works as in Definition 3.2.3 if the total match $L \to G^+$ satisfies all NACs of a production, i.e., G^+ does not contain any prohibited structure (specified by a NAC) at the match of L. With the following example in terms of labeled directed graphs we will show that this notion is not yet the right one.

On the right-hand side of Figure 3.1 we depict two servers as graphs with interfaces: $\mathsf{J} \to \mathsf{G}$ and $\mathsf{J} \to \mathsf{G}'$. An s-node represents a server. Each server has two queues Q_1 and Q_2 where it receives tasks to be processed. The queue Q_1 is of high priority. Tasks are modeled as loops and may either be standard (T) or urgent (U). In real world applications, standard tasks may come from regular users while urgent ones come from administrators. On the left-hand side of Figure 3.1 we depict how the servers work. Rule_1 says that an urgent task in Q_2 must be immediately executed, whereas Rule_2 specifies how a standard task T in Q_2 is executed. The negative application condition NAC_1 allows rule_2 to be applied only when there is no other T-task waiting in the high priority queue Q_1. We assume that a processed task is consumed by the server (see R_1 and R_2).

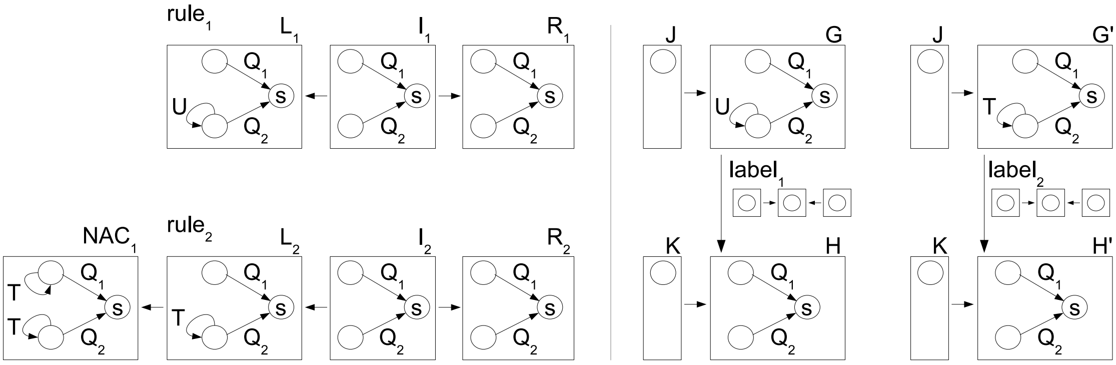


Figure 3.1: Rules for task processing (left) and the LTSs of two servers (right).

From the servers $\mathsf{J} \to \mathsf{G}$ and $\mathsf{J} \to \mathsf{G}'$ we derive the labeled transition system (LTS) to the right of Figure 3.1 w.r.t. rule_1 and rule_2. The derivation of label_2 is depicted in Figure 3.2. No further label can be derived from $\mathsf{K} \to \mathsf{H}$ and $\mathsf{K} \to \mathsf{H}'$ and the labels leading to these graphs are equal. By Definition 3.2.6 (bisimulation) we could conclude that $(\mathsf{J} \to \mathsf{G}) \sim (\mathsf{J} \to \mathsf{G}')$. Since bisimilarity is a congruence (at least for rules without NACs), the insertion of $\mathsf{J} \to \mathsf{G}$ and $\mathsf{J} \to \mathsf{G}'$ into a context C, as in Definition 3.2.2, produces graphs $\overline{\mathsf{J}} \to \overline{\mathsf{G}}$ and $\overline{\mathsf{J}} \to \overline{\mathsf{G}}'$ respectively, which should be bisimilar. Figure 3.3 shows a context C with a standard task, the resulting graphs $\overline{\mathsf{J}} \to \overline{\mathsf{G}}$ and $\overline{\mathsf{J}} \to \overline{\mathsf{G}}'$ which received the T-task in queue Q_1 via the interface J, and their

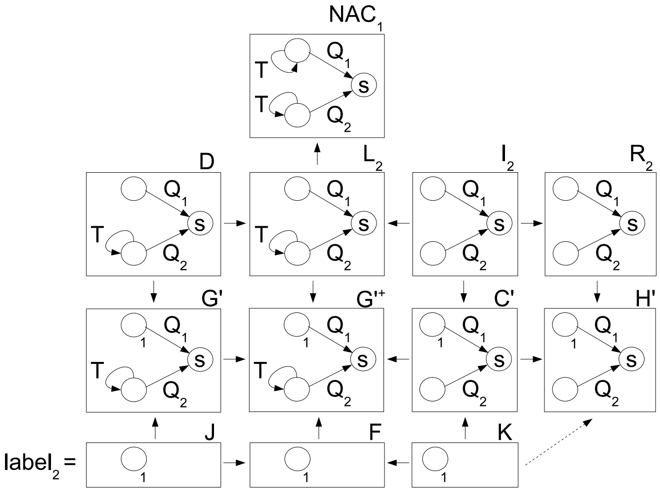


Figure 3.2: Derivation of label_2 from $\mathsf{J} \to \mathsf{G}'$ via rule_2

LTS. The server $\overline{\mathsf{J}} \to \overline{\mathsf{G}}'$ cannot perform any transition since NAC_1 of rule_2 forbids the BC step, i.e., the T-task in Q_2 cannot be executed because there is another standard task in the high priority queue Q_1. However, $\overline{\mathsf{J}} \to \overline{\mathsf{G}}$ is still able to perform a transition and evolve to $\overline{\mathsf{K}} \to \overline{\mathsf{H}}$. Thus, bisimilarity is no longer a congruence when productions have NACs.

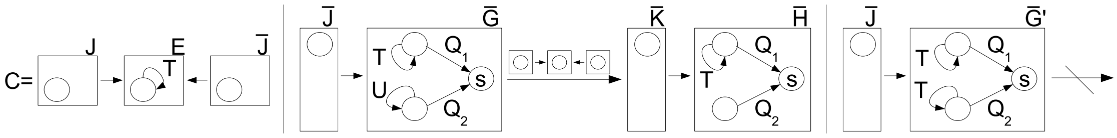


Figure 3.3: A context C (left) and the resulting LTSs for $\overline{\mathsf{J}} \to \overline{\mathsf{G}}$ and $\overline{\mathsf{J}} \to \overline{\mathsf{G}'}$ (right).

The LTS for $\mathsf{J} \to \mathsf{G}$ and $\mathsf{J} \to \mathsf{G}'$ (see Figure 3.1) shows that label_1, which is derived from rule_1 (without NAC) is matched by label_2, which is generated by rule_2 (with NAC). These matches between labels obtained from rules with and without NACs are the reason why the congruence property does no longer hold. In fact, the actual definitions of bisimulation and borrowed context step are too coarse to handle NACs.

Our idea is to enrich the transition labels $J \to F \leftarrow K$ with some information provided by the NACs in order to define a finer bisimulation based on these labels. A label must not only know which structures (borrowed context) are needed to perform it, but also which forbidden structures (defined by the NACs) cannot be additionally offered by the environment in order to guarantee NAC satisfiability. These forbidden structures will be called *negative borrowed contexts* and are represented by objects N_z attached to the label via monos from the borrowed context F (see Figure 3.4). In our server example, label_1 would remain without any negative borrowed context since rule_1 has no NAC. However, label_2 would be the label depicted in Figure 3.4, where the negative borrowed context $\mathsf{F} \rightarrow \mathsf{N}_1$ specifies that if a T-task was in Q_1, then NAC_1 would have forbidden the BC step of $\mathsf{J} \rightarrow \mathsf{G}'$ via rule_2. That is, with the new form of labels the two graphs are no longer bisimilar and hence we no longer have a counterexample to the congruence property.

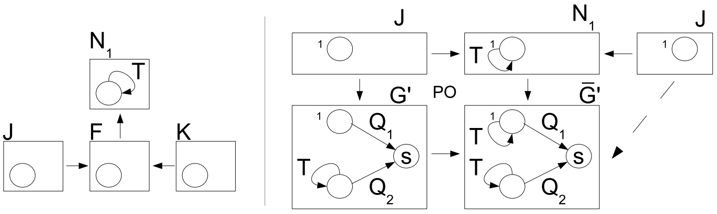


Figure 3.4: Transition label enriched with negative borrowed context (left) and insertion of $\mathsf{J} \rightarrow \mathsf{G}'$ into a forbidden context (right).

The intuition of negative borrowed contexts is the following: given $J \rightarrow G$, whenever it is possible to derive a label $J \rightarrow F \leftarrow K$ with negative borrowed context $F \rightarrow N_z$ via a production p with NACs, then if $J \rightarrow G$ is inserted into a context[1] $J \rightarrow N_z \leftarrow J$ leading to $J \rightarrow \overline{G}$ no further label can be derived from $J \rightarrow \overline{G}$ via p since some of its NACs will forbid the rule application (see the right-hand side of Figure 3.4). Put differently, the label says that a transition can be executed if the environment "lends" F as minimal context. Furthermore, the environment can observe that the NACs of a production are only satisfiable under certain constraints on the context. Finally, it is not executable at all if the object G^+ with borrowed context already contains the NAC.

3.3.2 Borrowed Contexts – Extension to Rules with NACs

Now we are ready to extend the DPO-BC framework to deal with productions with NACs. First we define when a BC step is NAC consistent.

Definition 3.3.2 (NAC-Consistent Borrowed Context Step). *Assume that all arrows are mono. Given $J \rightarrow G$ and a production $p\colon L \leftarrow I \rightarrow R; \{n_y\colon L \rightarrow NAC_y\}_{y \in Y}$ we say that a partial match $pm\colon G \leftarrow D \rightarrow L$ leads to a* NAC consistent BC step *with respect to $J \rightarrow G$ and p if for the pushout G^+ in the diagram below there is no $q_y\colon NAC_y \rightarrow G^+$ with $m = q_y \circ n_y$ for every $y \in Y$.*

$$\begin{tikzcd}
 & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[r, "n_y"] \arrow[d, "m"'] \arrow[dr, phantom, "="] & NAC_y \arrow[dl, "q_y", "|" marking] \\
J \arrow[r] & G \arrow[r] & G^+ &
\end{tikzcd}$$

[1] $J \rightarrow N_z$ is the composition of $J \rightarrow F \rightarrow N_z$.

In the following we need the concept of a pair of jointly epi arrows in order to "cover" an object with two other objects. That is needed to find possible overlaps between the NACs and the object G^+ which includes the borrowed context.

Definition 3.3.3 (**Jointly Epi Arrows**). *Two arrows $f\colon A \to B$ and $g\colon C \to B$ are* jointly epi *whenever for every pair of arrows $a, b\colon B \to D$ such that $a \circ f = b \circ f$ and $a \circ g = b \circ g$ it holds that $a = b$.*

In a pushout square the generated arrows are always jointly epi. This is a straightforward consequence of the uniqueness of the mediating arrow.

Definition 3.3.4 (**Borrowed Context Rewriting for Rules with NACs**). *Given $J \to G$, a production $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ and a partial match $G \leftarrow D \to L$, we say that $J \to G$ reduces to $K \to H$ with transition label $J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}$ if the following holds:*

(i) the BC step is NAC consistent (as in Definition 3.3.2);

(ii) there are objects G^+, C and additional arrows such that Diagram (3.1) below commutes and the squares are either pushouts (PO) or pullbacks (PB) with monos;

(iii) the set $\{F \to N_z\}_{z \in Z}$ contains exactly the arrows constructed via Diagram (3.2) (where all arrows are mono). (That is, there exists an object M_z such that all squares commute and are pushouts or arrows are jointly epi as indicated.)

$$\begin{tikzcd}
& NAC_y & & \\
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[u, "n_y"] \arrow[d, "m"] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[u] \arrow[r] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[d] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
& N_z & &
\end{tikzcd}$$

(3.1)

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u, "n_y"] \arrow[r, "m"'] \arrow[ur, phantom, "="] & G^+ \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd}$$

(3.2)

In this case a borrowed context step *(BC step) is feasible and we write: $(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H)$.*

Observe that Definition 3.3.4 coincides with Definition 3.2.3 when no NACs are present (cf. Condition (ii)). By taking NACs into account, a BC step can only be NAC consistent when G^+ contains no forbidden structure of any negative application condition NAC_y at the match of L (Condition (i)). Additionally, enriched labels are generated (Condition (iii)).

In Condition (iii) the arrows $F \to N_z$ are also called *negative borrowed contexts* and each N_z represents the structures that should not be in G^+ in order to enable a NAC-consistent BC step via p (see Lemma 3.3.8). This extra information in the label is of fundamental importance for the bisimulation game with NACs (Definition 3.3.5), where two objects with interfaces must not only agree on the borrowed context which enables a transition but also on what should not be offered by the environment in order to perform the transition. The negative borrowed contexts $F \to N_z$ are obtained from $NAC_y \overset{n_y}{\leftarrow} L \overset{m}{\to} G^+ \leftarrow F$ of Diagram (3.1) via Diagram (3.2), where we create all possible overlaps M_z of G^+ and NAC_y in order to check which structures the environment should not provide in order to assure a NAC-consistent BC step. To consider all possible overlaps is necessary in order to take into account that parts of the NAC might already be present in the object which is being rewritten.

Whenever the pushout complement in Diagram (3.2) exists, the object G^+ with borrowed context can be extended to M_z by attaching the negative borrowed context N_z via F. When the pushout complement does not exist, some parts of G^+ which are needed to perform the extension are not "visible" from the environment and no negative borrowed context is generated.

Due to the non-uniqueness of the jointly-epi square one single negative application condition NAC_y may produce more than one negative borrowed context as depicted in Figure 3.5. The rule used in the BC step on the left shows that an online server (marked with an ON-loop) can be turned off only if there is no standard task in any of its queues. Note that there are two possible overlaps between NAC_1 and G^+. On the right we show the two corresponding negative borrowed contexts $\{\mathsf{F} \to \mathsf{N_z}\}_{\mathsf{z} \in \{1,2\}}$. We depict in detail the construction of $\mathsf{F} \to \mathsf{N}_1$ as described in Definition 3.3.4.

Furthermore, in Definition 3.3.4 the set $\{F \to N_z\}_{z \in Z}$ is in general infinite, but if we consider finite objects L, NAC_y and G^+ (i.e., objects which have only finitely many subobjects) there exist only finitely many overlaps M_z up to iso. Hence the set $\{F \to N_z\}_{z \in Z}$ can be finitely represented by forming appropriate isomorphism classes of arrows. Note that $\mathsf{F} \overset{\mathsf{f}_1}{\to} \mathsf{N}_1$ and $\mathsf{F} \overset{\mathsf{f}_2}{\to} \mathsf{N}_2$ in Figure 3.5 are not isomorphic, because the mono $\mathsf{f}\colon \mathsf{N}_1 \to \mathsf{N}_2$ is not compatible with f_1 and f_2.

It is important to state that replacing the jointly-epi square by a pushout square in Definition 3.3.4 leads to a construction where each NAC may generate at most one negative borrowed context. For the example in Figure 3.5 we would build the pushout $\mathsf{NAC}_1 \to \mathsf{M}^{\mathsf{PO}} \leftarrow \mathsf{G}^+$ (see Figure 3.6) of $\mathsf{NAC}_1 \leftarrow \mathsf{L} \to \mathsf{G}^+$, where the pushout

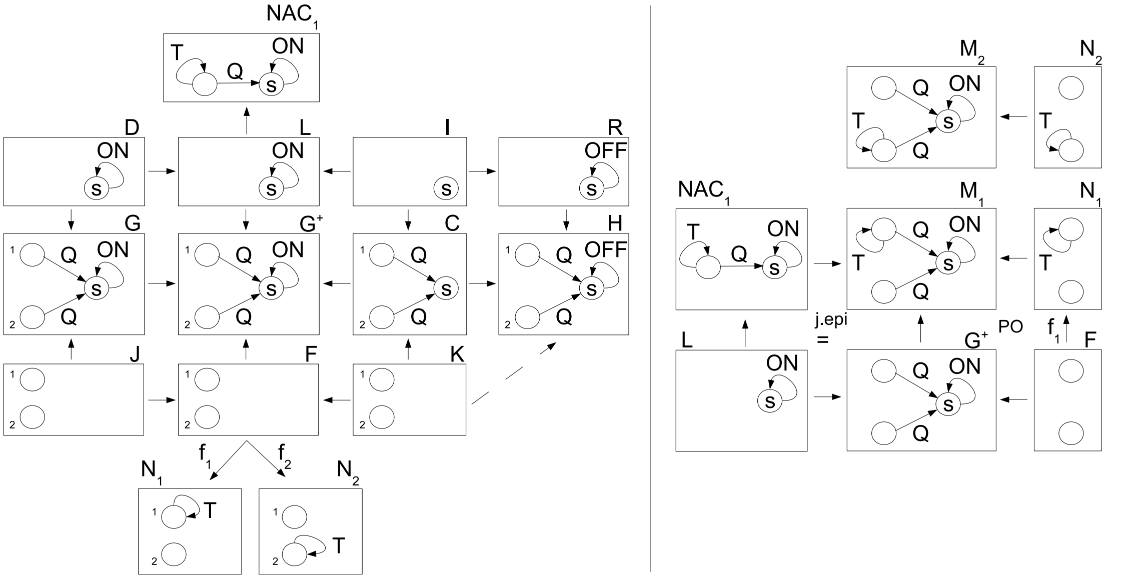


Figure 3.5: A borrowed context step (left) and the construction of negative borrowed contexts (right).

object M^{PO} contains three Q-queues. However, the top queue in M^{PO} can not be provided by the environment since the server is not in F. Hence, the negative borrowed context N^{PO} does not exist. Note that this construction with two pushouts fails to determine the structures that should not be provided to the other two queues in G^+ (cf. Figure 3.5) in order to ensure a NAC-consistent BC step. Therefore, we need the jointly-epi square to build transition labels which lead to bisimilarity as a congruence.

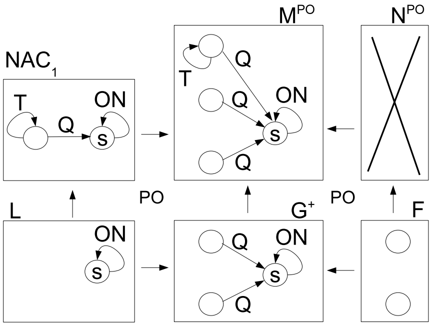


Figure 3.6: Hypothetical construction of negative borrowed contexts with two pushout squares.

Definition 3.3.5 (Bisimulation and Bisimilarity with NACs). *Let $\mathcal{P}$ be a set of productions with NACs and $\mathcal{R}$ a symmetric relation containing pairs of objects with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a* bisimulation with NACs *if,* *for every $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H),$$

there exists an object with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H')$$

such that $(K \to H)\,\mathcal{R}\,(K \to H')$.

We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two objects with interface. The relation $\sim$ is called bisimilarity with NACs.

We often drop "with NACs" from bisimulation (bisimilarity) when it is clear from context.

The difference between the bisimilarity of Definition 3.2.6 and the one above with NACs is the transition label, which in the latter case is enriched with negative borrowed contexts. Thus, Definition 3.3.5 yields in general a finer bisimulation.

In order to state Theorem 3.3.10 (bisimilarity with NACs is a congruence) we have to close a relation under all possible contexts.

Definition 3.3.6 (**Closure under Contexts**). *Let $\mathcal{R}$ be a relation containing pairs of objects with interfaces as in Definition 3.3.5. By $\hat{\mathcal{R}}$ we denote the closure of $\mathcal{R}$ under contexts, i.e., $\hat{\mathcal{R}}$ is the smallest relation that contains, for every pair $(J \to G, J \to G') \in \mathcal{R}$ and for every context of the form $J \to E \leftarrow \overline{J}$, the pair of objects with interface $(\overline{J} \to \overline{G}, \overline{J} \to \overline{G}')$ which results from the insertion of $J \to G$ and $J \to G'$ respectively into $J \to E \leftarrow \overline{J}$, as in Definition 3.2.2.*

A relation $\mathcal{R}$ is a congruence, i.e., closed under contexts whenever $\hat{\mathcal{R}} = \mathcal{R}$.

We need three extra lemmas before we show the congruence result. Recall that we are working in the framework of adhesive categories. Lemma 3.3.7, which is used in the proof of our main result, needs one extra requirement, namely that pullbacks preserve epis.

To prove that bisimilarity is a congruence, borrowed context steps as well as transition labels should allow being composed and decomposed [EK06]. Whenever rules have NACs we have to additionally ensure that negative borrowed contexts can be composed and decomposed, as stated in Lemma 3.3.7.

Lemma 3.3.7 (**NAC Compatibility**). *In the following let all arrows be mono and let Diagram (3.3) be given.*

If we have Diagram (3.5), then there exist objects M_z, N_z and M'_x such that Diagram (3.4)+(3.6) can be constructed as indicated. Furthermore, if we have Diagram (3.4)+ (3.6), then there exists an object $\overline{M}_x$ such that Diagram (3.5) can be constructed as indicated.

$$\begin{tikzcd}
L \arrow[d] \arrow[dr] \arrow[d, phantom, shift left=3, "="] & & \\
G^+ \arrow[r] & \overline{G}^+ & \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "PO"] & E_2 \arrow[u] \arrow[u, phantom, shift left=3, "="] & \overline{F} \arrow[l] \arrow[ul]
\end{tikzcd} \tag{3.3}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & \overline{M}_x & \overline{N}_x \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & \overline{G}^+ \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.5}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & G^+ \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.4}$$

$$\begin{tikzcd}
N_z \arrow[r] & M'_x & \overline{N}_x \arrow[l] \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & E_2 \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.6}$$

The lemmas below are required to infer the NAC-consistency property of one borrowed context step from another.

Lemma 3.3.8. *A borrowed context step (as in Definition 3.3.4) is not NAC consistent whenever there exists a mono $q_y \colon NAC_y \to G^+$ such that $m = q_y \circ n_y$ (see Definition 3.3.2). This is equivalent to the situation, in which:*

(i) there exists a negative borrowed context $F \to N_z$ which is an iso;

or

(ii) there exists a mono $N_z \to G^+$ such that $F \to G^+ = F \to N_z \to G^+$.

Lemma 3.3.9. *In the diagram below, where all arrows are mono, it holds: whenever $\overline{F} \to \overline{N}$ is not iso then $F \to N$ is not iso as well.*

$$\begin{tikzcd}
N \arrow[r] & M' & \overline{N} \arrow[l] \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & E_2 \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd}$$

We are now ready to show the congruence result. The proofs of all lemmas mentioned in Theorem 3.3.10 can be found in Appendix B.1. Furthermore, some steps in the proof can be conveniently illustrated by Venn-like diagrams, which are given in Appendix B.2.

Theorem 3.3.10 (Bisimilarity with NACs is a Congruence). *Whenever $\mathcal{R}$ is a bisimulation with NACs, then $\hat{\mathcal{R}}$ is a bisimulation with NACs as well. This implies that the bisimilarity relation $\sim$ is a congruence.*

Proof. In [EK06] it was shown for the category of graph structures that bisimilarity derived from graph productions of the form $L \leftarrow I \rightarrow R$ with monos is a congruence. The pushout and pullback properties employed in [EK06] also hold for any adhesive category. Here we will extend the proof of [EK06] to handle productions with NACs in adhesive categories. All constructions used in this current proof are compliant with adhesive categories, except for parts of Lemma 3.3.7, which requires that pullbacks preserve epis (see Appendix B.1).

We will show that whenever $\mathcal{R}$ is a bisimulation, then $\hat{\mathcal{R}}$, which is the contextualization of $\mathcal{R}$, is also a bisimulation. With the following argument we can infer that $\hat{\sim} \subseteq \sim$ and that $\sim$ is a congruence: Whenever $(\overline{J} \rightarrow \overline{G}) \mathrel{\hat{\sim}} (\overline{J} \rightarrow \overline{G}')$, there exists a bisimulation $\mathcal{R}$ such that $(\overline{J} \rightarrow \overline{G}) \mathrel{\hat{\mathcal{R}}} (\overline{J} \rightarrow \overline{G}')$. Since, as we will show, $\hat{\mathcal{R}}$ is a bisimulation, it follows that $(\overline{J} \rightarrow \overline{G}) \sim (\overline{J} \rightarrow \overline{G}')$.

Let $\mathcal{R}$ be a bisimulation and let $(\overline{J} \rightarrow \overline{G}) \mathrel{\hat{\mathcal{R}}} (\overline{J} \rightarrow \overline{G}')$. That is, there is a pair $(J \rightarrow G) \mathrel{\mathcal{R}} (J \rightarrow G')$ and a context $J \rightarrow E \leftarrow \overline{J}$ such that $\overline{J} \rightarrow \overline{G}$ and $\overline{J} \rightarrow \overline{G}'$ are obtained by inserting $J \rightarrow G$ and $J \rightarrow G'$ into this context.

Let us also assume that

$$(\overline{J} \rightarrow \overline{G}) \xrightarrow{\overline{J} \rightarrow \overline{F} \leftarrow \overline{K}; \{\overline{F} \rightarrow \overline{N}_x\}_{x \in X}} (\overline{K} \rightarrow \overline{H}).$$

Our goal is to show that there exists a transition

$$(\overline{J} \rightarrow \overline{G}') \xrightarrow{\overline{J} \rightarrow \overline{F} \leftarrow \overline{K}; \{\overline{F} \rightarrow \overline{N}_x\}_{x \in X}} (\overline{K} \rightarrow \overline{H}')$$

with $(\overline{K} \rightarrow \overline{H}) \mathrel{\hat{\mathcal{R}}} (\overline{K} \rightarrow \overline{H}')$, which implies that $\hat{\mathcal{R}}$ is a bisimulation. In *Step A* we construct a transition

$$(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z \cup Z'}} (K \rightarrow H)$$

which implies a transition

$$(J \rightarrow G') \xrightarrow{J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z \cup Z'}} (K \rightarrow H')$$

with $(K \rightarrow H) \mathrel{\mathcal{R}} (K \rightarrow H')$, since $\mathcal{R}$ is a bisimulation. In *Step B* we extend the second transition to obtain the transition stated in our goal above. This argument is basically the same as in [EK06], except for the fact that here we are dealing with a bisimulation definition involving transition labels with negative borrowed contexts.

Step A: From transition $(\overline{J} \rightarrow \overline{G}) \xrightarrow{\overline{J} \rightarrow \overline{F} \leftarrow \overline{K}; \{\overline{F} \rightarrow \overline{N}_x\}_{x \in X}} (\overline{K} \rightarrow \overline{H})$ we can derive Diagram (3.7), where the decomposition of $\overline{J} \rightarrow \overline{G}$ is shown explicitly, all arrows are mono and all squares are pushouts, except for the indicated pullback.

$$\begin{tikzcd}
 & & NAC_y & & \\
 & \overline{D} \arrow[r] \arrow[d] & L \arrow[u] \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & \overline{G} \arrow[r] & \overline{G}^+ \arrow[dd, phantom, "\scriptstyle PB" near start, shift left=8ex] & \overline{C} \arrow[l] \arrow[r] & \overline{H} \\
J \arrow[u] \arrow[r] & E \arrow[u] & & & \\
 & \overline{J} \arrow[u] \arrow[r] & \overline{F} \arrow[uu] \arrow[d] & \overline{K} \arrow[l] \arrow[uu] \arrow[uur, dotted] & \\
 & & \overline{N}_x & &
\end{tikzcd}$$

(3.7)

$$\begin{tikzcd}
 & & NAC_y & & \\
 & \overline{D} \arrow[r] \arrow[d] & L \arrow[u] \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & \overline{G} \arrow[r] & \overline{G}^+ \arrow[dr, phantom, "\scriptstyle PB"] & \overline{C} \arrow[l] \arrow[r] & \overline{H} \\
J \arrow[u] \arrow[r] & E \arrow[u] \arrow[r] & E_2 \arrow[u] \arrow[dr, phantom, "\scriptstyle PB"] & E_1 \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & \overline{J} \arrow[u] \arrow[r] & \overline{F} \arrow[u] \arrow[d] & \overline{K} \arrow[l] \arrow[u] \arrow[uur, dotted] & \\
 & & \overline{N}_x & &
\end{tikzcd}$$

(3.8)

Now we will project the borrowed context diagram of $\overline{J} \to \overline{G}$ (see Diagram (3.7)) to a borrowed context diagram of $J \to G$, first without taking into account NACs. In the following we summarize how this is carried out in [EK06]. For a detailed description of this projection process we refer the reader to Theorem 4.3 in [EK06].

Applying pushout and pullback splitting in Diagram (3.7) gives rise to Diagram (3.8), where all morphisms are mono and all squares are pushouts except for the indicated pullbacks. We build the pullback $G \leftarrow D \to \overline{D}$ of $G \to \overline{G} \leftarrow \overline{D}$ and then the pushout $G \to \tilde{G} \leftarrow \overline{D}$ of $G \leftarrow D \to \overline{D}$ as shown in Diagram (3.9). From the universal property of this pushout already built we can infer that the two triangles inside the upper leftmost square commute. The other squares in Diagram (3.9) are obtained via pushout and pushout complement splitting. All morphisms in Diagram (3.9) are mono.

$$\begin{tikzcd}
 & & & & NAC_y & & & & \\
D \arrow[rr] \arrow[dd] & & \overline{D} \arrow[rr] \arrow[dl] \arrow[dd] & & L \arrow[u] \arrow[dl] \arrow[dd] & & I \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R \arrow[dl] \arrow[dd] \\
 & \tilde{G} \arrow[rr] \arrow[dr] & & G^+ \arrow[dr] & & C \arrow[ll] \arrow[rr] \arrow[dr] & & H \arrow[dr] & \\
G \arrow[rr] \arrow[ur] & & \overline{G} \arrow[rr] & & \overline{G}^+ & & \overline{C} \arrow[ll] \arrow[rr] & & \overline{H} \\
 & F_1 \arrow[uu] \arrow[dr] & & & & & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[uu] \arrow[rr] & & E_2 \arrow[uu] & & E_1 \arrow[ll] \arrow[uu] \arrow[uurr, dotted] & & \\
 & & \overline{J} \arrow[u] \arrow[rr] & & \overline{F} \arrow[u] \arrow[d] & & \overline{K} \arrow[ll] \arrow[u] \arrow[uuurr, dotted] & & \\
 & & & & \overline{N}_x & & & &
\end{tikzcd}$$

(3.9)

$$\begin{tikzcd}
 & & & & NAC_y & & & & \\
D \arrow[rr] \arrow[dd] & & \overline{D} \arrow[rr] \arrow[dl] \arrow[dd] & & L \arrow[u] \arrow[dl] \arrow[dd] & & I \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R \arrow[dl] \arrow[dd] \\
 & \tilde{G} \arrow[rr] \arrow[dr] & & G^+ \arrow[dr] & & C \arrow[ll] \arrow[rr] \arrow[dr] & & H \arrow[dr] & \\
G \arrow[rr] \arrow[ur] & & \overline{G} \arrow[rr] & & \overline{G}^+ & & \overline{C} \arrow[ll] \arrow[rr] & & \overline{H} \\
 & F_1 \arrow[uu] \arrow[dr] \arrow[rr] & & F \arrow[uu] \arrow[dr] \arrow[dd] & & K \arrow[ll] \arrow[uu] \arrow[dr] \arrow[uurr, dotted] & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[uu] \arrow[rr] & & E_2 \arrow[uu] & & E_1 \arrow[ll] \arrow[uu] \arrow[uurr, dotted] & & \\
 & & & N_z & & & & & \\
 & & \overline{J} \arrow[uu] \arrow[rr] & & \overline{F} \arrow[uu] \arrow[d] & & \overline{K} \arrow[ll] \arrow[uu] \arrow[uuuurr, dotted] & & \\
 & & & & \overline{N}_x & & & &
\end{tikzcd}$$

(3.10)

In Diagram (3.9) we build the pullbacks $G^+ \leftarrow F \rightarrow E_2$ and $C \leftarrow K \rightarrow E_1$ from $G^+ \rightarrow \overline{G}^+ \leftarrow E_2$ and $C \rightarrow \overline{C} \leftarrow E_1$, respectively. The universal property of these two pullbacks gives us the morphisms $F_1 \rightarrow F$ and $K \rightarrow F$ to close the cubes in Diagram (3.10). It turns out that all morphisms in Diagram (3.10) are mono and, furthermore, all squares are pushouts with monos except for $(D, G, \overline{D}, \overline{G})$, front and back faces of the cube on the right and $(E_2, \overline{F}, E_1, \overline{K})$ that are pullbacks.

Now we have to focus on the issues concerning the NACs in the resulting Diagram (3.10). Observe that all negative borrowed contexts $\overline{N}_x$ of the transition are obtained via Diagram (3.11). It is shown in Lemma 3.3.7 (NAC compatibility) that such a diagram can be "decomposed" into Diagrams (3.12) and (3.13), where the former shows the derivation of negative borrowed contexts for G^+. That is, every negative borrowed context of the larger object $\overline{G}^+$ is associated with at least one borrowed context of the smaller object G^+. Note that the transformation of one negative borrowed context into the other is only dependent on the context $J \rightarrow E \leftarrow \overline{J}$, into which $J \rightarrow G$ is inserted, but not on G itself, since E_2 is the pushout of $\overline{J} \rightarrow E$, $\overline{J} \rightarrow \overline{F}$. This independence of G will allow us to use this construction for $J \rightarrow G'$ in *Step B* (see Appendix B.2 for this construction as a Venn diagram).

$$\begin{tikzcd}
NAC_y \arrow[r] & \overline{M}_x & \overline{N}_x \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & \overline{G}^+ \arrow[u, "j.epi"'] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.11}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & G^+ \arrow[u, "j.epi"'] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.12}$$

$$\begin{tikzcd}
N_z \arrow[r] & M'_x & \overline{N}_x \arrow[l] \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & E_2 \arrow[u, "j.epi"'] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.13}$$

In addition there might be further negative borrowed contexts $F \rightarrow N_z$ with indices $z \in Z'$, where Z and Z' are disjoint index sets. These are exactly the negative borrowed contexts for which Diagram (3.13) can not be completed since the pushout complement does not exist. If we could complete Diagram (3.13) we would be able to reconstruct Diagram (3.11) due to Lemma 3.3.7 (NAC compatibility).

Hence we obtain a transition from $J \rightarrow G$ which satisfies Conditions (ii) and (iii) of Definition 3.3.4 (borrowed context rewriting). We still have to show that the BC step from G^+ is NAC-consistent (Condition (i)). By assumption, the BC step from $\overline{J} \rightarrow \overline{G}$ of Diagram (3.10) is NAC-consistent. So by Lemma 3.3.8 there does not exist any iso $\overline{F} \rightarrow \overline{N}_x$, which by Lemma 3.3.9 implies that no $F \rightarrow N_z$, $z \in Z$ is an iso. Furthermore, no $F \rightarrow N_z$ with $z \in Z'$ can be an iso, since otherwise we could complete Diagram (3.13). Then by Lemma 3.3.8 we conclude that the BC step from $J \rightarrow G$ is NAC-consistent.

Since all conditions of Definition 3.3.4 are satisfied, we can derive the transition $(J \to G) \xrightarrow{J\to F\leftarrow K;\{F\to N_z\}_{z\in Z\cup Z'}} (K \to H)$ from Diagram (3.10) using Definition 3.3.5 (bisimulation with NACs). Since $\mathcal{R}$ is a bisimulation, this implies $(J \to G') \xrightarrow{J\to F\leftarrow K;\{F\to N_z\}_{z\in Z\cup Z'}} (K \to H')$ with $(K \to H)\ \mathcal{R}\ (K \to H')$. Additionally we can infer from Diagram (3.10) that $\overline{K} \to \overline{H}$ is the insertion of $K \to H$ into the context $K \to E_1 \leftarrow \overline{K}$.

Step B: In *Step A* we have shown that $J \to G'$ can mimic $J \to G$ due to the bisimulation $\mathcal{R}$. Here we will show that $(\overline{J} \to \overline{G}')$ can also mimic $(\overline{J} \to \overline{G})$ since $\mathcal{R}$ is a bisimulation and both objects with interface are derived from the insertion of $J \to G$ and $J \to G'$ into the context $J \to E \leftarrow \overline{J}$.

We take the transition from $J \to G'$ to $K \to H'$ with $(K \to H)\ \mathcal{R}\ (K \to H')$ from *Step A* and construct a transition from $(\overline{J} \to \overline{G}')$ to $(\overline{K} \to \overline{H}')$ with $(\overline{K} \to \overline{H})\ \hat{\mathcal{R}}\ (\overline{K} \to \overline{H}')$. Recall that $\overline{J} \to \overline{G}'$ is $J \to G'$ in the context $J \to E \leftarrow \overline{J}$.

$$\begin{tikzcd}
 & F_1 \arrow[rr] \arrow[dr] & & F \arrow[dr] \arrow[dd] & & K \arrow[ll] \arrow[dr] & \\
J \arrow[ur] \arrow[rr] & & E \arrow[rr] & & E_2 & & E_1 \arrow[ll] \\
 & & & N_z & & & \\
 & & \overline{J} \arrow[uu] \arrow[rr] & & \overline{F} \arrow[uu] \arrow[d] & & \overline{K} \arrow[uu] \arrow[ll] \\
 & & & & \overline{N}_x & &
\end{tikzcd} \quad (3.14)$$

$$\begin{tikzcd}
 & NAC'_y & & \\
D' \arrow[r] \arrow[d] & L' \arrow[u] \arrow[d] & I' \arrow[l] \arrow[r] \arrow[d] & R' \arrow[d] \\
G' \arrow[r] & G'^{+} \arrow[dr, phantom, "PB"] & C' \arrow[l] \arrow[r] & H' \\
J \arrow[u] \arrow[r] & F \arrow[u] \arrow[d] & K \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & N_z & &
\end{tikzcd} \quad (3.15)$$

Now we concentrate on building the BC steps, first without considering the NACs. This is done by following the procedure of [EK06], which can be summarized as follows. We cut away the upper part of Diagram (3.10) and we obtain Diagram (3.14), where all squares are pushouts except for $(E_2, E_1, \overline{F}, \overline{K})$ which is a pullback. In *Step A* we obtained an induced transition from $J \to G'$, which is derived via Diagram (3.15) for some rule $L' \leftarrow I' \to R'; \{L' \to NAC'_y\}_{y\in Y}$ and all squares are pushouts with monos except for the indicated pullback. The morphism $J \to F$ is split by F_1 and so we can split the leftmost pushouts of Diagram (3.15) by pushout and pushout complement splitting, which leads to Diagram (3.16) with monos.

$$\begin{tikzcd}
 & & NAC'_y & & \\
D' \arrow[r] \arrow[d] & \overline{D}' \arrow[r] \arrow[d] & L' \arrow[u] \arrow[d] & I' \arrow[l] \arrow[r] \arrow[d] & R' \arrow[d] \\
G' \arrow[r] & \tilde{G}' \arrow[r] & G'^{+} \arrow[dr, phantom, "PB"] & C' \arrow[l] \arrow[r] & H' \\
J \arrow[u] \arrow[r] & F_1 \arrow[u] \arrow[r] & F \arrow[u] \arrow[d] & K \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & & N_z & &
\end{tikzcd} \quad (3.16)$$

Composing Diagrams (3.14) and (3.16) leads to Diagram (3.17).

$$\begin{tikzcd}
 & & & & NAC'_y & & & & \\
D' \arrow[rr] \arrow[ddd] & & \overline{D}' \arrow[rr] \arrow[dl] & & L' \arrow[u] \arrow[dl] & & I' \arrow[ll] \arrow[rr] \arrow[dl] & & R' \arrow[dl] \\
 & \tilde{G}' \arrow[rr] & & G'^{+} & & C' \arrow[ll] \arrow[rr] & & H' & \\
 & & & & & & & & \\
 & F_1 \arrow[uu] \arrow[rr] \arrow[dr] & & F \arrow[uu] \arrow[dr] \arrow[dd] & & K \arrow[ll] \arrow[uu] \arrow[dr] \arrow[uurr, dotted] & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[rr] & & E_2 & & E_1 \arrow[ll] & & \\
 & & & N_z & & & & & \\
 & & \overline{J} \arrow[uu] \arrow[rr] & & \overline{F} \arrow[uu] & & \overline{K} \arrow[uu] \arrow[ll] & &
\end{tikzcd} \tag{3.17}$$

Note: in the diagram above, G' sits in row 4 between D' and J; $D' \to G'$, $J \to G'$ and $G' \to \tilde{G}'$:

$$\begin{tikzcd}
D' \arrow[d] & \\
G' \arrow[r] & \tilde{G}' \\
J \arrow[u] &
\end{tikzcd}$$

$$\begin{tikzcd}
NAC'_y \arrow[r] \arrow[dr, phantom, "="] & M_z & N_z \arrow[l] \\
L' \arrow[u] \arrow[r] & G'^{+} \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.18}$$

$$\begin{tikzcd}
N_z \arrow[r] \arrow[dr, phantom, "="] & M'_x & \overline{N}_x \arrow[l] \\
F \arrow[u] \arrow[r] & E_2 \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.19}$$

$$\begin{tikzcd}
NAC'_y \arrow[r] \arrow[dr, phantom, "="] & \overline{M}_x & \overline{N}_x \arrow[l] \\
L' \arrow[u] \arrow[r] & \overline{G}'^{+} \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{3.20}$$

In Diagram (3.17) we build the two remaining cubes that are shown in Diagram (3.21). We construct the pushouts $\tilde{G}' \to \overline{G}' \leftarrow E$, $G'^{+} \to \overline{G}'^{+} \leftarrow E_2$, $C' \to \overline{C}' \leftarrow E_1$ and $\overline{C}' \to \overline{H}' \leftarrow H'$ of $\tilde{G}' \leftarrow F_1 \to E$, $G'^{+} \leftarrow F \to E_2$, $C' \leftarrow K \to E_1$ and $\overline{C}' \leftarrow C' \to H'$, respectively. The universal property of the pushouts gives the remaining morphisms to complete the cubes in Diagram (3.21). By applying pushout and pullback properties in Diagram (3.21) (see [EK06] for details) it turns out that all squares are pushouts with monos, except for $(D', G', \overline{D}', \overline{G}')$, front and back faces of the cube on the right and $(E_2, \overline{F}, E_1, \overline{K})$ that are pullbacks.

$$\begin{tikzcd}
 & & & & NAC'_y & & & & \\
D' \arrow[rr] \arrow[dd] & & \overline{D}' \arrow[rr] \arrow[dl] \arrow[dd] & & L' \arrow[u] \arrow[dl] \arrow[dd] & & I' \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R' \arrow[dl] \arrow[dd] \\
 & \tilde{G}' \arrow[rr] \arrow[dr] & & G'^{+} \arrow[dr] & & C' \arrow[ll] \arrow[rr] \arrow[dr] & & H' \arrow[dr] & \\
G' \arrow[ur] \arrow[rr] & & \overline{G}' \arrow[rr] & & \overline{G}'^{+} & & \overline{C}' \arrow[ll] \arrow[rr] & & \overline{H}' \\
 & F_1 \arrow[uu] \arrow[rr] \arrow[dr] & & F \arrow[uu] \arrow[dr] \arrow[dd] & & K \arrow[ll] \arrow[uu] \arrow[dr] \arrow[uurr, dotted] & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[uu] \arrow[rr] & & E_2 \arrow[uu] & & E_1 \arrow[ll] \arrow[uu] \arrow[uurr, dotted] & & \\
 & & & N_z & & & & & \\
 & & \overline{J} \arrow[uu] \arrow[rr] & & \overline{F} \arrow[uu] \arrow[d] & & \overline{K} \arrow[uu] \arrow[ll] \arrow[uuuurr, dotted] & & \\
 & & & & \overline{N}_x & & & &
\end{tikzcd} \tag{3.21}$$

In Diagram (3.21) we then construct $\{\overline{F} \to \overline{N}_x\}_{x \in X}$ as shown in Diagram (3.19). The arrows $F \to E_2 \leftarrow \overline{F}$ and $\{F \to N_z\}_{z \in Z}$ are already present in Diagram (3.21) and so we build M'_x and $\overline{N}_x$ by considering all jointly epi squares. Each $\overline{F} \rightarrow \overline{N}_x$ constructed in this way can be also derived as a negative borrowed context with Diagram (3.20) due to Lemma 3.3.7 (NAC compatibility, where L, NAC_y, G^+ and $\overline{G}^+$ should be replaced by L', NAC'_y, G'^+ and $\overline{G'}^+$, respectively). Furthermore, we will not derive additional negative borrowed contexts because the arrows $F \rightarrow N_z$ with $z \in Z'$ can not be extended to negative borrowed contexts of the full object $\overline{G}'^+$ since an appropriate Diagram (3.19) does not exist. Hence, we obtain a transition label from $\overline{J} \rightarrow \overline{G}'$ which satisfies Conditions (ii) and (iii) of Definition 3.3.4. We still have to show that the BC step from $\overline{G}'^+$ is NAC-consistent (Condition (i)).

Observe that $F \rightarrow E_2 \leftarrow \overline{F}$ of Diagram (3.13) (from *Step A*) and Diagram (3.19) are equal and do not contain any information about G or G' (see Appendix B.2). We can conclude that Diagram 3.13 and Diagram 3.19 generate the same negative borrowed contexts in both steps. Since in Diagram (3.10) there is no negative borrowed context which is an iso (due to the NAC-consistency of its BC steps), then the same holds for Diagram (3.21). By Lemma 3.3.8 we conclude that the BC step from $\overline{J} \rightarrow \overline{G}'$ is also NAC-consistent.

Finally, by Definition 3.3.5 (bisimulation) we infer that $(\overline{J} \rightarrow \overline{G}') \xrightarrow{\overline{J} \rightarrow \overline{F} \leftarrow \overline{K}; \{\overline{F} \rightarrow \overline{N}_x\}_{x \in X}} (\overline{K} \rightarrow \overline{H}')$, and since the square $(K, H', E_1, \overline{H}')$ is a pushout, $\overline{K} \rightarrow \overline{H}'$ is $K \rightarrow H'$ inserted into the context $K \rightarrow E_1 \leftarrow \overline{K}$. From earlier considerations we know that $\overline{K} \rightarrow \overline{H}$ is obtained by inserting $K \rightarrow H$ into $K \rightarrow E_1 \leftarrow \overline{K}$. Hence, we can conclude that $(\overline{K} \rightarrow \overline{H})\ \hat{\mathcal{R}}\ (\overline{K} \rightarrow \overline{H}')$ and we have achieved our goal stated at the beginning of the proof, which implies that $\hat{\mathcal{R}}$ is a bisimulation and $\sim$ is a congruence.

□

3.4 Proof Techniques for DPO-BC with NACs

We introduce here some proof techniques to speed up the bisimulation checking procedure.

By taking a closer look at Definition 3.3.5 (bisimulation) we can notice that the very same relation $\mathcal{R}$ is mentioned in the hypothesis and conclusion. Thus, in order to check a pair $(J \rightarrow G, J \rightarrow G')$ for bisimilarity we need exactly every single pair of successors, which can be derived from $(J \rightarrow G, J \rightarrow G')$, to be present in the relation $\mathcal{R}$. Furthermore, none of these pairs of successors can be discarded from $\mathcal{R}$ or have its component objects manipulated. Hence, a bisimulation relation often contains many pairs strongly related with each other, i.e., pairs whose bisimilarity may be directly inferred from the bisimilarity of other pairs.

These redundancies in $\mathcal{R}$ often make the bisimulation verification heavy and tedious. In the DPO approach objects are defined up to isomorphism, and therefore its extension to borrowed contexts should also be able to handle isomorphisms in a natural way during bisimulation checks. However, many pairs of objects with interface to be checked for bisimilarity would require infinite bisimulation relations $\mathcal{R}$, because $\mathcal{R}$ is not closed under isomorphism.

In these cases up-to techniques [San95] are of great help since they can relieve the onerous task of bisimulation proofs by reducing the size of the relation needed to define a bisimulation. They also provide the means to check bisimilarity with finite up-to relations in some cases where any bisimulation is infinite. Now we have to introduce the notion of progression (see also [San95]).

Definition 3.4.1 (**Progression with NACs**). *Let $\mathcal{R}$, $\mathcal{S}$ be relations containing pairs of objects with interfaces of the form $(J \to G, J \to G')$, where $\mathcal{R}$ is symmetric. We say that $\mathcal{R}$ progresses to $\mathcal{S}$, abbreviated by $\mathcal{R} \rightarrowtail \mathcal{S}$, if for every $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H)$$

there exists an object with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H')$$

such that $(K \to H)\,\mathcal{S}\,(K \to H')$.

According to Definition 3.3.5, a relation $\mathcal{R}$ is a bisimulation with NACs if and only if $\mathcal{R} \rightarrowtail \mathcal{R}$. Sangiorgi investigates in [San95] progressions of the form $\mathcal{R} \rightarrowtail \mathcal{F}(\mathcal{R})$, where $\mathcal{F}$ is a function on relations. This leads to the definition below, where pairs $(J \to G, J \to G')$ related by $\mathcal{R}$ evolve to pairs $(K \to H, K \to H')$ related by $\mathcal{F}(\mathcal{R})$, instead of remaining within $\mathcal{R}$.

Definition 3.4.2 (**Bisimulation up to $\mathcal{F}$ with NACs**). *Let $\mathcal{P}$ be a set of productions with NACs, $\mathcal{R}$ a symmetric relation containing pairs of objects with interfaces $(J \to G, J \to G')$ and $\mathcal{F}$ a function from relations to relations. The relation $\mathcal{R}$ is called a* bisimulation up to $\mathcal{F}$ with NACs *if, for every $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H),$$

there exists an object with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H')$$

such that $(K \to H)\ \mathcal{F}(\mathcal{R})\ (K \to H')$. In this case, $\mathcal{R} \rightarrowtail \mathcal{F}(\mathcal{R})$.

The function $\mathcal{F}$ is in charge of creating a relation with new pairs induced by related pairs in $\mathcal{R}$. This allows us to work with relations that are often much smaller than those needed to show $\mathcal{R} \rightarrowtail \mathcal{R}$. For example, in the BC framework a function $\mathcal{F}^{iso}$ (defined later on) to close a relation under isomorphisms is quite handy since it allows us to handle many infinite bisimulation relations in a finite way. In practice we do not employ $\mathcal{F}$ to create a new relation from $\mathcal{R}$, instead we only check the membership of specific pairs, i.e., we ask whether a pair $(K \to H, K \to H')$ belongs to $\mathcal{F}(\mathcal{R})$.

However, not any function $\mathcal{F}$ is suitable for an up-to technique since it might not be *sound* with respect to bisimilarity $\sim$.

Definition 3.4.3 (**Soundness**). *A function $\mathcal{F}$ on relations is* sound *if any $\mathcal{R}$ as a bisimulation up to $\mathcal{F}$, i.e., $\mathcal{R} \rightarrowtail \mathcal{F}(\mathcal{R})$, implies $\mathcal{R} \subseteq \sim$.*

Definition 3.4.4 (**Respectfulness**). *A function $\mathcal{F}$ on relations is* respectful *if whenever $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightarrowtail \mathcal{S}$, then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightarrowtail \mathcal{F}(\mathcal{S})$.*

An important property of respectful functions is that they are preserved by function composition, i.e., they can be combined to form new up-to techniques. Furthermore, respectful functions are sound.

Theorem 3.4.5. *A respectful function $\mathcal{F}$ is sound.*

Proof. See [San95]. □

In the following we define three up-to techniques for the borrowed context framework. With the first technique we avoid checking the bisimilarity of one pair of objects with interface more than once since we close $\mathcal{R}$ under isomorphism, i.e., we are able to detect if a pair already has an isomorphic counterpart in $\mathcal{R}$.

Definition 3.4.6 ($\mathcal{F}^{iso}$ **function**). *Let $\mathcal{R}$ be a symmetric relation containing pairs of objects with interfaces of the form $(J \to G, J \to G')$. The $\mathcal{F}^{iso}$ function closes $\mathcal{R}$ under isomorphisms:*

$$\mathcal{F}^{iso}(\mathcal{R}) = \{(K \to H, K \to H') \mid \exists (J \to G, J \to G') \in \mathcal{R} \text{ and there exist isomorphisms } K \xrightarrow{\sim} J,\ H \xrightarrow{\sim} G,\ H' \xrightarrow{\sim} G' \text{ such that } (1), (2) \text{ commute}\}$$

$$\begin{tikzcd} K \arrow[r] \arrow[d, "\wr"'] \arrow[dr, phantom, "(1)"] & H \arrow[d, "\wr"'] \\ J \arrow[r] & G \end{tikzcd} \qquad \begin{tikzcd} K \arrow[r] \arrow[d, "\wr"'] \arrow[dr, phantom, "(2)"] & H' \arrow[d, "\wr"'] \\ J \arrow[r] & G' \end{tikzcd}$$

A more powerful technique is called up-to context. As defined in [EK06] two objects with interface $(K \to H, K \to H')$ are bisimilar up to context if after removal of identical contexts the resulting pair of objects can be found in the relation $\mathcal{R}$.

Definition 3.4.7 ($\mathcal{F}^C$ **function**)**.** *Let $\mathcal{R}$ be a symmetric relation containing pairs of objects with interfaces of the form $(J \rightarrow G, J \rightarrow G')$. The $\mathcal{F}^C$ function closes $\mathcal{R}$ under contextualization, as in Definition 3.3.6:*

$$\mathcal{F}^C(\mathcal{R}) = \{(K \rightarrow H, K \rightarrow H') \mid \exists (J \rightarrow G, J \rightarrow G') \in \mathcal{R} \textit{ and there exists } \\ \textit{a context } J \rightarrow E \leftarrow K \textit{ inducing the diagrams below}\}$$

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & K \arrow[l] \arrow[dl, dotted] \\
G \arrow[r] & H &
\end{tikzcd}
\qquad
\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & K \arrow[l] \arrow[dl, dotted] \\
G' \arrow[r] & H' &
\end{tikzcd}$$

Observe that the up-to context technique subsumes the up-to isomorphism. However, it is an important technique in practice since its membership can often be determined faster than by using the up-to context (see Section 6.6).

Finally, we define Milner's classical up-to bisimilarity [Mil89] for borrowed contexts. The intuition of this technique is to automatically infer the bisimilarity of a pair from the bisimilarity of other pairs already present in the relation $\mathcal{R}$ by using the transitivity property of $\sim$.

Definition 3.4.8 ($\mathcal{F}^{\sim}$ **function**)**.** *Let $\mathcal{R}$ be a symmetric relation containing pairs of objects with interfaces of the form $(J \rightarrow G, J \rightarrow G')$. The $\mathcal{F}^{\sim}$ function closes $\mathcal{R}$ under bisimilarity:*

$$\mathcal{F}^{\sim}(\mathcal{R}) = \{(K \rightarrow H, K \rightarrow H') \mid \exists\, (J \rightarrow G) \sim (K \rightarrow H) \textit{ and} \\ \exists\, (J \rightarrow G') \sim (K \rightarrow H') \textit{ such that } (J \rightarrow G)\mathcal{R}(J \rightarrow G')\}$$

Remark 3.4.9. *The instantiations of Definition 3.4.2 (bisimulation up to $\mathcal{F}$ with NACs) with the functions $\mathcal{F}^{iso}$, $\mathcal{F}^C$ and $\mathcal{F}^{\sim}$ are also called bisimulation up to isomorphism with NACs, bisimulation up to context with NACs and bisimulation up to bisimilarity with NACs, respectively.*

The relation obtained by applying $\mathcal{F}^C$ to $\mathcal{R}$, i.e., $\mathcal{F}^C(\mathcal{R})$, is also denoted by $\hat{\mathcal{R}}$ (cf. Definition 3.3.6).

It can be shown that each of the techniques above is respectful, and therefore it implies bisimilarity. Moreover, they can be composed to form new techniques since respectful functions are closed under composition, e.g., from $\mathcal{F}^C$ and $\mathcal{F}^{iso}$ we obtain a respectful function $\mathcal{F}^C \circ \mathcal{F}^{iso}$.

Proposition 3.4.10 (**Bisimulation up to $\mathcal{F}$ ($\mathcal{F} = \mathcal{F}^{iso}, \mathcal{F}^C$ or $\mathcal{F}^{\sim}$) with NACs implies $\mathcal{F}$ Respectful**)**.** *Let $\mathcal{R}$ be a bisimulation up to $\mathcal{F}$ with NACs, where $\mathcal{F} = \mathcal{F}^{iso}, \mathcal{F}^C$ or $\mathcal{F}^{\sim}$. Then $\mathcal{F}$ is respectful.*

Proof.

Up-to isomorphism: Follows from the definition of objects up to isomorphism.

Up-to context: Follows easily from the proof of Theorem 3.3.10 (see also [EK06]).

Up-to bisimilarity: Follows basically from the transitivity of $\sim$ (see also [Mil89] and [San95]).

□

In Chapter 4 we extend Hirschkoff's on-the-fly bisimulation checking algorithm [Hir01] to the borrowed context setting in order to mechanize bisimulation proofs for graphs with interfaces. Since this algorithm also handles up-to techniques we give in Section 6.6 algorithms to implement the up-to isomorphism and context techniques.

Now we will investigate the feasibility of checking a pair of objects with interface for bisimilarity without having to deal with all transition labels generated by Definition 3.3.4. For productions of the form $L \leftarrow I \rightarrow R$ (without NACs) a bisimulation checking technique is proposed in [EK06] and it takes into account only certain labels. A label is considered superfluous and called *independent* if we can add two arrows $D \rightarrow J$ and $D \rightarrow I$ to the BC step diagram in Definition 3.2.3 such that $D \rightarrow I \rightarrow L = D \rightarrow L$ and $D \rightarrow J \rightarrow G = D \rightarrow G$. That is, intuitively, the object G to be rewritten and the left-hand side L overlap only in their interfaces. Any move made by an independent label need not be matched in the bisimulation game, since a matching transition is always possible (see Proposition 3.4.12). Hence, only *dependent* labels have to be checked. Dependent labels are called *engaged* in Milner's approach [Mil06]. In the following we will investigate whether this technique can be carried over to productions with NACs.

First, we repeat the relevant definition for productions without NACs.

Definition 3.4.11 ((In)Dependent Transition Labels of Productions without NACs). *Let* $(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H)$ *be a transition of* $(J \rightarrow G)$. *We say that this transition is* independent *whenever we can add two arrows* $D \rightarrow J$ *and* $D \rightarrow I$ *to the diagram in Definition 3.2.3 such that the diagram below commutes, i.e.,* $D \rightarrow I \rightarrow L = D \rightarrow L$ *and* $D \rightarrow J \rightarrow G = D \rightarrow G$. *We write* $(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K}_d (K \rightarrow H)$ *if the transition is not independent and we call it* dependent.

$$\begin{tikzcd}
D \arrow[r] \arrow[rr, bend left] \arrow[d] \arrow[dd, bend right] & L \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[u] \arrow[r] & F \arrow[u] & K \arrow[l] \arrow[u] \arrow[ur, dotted]
\end{tikzcd} \tag{3.22}$$

Let $\mathcal{R}$, $\mathcal{S}$ be relations containing pairs of objects with interfaces of the form $(J \to G, J \to G')$, where $\mathcal{R}$ is symmetric. We say that $\mathcal{R}$ d-progresses to $\mathcal{S}$, abbreviated by $\mathcal{R} \rightarrowtail_d \mathcal{S}$, if whenever $(J \to G)\,\mathcal{R}\,(J \to G')$ and $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$, there exists an object with interface $K \to H'$ such that[2] *$(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$ and $(K \to H)\,\mathcal{S}\,(K \to H')$.*

If no NACs are present, the proof technique works according to the following proposition.

Proposition 3.4.12.

(i) Let $\mathcal{R}$ be a relation with $(J \to G)\,\mathcal{R}\,(J \to G')$. Given an independent transition $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$, then there is an independent transition $(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$ via the same production and context such that $(K \to H)\,\hat{\mathcal{R}}\,(K \to H')$.

(ii) Let $\mathcal{R}$ be symmetric and let $\mathcal{R} \rightarrowtail_d \hat{\mathcal{R}}$. This implies that $\mathcal{R}$ is contained in $\sim$.

Proof. See Proposition 5.5 in [EK06]. □

Unfortunately, as we will show in the following counterexample, the proof technique based on (in)dependent labels does not carry over straightforwardly into the setting with NACs. We will give an example for a transition with an independent label for one graph that can not be simulated by its partner due to the fact that the negative application condition is satisfied for the first graph, but not for the second.

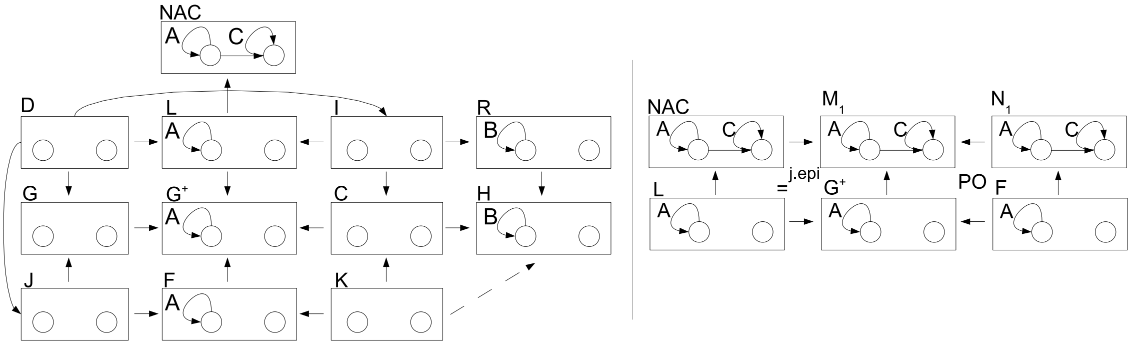


Figure 3.7: A borrowed context step from $\mathsf{J} \to \mathsf{G}$

Consider two graphs with interface: $\mathsf{J} \to \mathsf{G}$ (Figure 3.7) and $\mathsf{J} \to \mathsf{G}'$ (Figure 3.8). Above we depict how the independent label $l_i = \mathsf{J} \to \mathsf{F} \leftarrow \mathsf{K}; \{\mathsf{F} \to \mathsf{N_1}\}$ is derived from $\mathsf{J} \to \mathsf{G}$ via a graph production $\mathsf{p} = \mathsf{L} \leftarrow \mathsf{I} \to \mathsf{R}; \{\mathsf{L} \to \mathsf{NAC}\}$ using Definition 3.3.4.

[2] Note that $J \to G'$ may answer with an independent transition label.

Recall that if p was a rule without NAC then the same transition label l_i would be induced for $\mathsf{J} \rightarrow \mathsf{G}'$ and it would not be necessary to consider these labels in the bisimulation checking procedure (see (i) in Proposition 3.4.12). However, the presence of a NAC in p restricts the applicability of this proof technique since the transition with label l_i of $\mathsf{J} \rightarrow \mathsf{G}'$ is only feasible if the BC step is NAC consistent (see Condition (i) of Definition 3.3.4). Figure 3.8 illustrates the induced BC step from $\mathsf{J} \rightarrow \mathsf{G}'$ via p. This BC step is not NAC consistent since G'^{+} contains NAC. However, for completeness we still list all negative borrowed contexts (on the right).

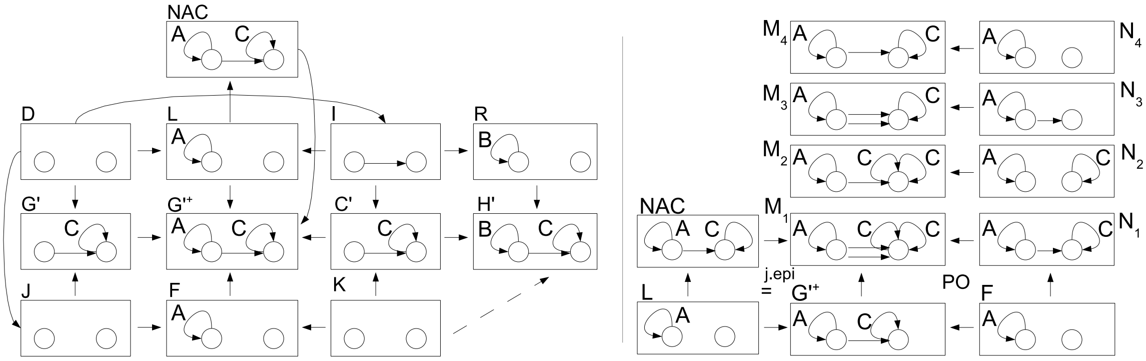


Figure 3.8: A borrowed context step from $\mathsf{J} \rightarrow \mathsf{G}'$

Note that in other cases the BC step of the second graph could be feasible, but with different negative borrowed contexts. Again, in this case the two steps would not properly match each other and the two graphs would not be bisimilar.

Due to the problems with NAC consistency discussed above Proposition 3.4.12 (for rules without NACs) has a much weaker counterpart in the setting with NACs. In Proposition 3.4.13 we consider independent and dependent transition labels for borrowed context steps with NACs analogously as for BC steps without NACs (see Definition 3.4.11), i.e., a label $l = J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z}$ is independent whenever there exist arrows $D \rightarrow I$ and $D \rightarrow J$ with the required commutativity; otherwise l is dependent.

Proposition 3.4.13.

(i) *Let $\mathcal{R}$ be a relation with $(J \rightarrow G)\,\mathcal{R}\,(J \rightarrow G')$. Given an independent transition $t = (J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z}} (K \rightarrow H)$, then there is an independent transition $t' = (J \rightarrow G') \xrightarrow{J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z}} (K \rightarrow H')$ via the same production with NACs and context with $(K \rightarrow H)\,\hat{\mathcal{R}}\,(K \rightarrow H')$ whenever Condition (iii) of Definition 3.3.4 yields for t' a set $\{F \rightarrow N_z\}_{z \in Z}$ which is isomorphic to the set in t.*

(ii) Let $\mathcal{R}$ be symmetric and let $\mathcal{R} \rightarrowtail_x \hat{\mathcal{R}}$, where x is either an independent label which does not satisfy (i) or a dependent label. This implies that $\mathcal{R}$ is contained in $\sim$.

Proof. This follows directly from Proposition 3.4.12 with support of Lemma 3.3.8 to ensure NAC consistency for the BC step from $J \to G'$. □

As we have shown, the proof technique based on independent labels does not work in general for rules with NACs. On the other hand, we can still use Proposition 3.4.13 to improve efficiency of the bisimulation checking procedure. This works as follows: whenever we derive an independent label from $J \to G$ via a production with NACs, we can construct the same borrowed context diagram for $J \to G'$ and it only remains to show that in both cases the same set of negative borrowed contexts is produced. Then for sure the BC step from $J \to G'$ is NAC consistent and both independent labels are suitable matched. Furthermore, in this case it is not necessary to check whether the pair of successors is contained in the bisimulation relation since both can be obtained by inserting $J \to G$, $J \to G'$ into the same context (this is analogous to the corresponding proof of Proposition 3.4.12). This simplification will often lead to smaller bisimulations.

We illustrate this efficiency improvement for our next example.

3.5 Example 1: Servers as Graphs with Interfaces

Here we apply the DPO-BC extension to NACs in order to check the bisimilarity of two graphs with interfaces $\mathsf{J}_1 \to \mathsf{G}_1$ and $\mathsf{J}_1 \to \mathsf{G}_2$ (shown in Figure 3.9 to the right) with respect to rule_1 and rule_2 of Section 3.3.1. Here G_1 contains only one server, whereas G_2 contains two servers which may work in parallel.

On the left side of Figure 3.9 it is shown a transition derivation for $\mathsf{J}_1 \to \mathsf{G}_1$ (which contains only one server) via rule_2 according to Definition 3.3.4. There is no mono $\mathsf{NAC}_1 \to \mathsf{G}_1^+$ forbidding the BC rewriting (Condition (i)), and hence the step is NAC consistent. The graph C_1 and additional monos lead to the BC step (Condition (ii)). The construction of the negative borrowed context $\mathsf{F}_1 \to \mathsf{N}_1$ from $\mathsf{NAC}_1 \leftarrow \mathsf{L}_2 \to \mathsf{G}_1^+ \leftarrow \mathsf{F}_1$, as specified in Condition (iii), is shown on the right. Here the graph M_1 is the only possible overlap of NAC_1 and G_1^+ such that the square with indicated jointly epi monos commutes. Since the pushout complement $\mathsf{F}_1 \to \mathsf{N}_1 \to \mathsf{M}_1$ exists, G_1^+ can be indeed extended to M_1 by gluing N_1 via F_1. All three conditions of Definition 3.3.4 are satisfied and so the BC step above with $\mathsf{label} = \mathsf{J}_1 \to \mathsf{F}_1 \leftarrow \mathsf{J}_1; \{\mathsf{F}_1 \to \mathsf{N}_1\}$ is feasible. This transition can be interpreted as follows: the environment provides G_1 with a T-task in Q_2 (see borrowed context F_1)

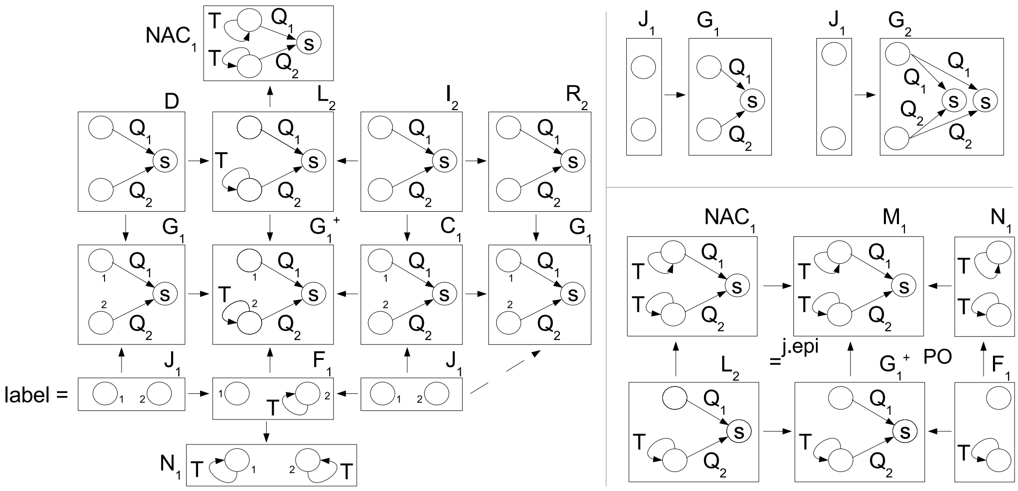


Figure 3.9: Servers $J_1 \rightarrow G_1$ and $J_1 \rightarrow G_2$ and a borrowed context step from $J_1 \rightarrow G_1$

in order to enable the BC step, but the rewriting is only possible if no T-task is waiting in queue Q_1 (see N_1). It can be shown that $J_1 \rightarrow G_2$ can do a matching step with the same label.

Analogously, we can derive other transitions from $J_1 \rightarrow G_1$ and $J_1 \rightarrow G_2$, where the labels generated via rule_1 (without NAC) do not have any negative borrowed context. In Figure 3.10 we depict schematically the resulting labeled transition systems (LTS), for which we have already shown the derivation of label for $J_1 \rightarrow G_1$. In each LTS the labels on the left are derived via rule_1 and the labels on the right via rule_2. The label l' results from rule_1 with a maximal overlap of the graph (G_1 or G_2) and the left-hand side (similar to label). Both LTSs have several transitions pointing downwards (labelled l'',l''', etc.), which are derived with partial matches smaller than the matches of the loops. In fact, the labels in both LTSs are the same and the resulting states can be matched. So $J_1 \rightarrow G_1$, $J_1 \rightarrow G_2$ and all their successors can be matched via a bisimulation and we conclude that $(J_1 \rightarrow G_1) \sim (J_1 \rightarrow G_2)$.

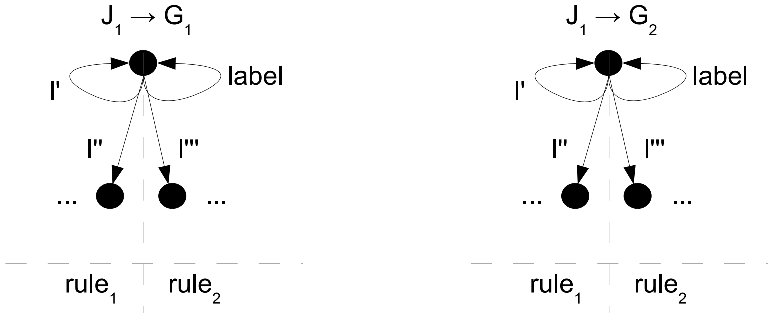


Figure 3.10: Labeled transition systems for $J_1 \rightarrow G_1$ and $J_1 \rightarrow G_2$

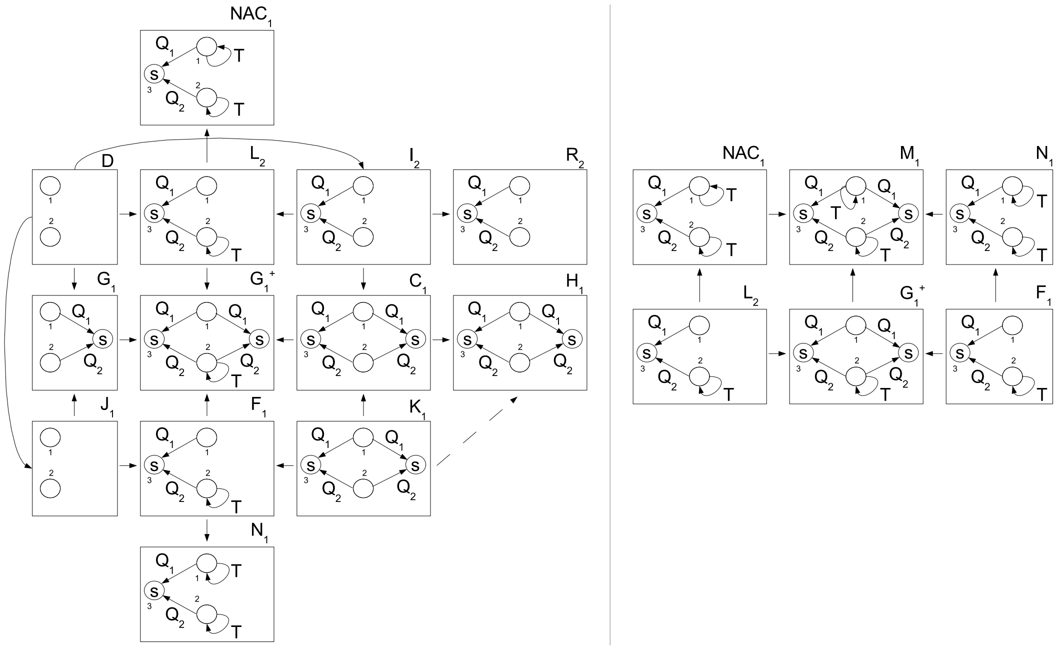


Figure 3.11: Example of independent label derivation from $\mathsf{J}_1 \to \mathsf{G}_1$

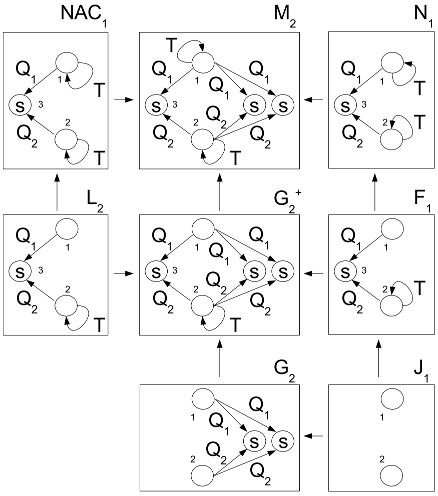
Note that in order to obtain an extended example, we could add a rule modeling the processing of tasks waiting in queue Q_1.

In the following we illustrate how independent labels can be used to improve the efficiency of the bisimulation checking procedure. Consider the servers $\mathsf{J}_1 \to \mathsf{G}_1$ and $\mathsf{J}_1 \to \mathsf{G}_2$ of Figure 3.9. In Figure 3.11 we derive the label $\mathsf{J}_1 \to \mathsf{F}_1 \leftarrow \mathsf{K}_1; \{\mathsf{F}_1 \to \mathsf{N}_1\}$ via rule_2 according to Definition 3.3.4. This transition label is independent w.r.t. Definition 3.4.11 since there exist $\mathsf{D} \to \mathsf{J}_1$ and $\mathsf{D} \to \mathsf{I}_2$ such that $\mathsf{D} \to \mathsf{G}_1 = \mathsf{D} \to \mathsf{J}_1 \to \mathsf{G}_1$ and $\mathsf{D} \to \mathsf{L}_2 = \mathsf{D} \to \mathsf{I}_2 \to \mathsf{L}_2$. In this case the environment provides G with the entire left-hand side of the rule (see F_1).

Since the label above is independent it induces the derivation of an independent label $\mathsf{J}_1 \to \mathsf{F}_1 \leftarrow \mathsf{K}_1$ (without negative borrowed contexts) for the transition of $\mathsf{J}_1 \to \mathsf{G}_2$ via rule_2 (compare with Proposition 3.4.12 for productions without NACs). However, rule_2 has a NAC and so we have to check an additional requirement, namely that the negative borrowed contexts are the same for both independent transition labels (see Proposition 3.4.13). Whenever this extra requirement is satisfied the independent transition label $\mathsf{J}_1 \to \mathsf{F}_1 \leftarrow \mathsf{K}_1; \{\mathsf{F}_1 \to \mathsf{N}_1\}$ is the same for both BC steps and the BC step of $\mathsf{J}_1 \to \mathsf{G}_2$ is NAC consistent (due to Lemma 3.3.8).

We describe how this condition is checked. In Diagram (3.23) the morphisms $NAC_1 \leftarrow L_2 \to F_1 \leftarrow J_1$ and $J_1 \to G_2$ are already known. The morphisms $L_2 \to F_1 \leftarrow J_1$ stem from the BC step of $\mathsf{J}_1 \to \mathsf{G}_1$. We then construct the pushout square in the second row and $L_2 \to G_2^+$ is the composition of $L_2 \to F_1$ and $F_1 \to G_2^+$. The construction of the upper part of the diagram proceeds as in Definition 3.3.4 (BC rewriting with NACs). On the right we depict this diagram for our current example. Note that NAC_1 is not present in G_2^+ and the negative borrowed context $\mathsf{F}_1 \to \mathsf{N}_1$ obtained below is the same as for the independent label of $\mathsf{J}_1 \to \mathsf{G}_1$. In this case, the induced BC step of $\mathsf{J}_1 \to \mathsf{G}_2$ is NAC consistent and provides a matching label for the transition of $\mathsf{J}_1 \to \mathsf{G}_1$. Furthermore, we do not have to check whether the pair of successors is contained in the bisimulation relation since both can be obtained by inserting $\mathsf{J} \to \mathsf{G}_1$, $\mathsf{J} \to \mathsf{G}_2$ into the same context.

$$\begin{tikzcd}
NAC_1 \arrow[r] & M_2 & N_1 \arrow[l] \\
L_2 \arrow[u] \arrow[r] \arrow[rr, bend right] \arrow[ur, phantom, "\overset{j.epi}{=}"] & G_2^+ \arrow[u] \arrow[ur, phantom, "PO"] & F_1 \arrow[l] \arrow[u] \\
& G_2 \arrow[u] \arrow[ur, phantom, "PO"] & J_1 \arrow[l] \arrow[u]
\end{tikzcd} \qquad (3.23)$$



3.6 Example 2: Blade Servers

In this section we present a more elaborate example in terms of blade server systems. A blade server is a server chassis which has multiple circuit boards, known as blade units. Each blade unit is a server in its own right with components such as processors, memory and local disk storage. These systems are flexible and modular since their processing power can be extended by just adding blade units.

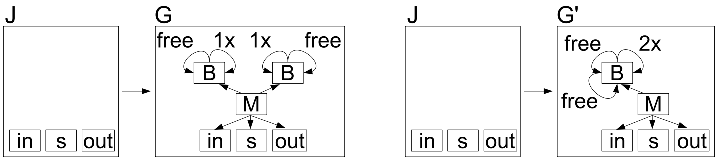


Figure 3.12: Two blade serves as graphs with interface.

Figure 3.12 illustrates two blade servers. Each server chassis (M-labeled node) has three ports: input (in-node), output (out-node) and status (s-node). The input receives commands from external systems (not modeled here) which are executed by a blade. The output makes the result of command executions available to external systems. The status indicates if a blade server is busy and cannot handle any further request. The external systems have only access to these three ports (see interfaces J). Each blade (B-node) performs command executions independently from other blades. Single-processed blades (marked with a 1x-loop) perform one command execution at a time, while double processed blades (depicted with a 2x-loop) perform up to two commands.

The operational semantics for our blade servers, which is not intended to be comprehensive, is given by the rules in Figure 3.13. The NACs of all rules, except for Update-Status2, are depicted as extensions of left-sides L, i.e., a full NAC is $\mathsf{L} \cup \mathsf{NAC_i}$. Read-in shows how a command (cmd-node) is read by an available blade (indicated by a free-loop). The free-loops on each blade specify its processing power currently available. To improve efficiency, each blade handles incoming command requests simultaneously. The NACs of Read-in restrict a command to be read only when the blade server is not busy ($\mathsf{NAC_1}$) and not fully loaded ($\mathsf{NAC_2}$ and $\mathsf{NAC_3}$).

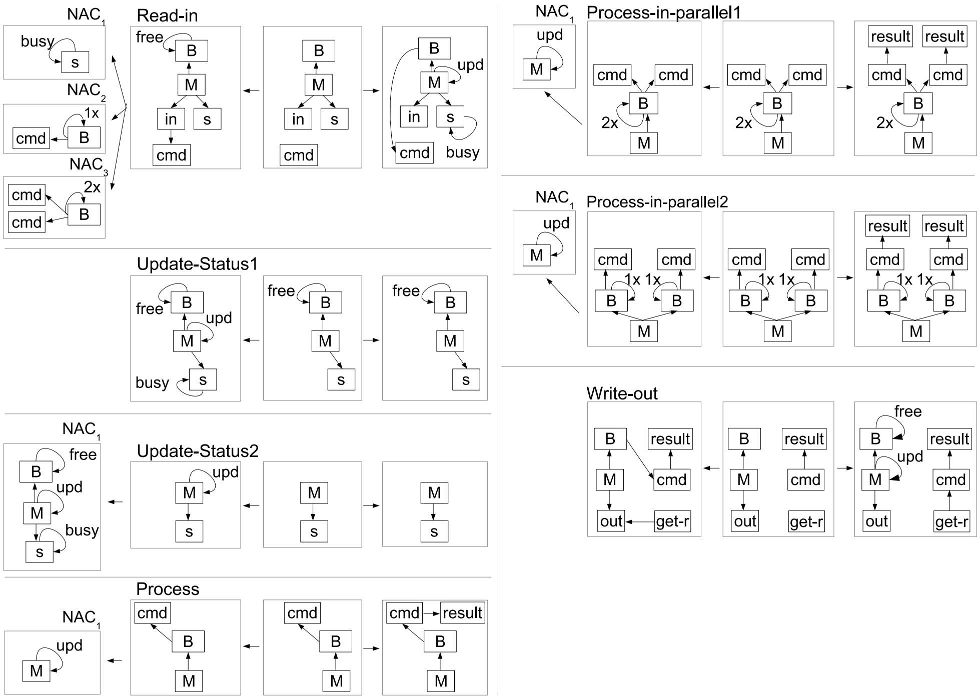


Figure 3.13: Operational semantics rules for blade serves.

Update-Status1 and Update-Status2 update the current status of a server. When a command is read we set the status to busy and force updating. Update-Status1 verifies if there is a free blade so that the busy-flag can be removed. The rule Update-Status2 checks if the server is not fully loaded. Process executes commands if the server is not currently updating (see $\mathsf{NAC_1}$). Process-in-parallel1 allows a double capacity blade to execute two commands in one single step. Process-in-parallel2 specifies that two single capacity blades can execute their commands in parallel. Finally, Write-out returns the result, sets the blade as available (free-loop) and flags the blade server to update its status.

From $\mathsf{J} \rightarrow \mathsf{G}$ and $\mathsf{J} \rightarrow \mathsf{G}'$ and the operational semantics rules we derive the labeled transition system (LTS) of Figure 3.14 w.r.t. to Definition 3.3.4. Black circles are states and arrows are annotated with labels and rule names. In our example, for each state the processing capability of a free blade is represented as “_”. In the following C abbreviates “command” and R stands for “result”. So when a blade reads a command, then “_” becomes “C” and when the command is executed, then “C” becomes “R”. Since graphs are considered up to isomorphism “C_” (“R_”) represents the same system state as “_C” (“_R”). If the start state is $\mathsf{J} \rightarrow \mathsf{G}$ ($\mathsf{J} \rightarrow \mathsf{G}'$) then Process-in-parallel1 (Process-in-parallel2) is applied to derive the transition between the “CC” and “RR” states. Labels with the same index (e.g. l_2) are equal. Figure 3.15 shows the remaining labels ($\mathsf{l}_2, \mathsf{l}_3, \mathsf{l}_5, \mathsf{l}_6, \mathsf{l}_7$ and l_8) which are associated with empty sets of negative borrowed contexts.

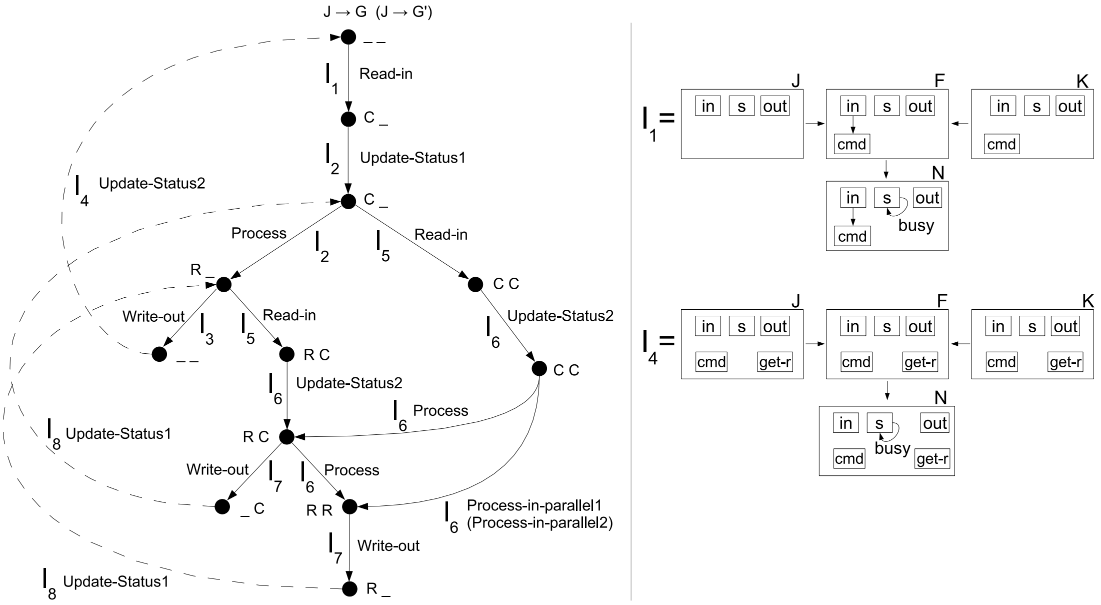


Figure 3.14: Labeled transition system for $\mathsf{J} \rightarrow \mathsf{G}$ ($\mathsf{J} \rightarrow \mathsf{G}'$).

The LTS of Figure 3.14 is a simplification since the dashed arrows point to graphs that are basically $\mathsf{J} \rightarrow \mathsf{G}$ ($\mathsf{J} \rightarrow \mathsf{G}'$) plus some extra elements to represent the results of processed commands (which are the same in both cases). This issue is automatically handled by the up-to context technique given in Definition 3.4.7. For example, consider the dashed transition l_4: whenever we perform l_4 in the LTS of $\mathsf{J} \rightarrow \mathsf{G}$ and also in the LTS of $\mathsf{J} \rightarrow \mathsf{G}'$ they properly match each other, and furthermore, the pair of resulting graphs can be found in the bisimulation relation after removal of identical

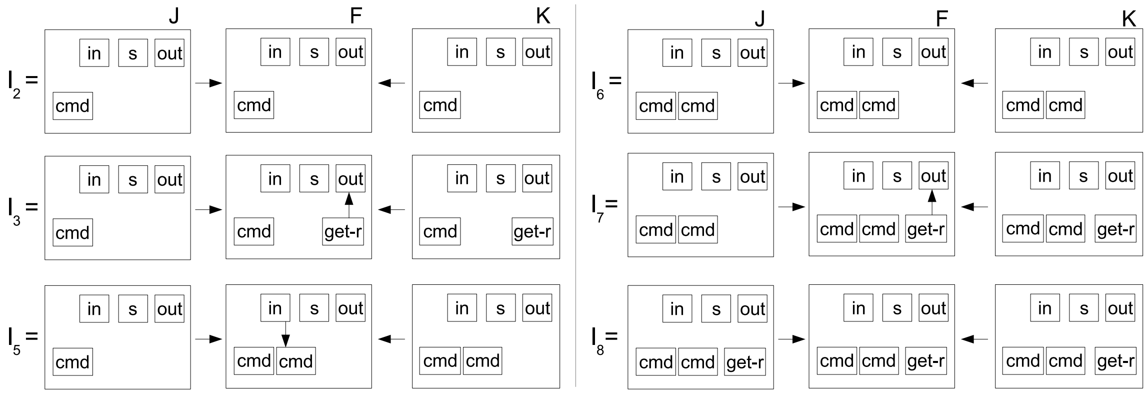


Figure 3.15: Remaining transition labels of Figure 3.14.

contexts, i.e., after peeling off the results of processed commands generated through the paths $\bullet \xrightarrow{\mathsf{l}_1} \bullet \xrightarrow{\mathsf{l}_2} \bullet \xrightarrow{\mathsf{l}_2} \bullet \xrightarrow{\mathsf{l}_3} \bullet$ (see leftmost loop in Figure 3.14). Compare the blade server $\mathsf{J} \to \mathsf{G}$ with the resulting server obtained from the derivation of label l_4 in Figure 3.17.

We illustrate two borrowed context steps for the LTS of the blade server $\mathsf{J} \to \mathsf{G}$. Figure 3.16 depicts the derivation of label l_1. On the right we show how the negative borrowed context is generated. Observe that $\mathsf{L} \to \mathsf{NAC}_2$ and $\mathsf{L} \to \mathsf{NAC}_3$ of Read-in do not yield any negative borrowed contexts.

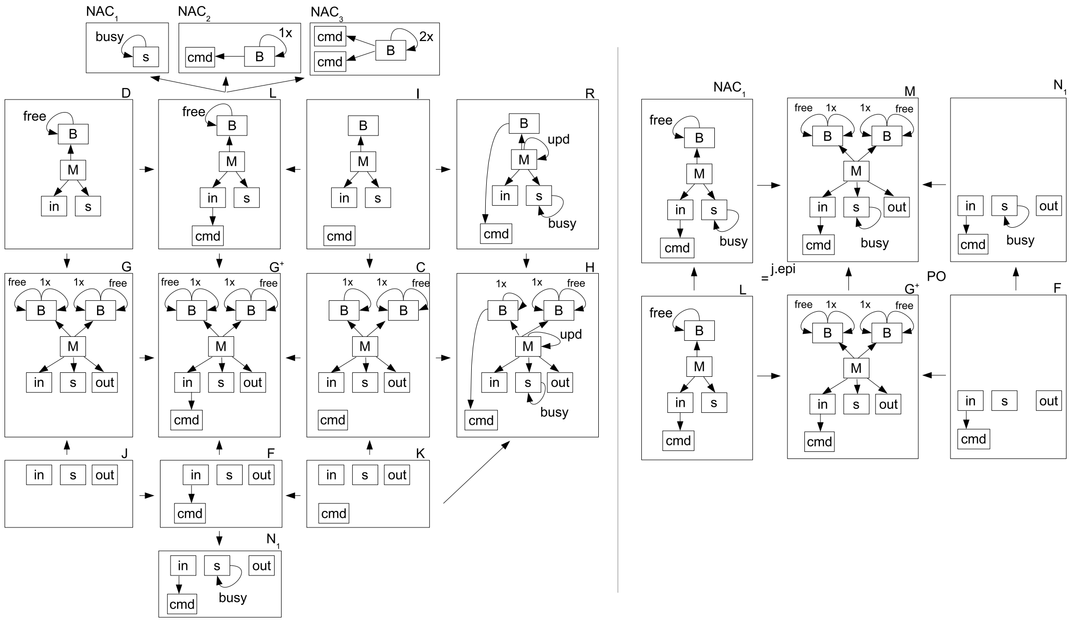


Figure 3.16: Label derivation – label l_1

Figure 3.17 shows the derivation of l_4 via Update-Status2.

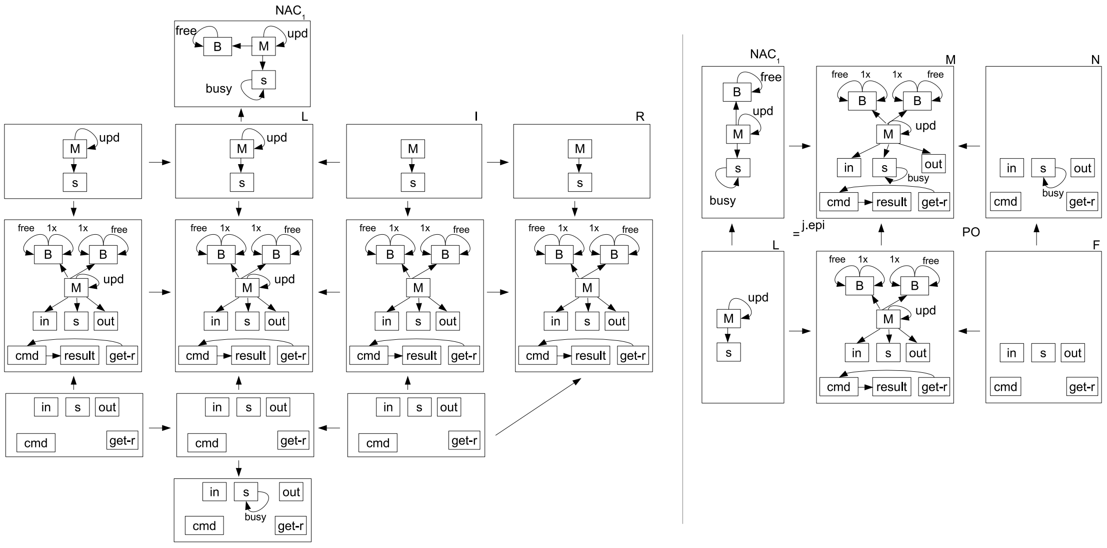


Figure 3.17: Label derivation – label l_4

Blade servers are very flexible in terms of adding and removing blades to cover non-constant demands in terms of computational power. In our example even though each blade processes commands independently from other blades, we had to make this explicit for two single-processor blades inside the same blade server. If we had not defined Process-in-parallel2, it would not have been possible to show $(\mathsf{J} \to \mathsf{G}) \sim (\mathsf{J} \to \mathsf{G}')$. This problem might be solvable by using parallel rule applications, but this might lead to problems with non-injective matches. Furthermore, we would get a different notion of behavioral equivalence that is reminiscent of step bisimilarity.

3.7 Conclusions and Future Work

We have shown how rules with NACs should be handled in the DPO with borrowed contexts and proved that the derived bisimilarity relation is a congruence. This extension to NACs is relevant for the specification of several kinds of non-trivial systems, where complex conditions play an important role. They are also frequently used when specifying model transformation, such as transformations of UML models. Behavior preservation is an important issue for model transformation.

Here we have obtained a finer congruence than the usual one. Instead, if one would reduce the number of possible contexts (for instance by forbidding contexts that contain certain patterns or subobjects), we would obtain coarser congruences, i.e., more objects would be equivalent. Studying such congruences will be a direction of future work.

Furthermore, a natural question to ask is whether there are other extensions to the

DPO approach that, when carried over to the DPO-BC framework, would require the modification of transition labels. One such candidate are generalized application conditions, so-called graph conditions [Ren04], which are equivalent to first-order logic and of which NACs are a special case. Such conditions would lead to fairly complex labels.

As a straightforward consequence of a technique based on initial pushouts proposed in [BGK06a] we defined the notion of gluing condition for borrowed context rewriting, where one can easily check whether a given partial match guarantees the existence of the pushout complements required to perform a label derivation.

Due to the fact that the bisimulation checking procedure is time consuming and error-prone when done by hand, we extend in the next chapter the on-the-fly bisimulation checking algorithm, defined in [RKE07, Hir01], to handle productions with NACs. However, in order to do this efficiently we still need further speed-up techniques such as additional up-to techniques and methods for downsizing the transition system, such as the elimination of independent labels. We discussed in Section 3.4 that the proof technique eliminating independent labels as in [EK04, EK06] (or non-engaged labels as they are called in [Mil06]) does not carry over straightforwardly from the case without NACs, but that it can still be useful. This needs to be studied further.

Some open questions remain for the moment. First, in the categorical setting it would be good to know whether pullbacks always preserve epis in adhesive categories. This question is currently open, as far as we know. Second, it is unclear where the congruence is located in the lattice of congruences that respect rewriting steps with NACs. As for IPO bisimilarity it is probably not the coarsest such congruence, since saturated bisimilarity is in general coarser [BKM06]. So it would be desirable to characterize such a congruence in terms of barbs [RSS07].

Finally, it is not clear to us at the moment how NACs could be integrated directly into reactive systems and how the corresponding notion of IPO would look like. In our opinion this would lead to fairly complex notions, for instance one would have to establish a concept similar to that of jointly epi arrows.

Chapter 4

Bisimulation Verification

4.1 Motivation

Model transformation [MG06] concerns the automatic generation of models from other models according to a transformation definition, which describes how a model in the source language can be transformed into a model in the target language. Such transformations can take place between different models or, more specifically, inside one single model (refactoring). Software refactoring is a modern software development activity to cope with the internal modification of source code to improve system quality, without changing the observable behavior. In extreme programming [BF01] refactoring belongs to the software development cycle and developers alternate between adding new functionality and refactoring the code to improve its internal consistency, reusability, flexibility and clarity.

Graph transformation systems (GTS) are well-suited to model not only refactorings but also model transformation (see [MT04] for the correspondence between refactoring and GTS). A GTS specifies model transformation by defining graph transformation rules to translate one model into another. The general idea is to have a graph describing an instance of the source model as a start graph and to apply graph productions until no further production can be applied and the resulting graph is an instance of the target model. Model transformations via GTS can be found in [EE06, EW06, VVGE$^+$06]. A crucial question that must be asked is whether a given refactoring (or model transformation) is behavior-preserving, which means that transforming one model into another model does not change the original external behavior. In practice, the proof of behavior-preserving transformations is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. In a recent paper Narayanan and Karsai [NK06] proposed a method for checking bisimilarity in model transformations using GTS which is similar to ours. The new contribution of our work here is to present an efficient bisimulation checking algorithm, which works on the fly for infinite state spaces, and to develop the theory in the very general framework of borrowed contexts [EK06, RKE08a].

The work presented in this chapter extends the results of our paper [RKE07] to handle graph rules with negative application conditions, as in Chapter 3.

We give a formal treatment of the question of behavior-preserving refactoring. We define model refactoring by graph transformation rules in the Double Pushout Approach (DPO) [CMR$^+$97, EEPT06], which is one of the standards for GTS. Our goal is to show that instances of one model are bisimilar to their refactored counterparts, which implies behavior preservation. We employ the extension of DPO to borrowed contexts [EK06, RKE08a], which provides the means to reason about bisimilarity. We also extend Hirschkoff's [Hir01] on-the-fly bisimulation checking algorithm to deal with our setting [EK06, RKE08a]. A case study of refactoring is presented in terms of minimization of deterministic finite automata (*DFA*), where we can test if a given *DFA* is bisimilar to its minimal refactored version. Another case study concerning the flattening of hierarchical statecharts is also given.

4.2 Double-Pushout with Borrowed Contexts

Here we recall the DPO extension to borrowed contexts (BC) already presented in Section 3.3 for rules with negative application conditions. While in Chapter 3 the BC machinery is defined upon the general framework of adhesive categories [LS05], in this chapter it is tailored to the category of labeled graphs[1]. Those already familiar with Section 3.3 may safely skip this section and go directly to Section 4.3, where we present algorithms for the mechanization of bisimulation proofs.

Definition 4.2.1 (Graph and Graph Morphism). *A* graph $G = (V, E, s, t, l_v, l_e)$ *consists of a set V of nodes, a set E of edges, two functions $s, t\colon E \to V$ (source and target) and two labeling functions for nodes and edges $l_v\colon V \to \Omega_V$, $l_e\colon E \to \Omega_E$, where Ω_V and Ω_E are node and edge labels. A* graph morphism $f\colon G_1 \to G_2$ *is a pair of functions $f = (f_E\colon E_1 \to E_2,\ f_V\colon V_1 \to V_2)$, which is compatible with source, target and labeling functions of G_1 and G_2, i.e., $f_V \circ s_1 = s_2 \circ f_E$, $f_V \circ t_1 = t_2 \circ f_E$, $l_{e_2} \circ f_E = l_{e_1}$ and $l_{v_2} \circ f_V = l_{v_1}$.*

In the standard DPO approach, graph productions rewrite graphs with no interaction with any other entity than the graph itself and the production. In the DPO with

[1] The category of labeled graphs is called **Graphs** and is defined in Appendix A. Moreover, it is an instance of an adhesive category [EEPT06].

borrowed contexts [EK06, RKE08a] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

Definition 4.2.2 (**Graphs with Interfaces and Graph Contexts**). *A* graph G with interface J *is a morphism* $J \to G$ *and a context consists of two morphisms* $J \to E \leftarrow \overline{J}$. *The* embedding *of a graph with interface* $J \to G$ *into a context* $J \to E \leftarrow \overline{J}$ *is a graph with interface* $\overline{J} \to \overline{G}$ *which is obtained by constructing* $\overline{G}$ *as the pushout of* $J \to G$ *and* $J \to E$ *(see diagram below).*

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[dl, dotted] \\
G \arrow[r] & \overline{G} &
\end{tikzcd}$$

The embedding is defined up to isomorphism since the pushout object is unique up to isomorphism. Moreover, embedding/insertion into a context and contextualization are used as synonyms.

We consider graph rules with negative application conditions, as in Chapter 3. Below we define when a borrowed context step is NAC consistent, i.e., the NACs of a rule do not forbid the BC step.

Definition 4.2.3 (**NAC-Consistent Borrowed Context Step**). *Assume that all morphisms are injective. Given* $J \to G$ *and a production* $p\colon L \leftarrow I \to R; \{n_y\colon L \to NAC_y\}_{y \in Y}$ *we say that a partial match* $pm\colon G \leftarrow D \to L$ *leads to a* NAC consistent BC step *with respect to* $J \to G$ *and* p *if for the pushout* G^+ *in the diagram below there is no* $q_y\colon NAC_y \to G^+$ *with* $m = q_y \circ n_y$ *for every* $y \in Y$.

$$\begin{tikzcd}
 & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[r, "n_y"] \arrow[d, "m"'] \arrow[dr, phantom, "="] & NAC_y \arrow[dl, "|" marking, "q_y"] \\
J \arrow[r] & G \arrow[r] & G^+ &
\end{tikzcd}$$

We need the concept of a pair of jointly surjective morphisms in order to "cover" a graph with two other graphs. That is needed to find possible overlaps between the NACs and the graph G^+ which includes the borrowed context.

Definition 4.2.4 (**Jointly Surjective Morphisms**). *Two morphisms* $f\colon A \to B$ *and* $g\colon C \to B$ *are* jointly surjective *whenever for every pair of morphisms* $a, b\colon B \to D$ *such that* $a \circ f = b \circ f$ *and* $a \circ g = b \circ g$ *it holds that* $a = b$.

In a pushout square the generated morphisms are always jointly surjective. This is a straightforward consequence of the uniqueness of the mediating morphism.

Definition 4.2.5 (Borrowed Context Rewriting for Rules with NACs). *Given $J \to G$, a production $L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ and a partial match $G \leftarrow D \to L$, we say that $J \to G$ reduces to $K \to H$ with transition label $J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}$ if the following holds:*

(i) the BC step is NAC consistent (as in Definition 4.2.3);

(ii) there exist graphs G^+, C and additional injective morphisms such that Diagram (4.1) *below commutes and the squares are either pushouts (PO) or pullbacks (PB);*

(iii) the set $\{F \to N_z\}_{z \in Z}$ contains exactly the morphisms constructed via Diagram (4.2) (where all morphisms are injective). (That is, there exists a graph M_z such that all squares commute and are pushouts or morphisms are jointly surjective as indicated.)

$$\begin{tikzcd}
 & NAC_y & & \\
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[u, "n_y"'] \arrow[d, "m"] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] \arrow[dr, phantom, "PO"] & G^+ \arrow[dr, phantom, "PB"] & C \arrow[l] \arrow[r] & H \\
J \arrow[u] \arrow[r] & F \arrow[u] \arrow[d] & K \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & N_z & &
\end{tikzcd} \tag{4.1}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u, "n_y"] \arrow[r, "m"'] \arrow[ur, phantom, "\overset{j.surj.}{=}"] & G^+ \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd} \tag{4.2}$$

In this case a borrowed context step *(BC step) is feasible and we write:* $(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H)$.

We explain the construction of BC steps in two situations: the simplest case is when the rule does not have NACs, whereas in the presence of NACs we have to handle additional conditions.

When no NACs are present in Definition 4.2.5 Condition (ii) does the job, i.e., we can safely discard the other two conditions. In this case consider Diagram (4.1). The upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 4.2.5, namely C and F, may not exist. In this case, the rewriting step is not feasible. Due to the absence of NACs Diagram (4.1) produces no $F \to N_z$.

By taking NACs into account, a BC step can only be executed when G^+ contains no forbidden structure of any negative application condition NAC_y at the match of L (Condition (i)). Additionally, enriched labels are generated (Condition (iii)). The morphisms $F \to N_z$ in Condition (iii) are also called *negative borrowed contexts* and each N_z represents the structures that should not be in G^+ in order to enable the BC step. This extra information in the label is of fundamental importance for the bisimulation game with NACs (Definition 4.2.6), where two graphs with interfaces must not only agree on the borrowed context which enables a transition but also on what should not be offered by the environment in order to perform the transition. The negative borrowed contexts $F \to N_z$ are obtained from $NAC_y \overset{n_y}{\leftarrow} L \overset{m}{\to} G^+ \leftarrow F$ of Diagram (4.1) via Diagram (4.2), where we create all possible overlaps M_z of G^+ and NAC_y in order to check which structures the environment should not provide in order to assure a NAC-consistent BC step. To consider all possible overlaps is necessary in order to take into account that parts of the NAC might already be present in the graph which is being rewritten. Due to the non-uniqueness of the jointly-surjective square one single negative application condition NAC_y may produce more than one negative borrowed context $F \to N_z$.

Whenever the pushout complement in Diagram (4.2) exists, the graph G^+ with borrowed context can be extended to M_z by attaching the negative borrowed context N_z via F. When the pushout complement does not exist, some parts of G^+ which are needed to perform the extension are not "visible" from the environment and no negative borrowed context is generated.

Now we show how transition labels are used to check bisimilarity between two graphs with interface. A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 4.2.6 (Bisimulation and Bisimilarity with NACs). *Let $\mathcal{P}$ be a set of productions with NACs and $\mathcal{R}$ a symmetric relation containing pairs of graphs with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a* bisimulation with NACs *if, for every $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H),$$

there exists a graph with interface $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H')$$

such that $(K \to H) \, \mathcal{R} \, (K \to H')$.

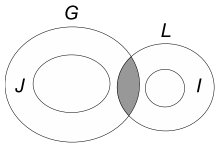
We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two graphs with interface. The relation $\sim$ is called bisimilarity with NACs*.*

We often drop "with NACs" from bisimulation (bisimilarity) when it is clear from context.

Not all labels derived from a graph and a set of productions are relevant for the bisimulation. Whenever a rule has no NACs we can distinguish two kinds of transition labels.

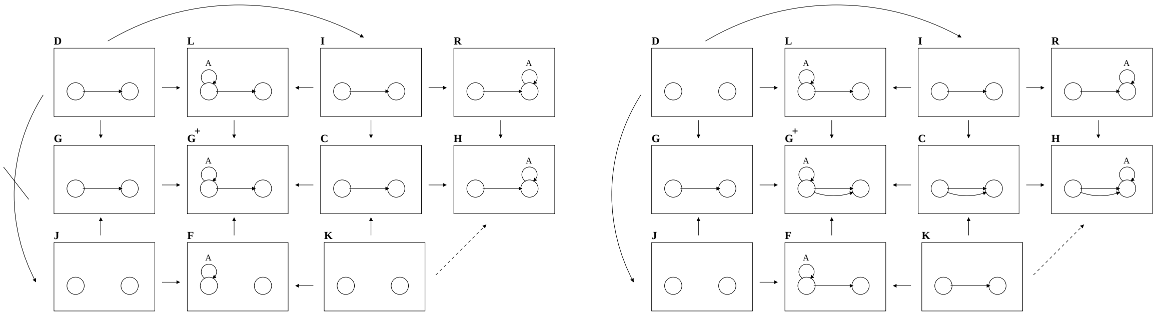
Definition 4.2.7 (**(In)Dependent Transition Labels of Productions without NACs**)**.** *Let $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$ be a transition of $(J \to G)$. We say that the transition is* independent *whenever we can add two morphisms $D \to J$ and $D \to I$ to the diagram in Definition 4.2.5 such that the diagram below commutes, i.e., $D \to I \to L = D \to L$ and $D \to J \to G = D \to G$. We write $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ if the transition is not independent and we call it* dependent*.*

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[rr, bend left] \arrow[dd, bend right] & L \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] & F \arrow[u] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$



An independent label has a borrowed context F that provides the entire left-hand side L for G, and hence G does not contribute to the rewriting. (A trivial example is a label derived with $D = \emptyset$.) The figure above on the right schematically depicts this situation where the partial match occurs only in the overlap of the interfaces J and I leading to an independent label. In Figure 4.1(a) we give an example of a dependent label where the borrowed context F is the minimal required to allow a borrowed context step. Figure 4.1(b) shows an independent label. Remind that independent labels do not have a minimal borrowed context F.

When graph rules have no NACs the bisimulation game for graphs mainly takes dependent labels into account. That is, if we modify Definition 4.2.6 in such a way that only dependent transitions $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ have to be simulated (either by a dependent or independent transition), then the resulting bisimilarity $\sim$ is unchanged. Informally, an independent label l^i generated from $J \to G$ and a rule p without NACs implies the same label l^i for $J \to G'$ via the same rule p. This is



(a) Dependent label

(b) Independent label

Figure 4.1: Examples of derived labels

formalized in Proposition 3.4.12 for rules without NACs. However, this result does not scale up automatically for the case with NACs (see discussion in Section 3.4) because the BC step for $J \to G'$ is not NAC consistent in general (G' may contain elements forbidden by the NAC). Therefore, in this chapter we use this technique only when a rule does not have NACs.

One of the main advantages of the borrowed context technique is that the derived bisimilarity is automatically a congruence, which means that whenever one graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace one part of the model by another bisimilar one.

Theorem 4.2.8 (Bisimilarity based on Productions with NACs is a Congruence). *The bisimilarity relation $\sim$ is a congruence, i.e., it is preserved by contextualization as in Definition 4.2.2.*

Bisimulation proofs often yield infinite relations. Hence up-to techniques [San95] for bisimulation are useful to relieve the onerous task of bisimulation proofs by reducing the size of the relation needed to define a bisimulation. Bisimulation up-to is defined by replacing $(K \to H)\,\mathcal{R}\,(K \to H')$ by $(K \to H)\,\mathcal{F}(\mathcal{R})\,(K \to H')$ in Definition 4.2.6, where $\mathcal{F}$ is a function from relations to relations that defines the up-to technique (for details see Section 3.4). The intuition behind up-to techniques is to automatically infer the bisimilarity of pairs of graphs from the bisimilarity of other related pairs. For example, the function $\mathcal{F}^{iso}$ generates all isomorphic copies of every pair in $\mathcal{R}$, and hence we can decide whether a pair of graphs is bisimilar by searching $\mathcal{R}$ for an isomorphic pair. A more powerful up-to technique is given by $\mathcal{F}^{C}$, which embeds all pairs into the same contexts (as in Definition 4.2.2), for all pairs and all compatible contexts. With $\mathcal{F}^{C}$ we can ensure that a pair of graphs is bisimilar whenever after removal of identical contexts we find the resulting pair of graphs in $\mathcal{R}$.

4.3 Bisimulation Verification for Borrowed Contexts

Bisimilarity is the most widespread notion of behavioral equivalence and hence algorithms for bisimulation checking are of fundamental importance for verifying that two systems are behaviorally equivalent (seen from the perspective of the environment).

The key aspects in any formal verification technique consist of properties of the underlying formal notation such as expressive power and ease of use. Another crucial feature a verification technique may have is tool support. Experience shows that very often the tasks involved in formal verification are in essence repetitive, time consuming, and in addition, when done by hand, prone to errors.

In this section we define algorithms to support bisimulation checking of graphs in the borrowed context framework. As already stated in Chapter 2 one of the main advantages of the BC technique to label derivation over the relative-pushout based techniques is that it lends itself better to mechanization due to the very simple categorical concepts required by the constructions. Chapter 6 contains further algorithms and additional implementation details.

4.3.1 Partial Match Finding

The first algorithm is for partial match finding, i.e., given a graph $J \to G$ and a rule $L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ we want to find partial matches of the form $G \leftarrow D \to L$ which lead to feasible borrowed context steps as in Definition 4.2.5.

The search of partial matches might lead to cases where the pushout complement F or C (see Definition 4.2.5) does not exist and so the borrowed context step is not feasible. This can be checked with the gluing condition of BC steps, which is based on the categorical concept of initial pushouts (see Appendix A.3) and has already been presented in Chapter 3 for adhesive categories.

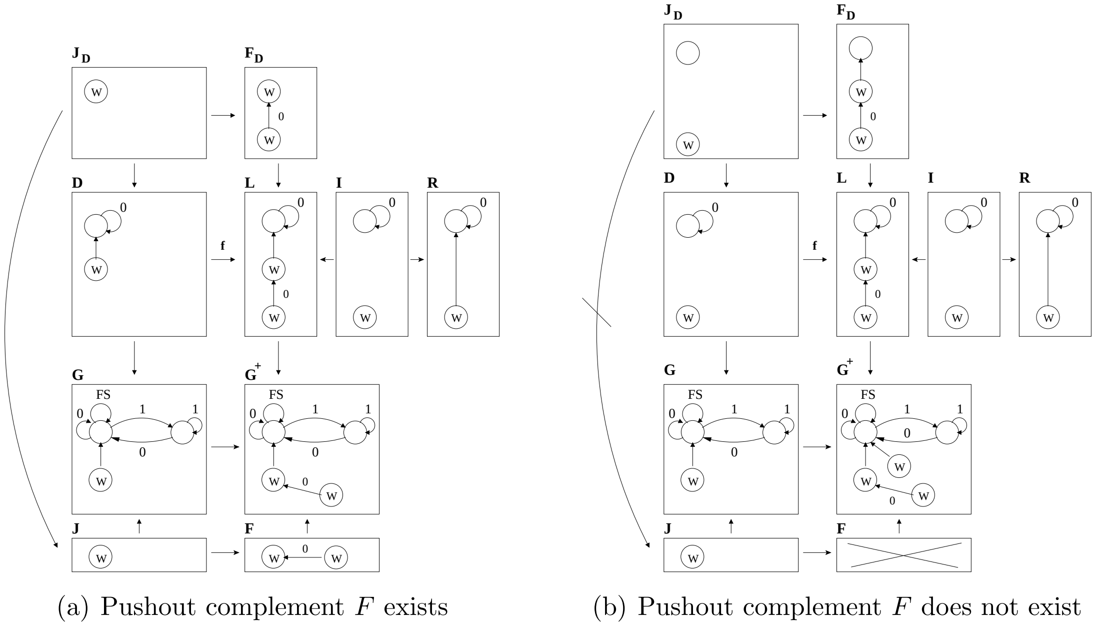
Definition 4.3.1 (**Gluing Condition of Borrowed Context Steps**). *Given a production $p\colon L \xleftarrow{l} I \xrightarrow{r} R$ and a graph with interface $J \xrightarrow{j} G$, then a partial match $G \xleftarrow{dg} D \xrightarrow{dl} L$ satisfies the* gluing condition of a borrowed context step *with respect to p and $J \xrightarrow{j} G$ if the following conditions hold for the diagram below:*

- *(i) for the initial pushout* (1) *over dl there exists an injective morphism $jj\colon J_D \to J$ such that $dg \circ jd = j \circ jj$;*
- *(ii) for the initial pushout* (2) *over dg there exists an injective morphism $ii\colon I_D \to I$ such that $dl \circ id = l \circ ii$.*

$$\begin{tikzcd}
& J_D \arrow[r] \arrow[d, "jd"] \arrow[ddd, bend right, "jj"'] \arrow[dr, phantom, "(1)"] & F_D \arrow[d] & & \\
I_D \arrow[r, "id"'] \arrow[d] \arrow[rrr, bend left, "ii"] \arrow[dr, phantom, "(2)"] & D \arrow[r, "dl"'] \arrow[d, "dg"] & L & I \arrow[l, "l"] \arrow[r, "r"'] & R \\
C_D \arrow[r] & G & & & \\
& J \arrow[u, "j"'] & & &
\end{tikzcd}$$

The gluing condition of a borrowed context step can be easily checked: we only need to build $J_D \overset{jd}{\to} D$ and $I_D \overset{id}{\to} D$ (the construction for the category of graphs is given in Appendix A.3) and check whether there exist jj and ii leading to the required commutativity. Note that this is usually easier than building the pushout of dg and dl and checking the existence of F and C by using the gluing condition of standard DPO (Definition 2.3.7).

In Figure 4.2 we illustrate Condition (i) of Definition 4.3.1: whenever there does not exist $J_D \to J$ injective with $J_D \to D \to G = J_D \to J \to G$, the pushout complement F does not exist either.



(a) Pushout complement F exists

(b) Pushout complement F does not exist

Figure 4.2: Examples of gluing condition for BC steps

In the following we propose an algorithm that takes as input a graph with interface $J \to G$ and a set $\mathcal{P}$ of productions of the form $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y\in Y}$ to find all possible partial matches $G \leftarrow D \rightarrow L$ that will lead to transition labels. We first need to introduce partial morphisms.

Definition 4.3.2 (**Partial Graph Morphism**). *Let $G = (V, E, s, t, l_v, l_e)$ be a graph as in Definition 4.2.1. A* subgraph *S of G, written $S \subseteq G$, is a graph with $V^S \subseteq V^G$, $E^S \subseteq E^G$, $s^S = s^G|_{E^S}$, $t^S = t^G|_{E^S}$, $l_v^S = l_v^G|_{V^S}$ and $l_e^S = l_e^G|_{E^S}$. A* partial graph morphism *$f \colon G \rightharpoonup G'$ is a total injective graph morphism $f \colon dom(f) \rightarrow G'$ from a subgraph $dom(f) \subseteq G$ to G'. Alternatively, a partial graph morphism can be represented as a span of total injective morphisms: $G \hookleftarrow dom(f) \rightarrow G'$.*

Given a production $p \colon L \leftarrow I \rightarrow R; \{L \rightarrow NAC_y\}_{y \in Y}$ and a graph with interface $J \rightarrow G$, we search for partial matches leading to feasible BC steps. We describe a procedure in 4 steps for one single production p, but it must be carried out for all productions of $\mathcal{P}$.

The procedure works as follows:

Step 1 : determine a subgraph L^{clean} of L, which is the largest subgraph of L containing only node and edge labels that also occur in G. The graph G^{clean} is defined analogously (with the roles of L and G exchanged).

Step 2 : create all possible subgraphs L_i^{sub} $(i \in I)$ of L^{clean}.

Step 3 : for each $i \in I$ find all injective total matches $m_x \colon L_i^{sub} \rightarrow G^{clean}$ $(x \in X)$. Each m_x forms a span of injective morphisms $G \hookleftarrow G^{clean} \overset{m_x}{\leftarrow} L_i^{sub} \hookrightarrow L^{clean} \hookrightarrow L$ which is a partial match $pm_x \colon G \leftarrow L_i^{sub} \hookrightarrow L$.

Step 4 : if p has no NACs proceed with **4.no.NAC**, otherwise with **4.NAC**:

- **4.no.NAC**: store each partial match $pm_x \colon G \leftarrow L_i^{sub} \hookrightarrow L$ that leads to a dependent label (as in Definition 4.2.7) and also satisfies the gluing condition of Definition 4.3.1.
- **4.NAC**: store each partial match $pm_x \colon G \leftarrow L_i^{sub} \hookrightarrow L$ that leads to a NAC consistent BC step (as in Definition 4.2.3) and also satisfies the gluing condition of Definition 4.3.1.

Step **4.no.NAC** exploits the fact that independent labels do not need to be matched when the underlying rule has no NAC (Proposition 3.4.12). Therefore, only partial matches leading to dependent labels are stored to derive BC steps. On the other hand, step **4.NAC** stores all partial matches that ensure NAC consistency (Definition 4.2.3) and the gluing condition.

Figure 4.3 schematically depicts the first three steps of the algorithm. In order to prove the correctness of this algorithm we need the following lemma.

$$\begin{tikzcd}
 & & NAC_y & & \\
L_i^{sub} \arrow[r, hook] \arrow[d, "m_x"'] & L^{clean} \arrow[r, hook] & L \arrow[u] & I \arrow[l] \arrow[r] & R \\
G^{clean} \arrow[d, hook] & & & & \\
G & & & & \\
J \arrow[u] & & & &
\end{tikzcd}$$

Figure 4.3: **Steps 1-3** of the partial match finding algorithm.

Lemma 4.3.3. *Given a partial match $pm\colon L \leftarrow D \rightarrow G$ between a production $p\colon L \leftarrow I \rightarrow R; \{L \rightarrow NAC_y\}_{y \in Y}$ and a graph with interface $J \rightarrow G$ the algorithm described by Steps 1-3 creates it.*

Proof. A partial morphism $pm\colon L \rightharpoonup G$ is by Definition 4.3.2 a total injective morphism $pm\colon dom(pm) \rightarrow G$, where $dom(pm)$ is a subgraph of L. Let $D = dom(pm)$ and again by Definition 4.3.2 this partial match pm can be represented as a span of total injective morphisms $L \hookleftarrow D \rightarrow G$.

Given subgraphs D_z of L ($z \in Z \cup Z'$), where Z and Z' are disjoint index sets, then in general not every D_z can be mapped onto G via a total injective morphism of the form $D_z \rightarrow G$. The subgraphs D_z with indices $z \in Z'$ are either bigger than G or contain nodes or edges with labels that are not present in G. Therefore they do not lead to partial matches. On the other hand, the subgraphs D_z with indices $z \in Z$ can be properly matched to G. For any L and G the index set Z is always non-empty because we can create at least a partial match with D_z being the empty graph.

By assumption $pm\colon L \leftarrow D_z \rightarrow G$ exists. An algorithm to create this partial match efficiently may safely discard every D_z with indices $z \in Z'$. However, in practice it is difficult to foresee all subgraphs D_z of L that will not lead to partial matches. The algorithm described by *Steps 1-3* applies some techniques to skip certain D_z with $z \in Z'$. Moreover, it uses an optimization to reduce the search space for total injective morphisms $D_z \rightarrow G$.

Step 1 discards some D_z with $z \in Z'$ by considering only the subgraphs of L which contain elements (nodes and edges) whose labels are not present in G. This leads to the inclusion $L^{clean} \hookrightarrow L$, where L^{clean} contains only elements of L whose labels are also in G. Analogously, G may contain elements whose labels are not in L, and hence these elements will never be the image of a mapping from L. This gives rise to the inclusion $G^{clean} \hookrightarrow G$, where G^{clean} contains only elements of G whose labels are in L. These inclusions are illustrated in Figure 4.3.

Now *Step 2* and *Step 3* may construct partial matches. *Step 2* creates all subgraphs L_i^{sub} of L^{clean} $(i \in I)$ and we obtain $L_i^{sub} \hookrightarrow L^{clean} \hookrightarrow L$. For every $i \in I$ *Step 3* creates total injective morphisms $m_x \colon L_i^{sub} \to G^{clean}$ $(x \in X)$, if any. Every $x \in X$ leads to a span of injective morphisms $G \hookleftarrow G^{clean} \overset{m_x}{\leftarrow} L_i^{sub} \hookrightarrow L^{clean} \hookrightarrow L$, which by composition is $pm_x \colon G \leftarrow L_i^{sub} \hookrightarrow L$. Finally, since $pm \colon L \leftarrow D_z \to L$ exists (assumption) there will be an $x \in X$ which delivers it.

□

Theorem 4.3.4. *The partial match finding algorithm defined by Steps 1-4 is correct, i.e., given $J \to G$ and a rule $p \colon L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ the algorithm creates the partial matches of the form $G \leftarrow D \to L$ that are required in the bisimulation game.*

Proof. From *Steps 1-3* and Lemma 4.3.3 we obtain partial matches $pm_x \colon G \leftarrow D \to L$ $(x \in X_1 \cup X_2 \cup X_3 \cup X_4)$, where X_1, X_2, X_3 and X_4 are disjoint index sets. However, not any pm_x leads to the a transition label. The partial matches pm_x with $x \in X_4$ do not satisfy the gluing condition of Definition 4.3.1, i.e., they will not lead to feasible BC steps because either F or C does not exist. The partial matches pm_x with $x \in X_3$ do not lead to NAC consistent BC steps since the NACs of the rule p forbids the BC step. Those partial matches pm_x with $x \in X_2$, whose underlying rule p has no NACs, may be safely discarded since they lead to independent labels and they are not necessary in the bisimulation game (see Proposition 3.4.12). Finally, *Step 4* returns only those partial matches pm_x with $x \in X_1$ that will generate the transition labels required to define a bisimulation, i.e., if the rule p possesses no NACs then the generated label l via pm_x will be dependent, whereas l will be an arbitrary label whenever p has NACs.

□

The algorithm proposed here defines a straightforward search strategy for partial matches. For rules without NACs this simple algorithm may end up producing too many "candidates" of partial matches which are later discarded since they generate independent labels (see *Step 4.no.NAC*). A more efficient algorithm should constrain the search space by skipping as many candidates as possible which will lead to independent labels. One initial idea in this direction is to find the partial match which is the threshold of independent labels, i.e., the partial match $pm \colon G \leftarrow D \to L$ where D is the largest graph such that $D \to J$ and $D \to I$ exist with the required commutativity (see Figure 4.4). Observe that other partial matches $pm' \colon G \leftarrow D' \to L$ with

$D' \hookrightarrow D$ can be automatically discarded since they lead to independent transitions labels. Hence, an improved search strategy would create partial matches $G \leftarrow D'' \rightarrow L$ with D'' bigger than the threshold D, which may end up resulting in dependent labels. More elaborate search strategies are part of our future work.

$$\begin{tikzcd}
D' \arrow[dr, hook] \arrow[drr, bend left] \arrow[ddr, bend right] & & & & \\
& D \arrow[r] \arrow[d] \arrow[rr, bend right] \arrow[dd, bend left] & L & I \arrow[l] \arrow[r] & R \\
& G & & & \\
& J \arrow[u] & & &
\end{tikzcd}$$

Figure 4.4: Strategy to improve the partial match finding.

4.3.2 Matching Transition Labels

The bisimulation game requires the comparison of labels. In the borrowed context framework we have to check transition labels for isomorphism as given below.

Definition 4.3.5 (Isomorphic Transition Labels). *Let $\mu_i = J_i \rightarrow F_i \leftarrow K_i; \{F_i \rightarrow N_i^z\}_{z \in Z}$ $(i = 1, 2)$ be transition labels as in Definition 4.2.5. Two labels μ_1 and μ_2 are* isomorphic *whenever it holds: there exist isomorphisms $J_1 \overset{\sim}{\rightarrow} J_2$, $F_1 \overset{\sim}{\rightarrow} F_2$, $K_1 \overset{\sim}{\rightarrow} K_2$ such that* (1), (2) *commute and, additionally, $\{F_1 \rightarrow N_1^z\}_{z \in Z}$ and $\{F_2 \rightarrow N_2^z\}_{z \in Z}$ are isomorphic sets, i.e., each $F_i \rightarrow N_i^z$ from one set has an isomorphic counterpart in the other set such that* (3) *commutes.*

$$\begin{tikzcd}
& J_1 \arrow[r, "\sim"] \arrow[d] \arrow[dr, phantom, "(1)"] & J_2 \arrow[d] & \\
N_1^z & F_1 \arrow[l] \arrow[r, "\sim"] & F_2 \arrow[r] & N_2^z \\
& K_1 \arrow[r, "\sim"] \arrow[u] \arrow[ur, phantom, "(2)"] & K_2 \arrow[u] &
\end{tikzcd}
\qquad
\begin{tikzcd}
N_1^z \arrow[r, "\sim"] & N_2^z \\
F_1 \arrow[u] \arrow[r, "\sim"] \arrow[ur, phantom, "(3)"] & F_2 \arrow[u]
\end{tikzcd}$$

We write $\mu_1 \cong \mu_2$ whenever μ_1 and μ_2 are isomorphic labels.

In order to check two graphs with interface $J \rightarrow G$ and $J \rightarrow G'$ for bisimilarity as in Definition 4.2.6 we first have to derive transition labels from both graphs by using the algorithm defined in Section 4.3.1 to find partial matches between the left-hand sides of the rules in $\mathcal{P}$ (set of graph rules) and the graph with interface. For every partial match we then use Definition 4.2.5 (borrowed context rewriting) to complete the borrowed context diagram, which gives us the transition label and the resulting graph with interface. Having the transitions from $J \rightarrow G$ and $J \rightarrow G'$ we can perform the matching of labels as specified in Definition 4.3.5 (isomorphic labels).

An alternative way to match labels consists in checking whether a label from $J \rightarrow G$ is also derivable from $J \rightarrow G'$ (Definition 4.3.6). This is often more efficient than deriving all labels from $J \rightarrow G'$, which could be a lot, and checking whether they match. However, this strategy does not work in general for behavioral equivalences (e.g. bisimulation), because $J \rightarrow G'$ may have additional labels, which are not present in $J \rightarrow G$. Still it is useful for behavioral inclusions (also known as as preorders) such as simulation and trace equivalence, where $J \rightarrow G'$ contains the behavior of $J \rightarrow G$, but may have extra behaviors. For bisimulation one would have to derive labels from $J \rightarrow G$, verify if they are derivable from $J \rightarrow G$ and vice versa.

Definition 4.3.6 (Derivable Label). *Given a graph $J \rightarrow G$, a label $\mu = J' \rightarrow F' \leftarrow K'; \{F' \rightarrow N'_z\}_{z \in Z}$ and a set $\mathcal{P}$ of productions of the form $L \leftarrow I \rightarrow R; \{L \rightarrow NAC_y\}_{y \in Y}$, we say that μ is* derivable *from $J \rightarrow G$ and $\mathcal{P}$ whenever it holds:*

(i) there exists an isomorphism $J \xrightarrow{\sim} J'$;

(ii) Diagrams (4.3) and (4.4) yield a feasible BC step (as in Definition 4.2.5);

(iii) the set $\{F' \rightarrow N'_z\}_{z \in Z}$ in μ is isomorphic to the set obtained via Diagram (4.4).

$$\begin{tikzcd}
 & & NAC_y & & \\
 & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO(PB)"] & L \arrow[u, "n_y"] \arrow[d, "m_1^i"] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
 & G \arrow[r, "m_2"'] \arrow[dr, phantom, "PO"] & G^+ \arrow[dr, phantom, "PB"] & C \arrow[l] \arrow[r] & H \\
J \arrow[r, "\sim"] \arrow[ur] & J' \arrow[r] & F' \arrow[u] \arrow[d] & K' \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & & N'_z & &
\end{tikzcd} \quad (4.3)$$

$$\begin{tikzcd}
NAC_y \arrow[r] \arrow[dr, phantom, "\overset{j.surj.}{=}"] & M_z \arrow[dr, phantom, "PO"] & N'_z \arrow[l] \\
L \arrow[u, "n_y"] \arrow[r, "m_1^i"'] & G^+ \arrow[u] & F' \arrow[l] \arrow[u]
\end{tikzcd} \quad (4.4)$$

This can be checked as follows: if there exists an isomorphism $J \xrightarrow{\sim} J'$ we build $G \rightarrow G^+ \leftarrow F'$ as a pushout of $G \leftarrow J \xrightarrow{\sim} J' \rightarrow F'$. For all productions $\mathcal{P}$ we find all possible total injective matches $m_1^i \colon L \rightarrow G^+$ $(i \in I)$. For each m_1^i and $m_2 \colon G \rightarrow G^+$, if m_1^i and m_2 are jointly surjective (i.e., $m^i_{1,V}(L_V) \cup m_{2,V}(G_V) = G^+_V$ and $m^i_{1,E}(L_E) \cup m_{2,E}(G_E) = G^+_E$) we can take $G \leftarrow D \rightarrow L$ as a pullback of $G \rightarrow G^+ \leftarrow L$ and thus obtain a pushout. Now we check whether the BC step is NAC consistent (Condition (i) of Definition 4.2.5). We then check the existence and compute the pushout complement $G^+ \leftarrow C \leftarrow I$ of $G^+ \leftarrow L \leftarrow I$ and the pushout

$C \rightarrow H \leftarrow R$ of $C \leftarrow I \rightarrow R$. We check if there exists an injective morphism $K' \rightarrow C$ such that the rightmost square in the second row is a pullback and add the induced morphism $K' \rightarrow H$. Now Condition (ii) of Definition 4.2.5 has been satisfied as well. Finally, from the morphisms $NAC_y \overset{n_y}{\leftarrow} L \overset{m_1^i}{\rightarrow} G^+ \leftarrow F'$ we build a set of negative borrowed context as in Condition (iii) of Definition 4.2.5 and check if it is isomorphic to $\{F' \rightarrow N'_y\}_{y \in Y}$ in μ. Hence, if there exists a total match $m_1^i \colon L \rightarrow G^+$, which allows us to complete this procedure, we say that the label $\mu = J' \rightarrow F' \leftarrow K'; \{F' \rightarrow N'_z\}_{z \in Z}$ is derivable from $J \rightarrow G$ and $\mathcal{P}$. Note that this is easier than partial match finding since we are only looking for total matches.

4.3.3 Checking Bisimulation "On the Fly"

Classical methods for bisimulation checking (e.g., see [PT87]) take as input the full state spaces which are derived from the initial processes to be compared. Their main drawback is that the whole state space must first be computed and stored. Since we want to check graphs with interfaces for bisimilarity in the borrowed context setting we can not afford to unfold the entire state space from two graphs since it may be very often infinite due to the definition of graphs up to isomorphism. However, Fernandez and Mounier defined in [FM91] a method for building the state space on the fly and checking bisimilarity based on depth-first search (DFS). Hirschkoff [Hir01] extended their work to not only allow breadth-first search (BFS), but also to deal with bisimulation up-to.

The idea behind Hirschkoff's algorithm is to take two states P and Q of labeled transition systems (LTSs) and check their bisimilarity by analyzing their state space product, which consists of pairs of the form (P, Q) as states and transition labels μ between states indicating that both states are able to evolve along the same label μ, i.e., $(P, Q) \xrightarrow{\mu} (P', Q')$. The algorithm initially checks whether P and Q are immediately bisimilar (none of them has further labels leading to successor states) or non-bisimilar (one makes a step which the other is not able to mimic). If P and Q are not found immediately (non-)bisimilar, their state space product is expanded by adding their successors reached by a common label and so the bisimilarity of (P, Q) can be only known after the recursive analysis of all successors in the state space product. With this basic technique, which is exactly the principle of Fernandez/Mounier's algorithm, the LTSs in question must be finite. The main advantage of Hirschkoff's algorithm is that in some cases it is able to perform finite proofs on infinite state space products by applying up-to techniques to handle infinite bisimulations as finite bisimulations up-to. At the end of Section 4.2 we informally define bisimulation up to $\mathcal{F}$ and give examples of functions $\mathcal{F}$. For a formal treatment of this topic we refer the reader to Section 3.4. Hirschkoff used his algorithm to check bisimilarity of polyadic π-calculus

[Mil93] processes.

We extend Hirschkoff's algorithm to check graphs with interface for bisimilarity with respect to a given set of graph productions with NACs. We also did minor efficiency improvements and added extra details to the algorithm, trying to make clear aspects that were not easy to understand in the original version.

In the following sections we recall Fernandez and Mounier's theoretical investigation [FM91] on bisimulations in terms of state space products. Since their results are abstract and hold for any system given by a labeled transition system we tailor them to our setting with borrowed contexts. The main motivation to do so is that Hirschkoff's algorithm, presented in Section 4.3.3.3, is also based on these results, but there one often becomes bogged down in several implementation details and data structures, and therefore understanding the overall process of bisimulation checking is more difficult.

4.3.3.1 Stratification of Bisimilarity

Here we briefly describe a process called stratification of bisimilarity, which dates back to Milner's book on CCS [Mil80]. The relevance of this process is that it builds the basis for algorithms to mechanically check bisimilarity, including Fernandez/Mournier's bisimulation checking technique presented in Section 4.3.3.2.

First we have to define a labeled transition system (LTS).

Definition 4.3.7 (Labeled Transition System). *A* labeled transition system *is a tuple $LTS = (S, L, T, s_0)$ where:*

- *S is a set of states;*
- *L is a set of labels;*
- *$T \subseteq S \times L \times S$ is a labeled transition relation;*
- *s_0 is the initial state.*

Example 4.3.1
From a graph with interface $J \to G$ and a set $\mathcal{P}$ of productions we can employ Definition 4.2.5 (borrowed context rewriting) to derive an LTS where: $J \to G$ and all its successors (graphs with interfaces) form the set S, the set L contains the generated transition labels $J \to F \leftarrow K, \{F \to N_z\}_{z \in Z}$, T is a relation induced by the BC steps and $J \to G$ is the initial state. A bisimulation relation $\mathcal{R}$ (as in Definition 4.2.6) relates states (graphs with interface) of LTSs that can properly mimic each other.

For the sake of readability we adopt some shortcuts: graphs with interface $J \rightarrow G$ are represented as P and Q (or $1, 2, 3, \ldots$ in the examples), and transition labels $J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z}$ as μ (or a, b, c in the examples).

A technique to obtain the bisimilarity relation $\sim$ of an LTS consists in using stratification of bisimilarity via bisimulation approximations. We also use the infix notation $P\ R_k\ Q$ for $(P, Q) \in R_k$.

Definition 4.3.8 (**Stratification of Bisimilarity**). *Given an $LTS = (S, L, T, s_0)$ with $P, P', Q, Q' \in S$ then we define bisimulation approximations as follows:*

- *$R_0 = S \times S$.*
- *$P\ R_{k+1}\ Q$, for $k \geq 0$, if*
    1. *for all P' with $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\ R_k\ Q'$;*
    2. *for all Q' with $Q \xrightarrow{\mu} Q'$, there is P' such that $P \xrightarrow{\mu} P'$ and $P'\ R_k\ Q'$.*

For each k, $P\ R_k\ Q$ if and only if $P\ R_n\ Q$ for all $n < k$. The stratification of bisimilarity *is then $R^{\sim} = \bigcap_{k \geq 0} R_k$.*

Each bisimulation approximation R_k in Definition 4.3.8 defines an equivalence relation. However, the relations R_k are not in general bisimulations. In the initial relation R_0 all states are considered equivalent, and then each subsequent relation refines the previous one by removing pairs that are certainly not bisimilar.

Proposition 4.3.9.

1. *If $k > n$ then $R_k \subseteq R_n$;*
2. *Let an LTS be* image-finite*, i.e., for every state P the set $\{P' \colon P \xrightarrow{\mu} P', for\ some\ \mu\}$ of possible successors is finite. Then it holds: the relations $\sim$ and $\bigcap_{k \geq 0} R_k$ coincide.*

Proof. See [Mil80]. □

According to Proposition 4.3.9 relations R_k decrease non-strictly as k increases. Furthermore, for image-finite LTSs the bisimilarity $\bigcap_{k \geq 0} R_k$ we obtain via stratification coincides with standard bisimilarity $\sim$.

Definition 4.3.8 (stratification) can also deal with $LTS_i = (S_i, L, T_i, s_i)$ $(i = 1, 2)$ such that $S_1 \cap S_2 = \emptyset$. In theses cases we consider $S = S_1 \cup S_2$ and $T = T_1 \cup T_2$ during the construction of $R^{\sim}$. Moreover, by item 2. of Proposition 4.3.9 given a pair (P, Q), where $P, Q \in S$, we can write: $P \sim Q$ if and only if $(P, Q) \in R^{\sim}$ and $P \nsim Q$ if and only if $(P, Q) \notin R^{\sim}$.

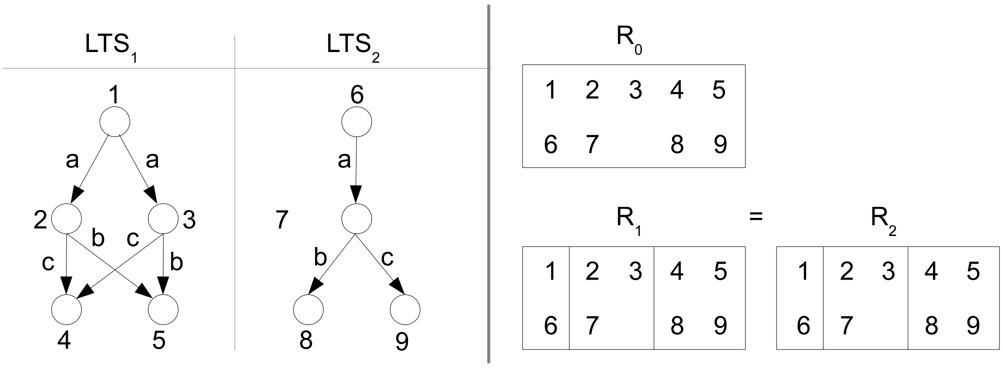


Figure 4.5: Example of stratification of bisimilarity: $1 \sim 6$.

Example 4.3.2
We show the stratification of bisimilarity for LTS_1 and LTS_2 of Figure 4.5. The bisimulation approximations obtained are depicted below, where we omit the symmetric cases and also the identities to keep the presentation short.

$$
\begin{aligned}
\mathsf{R}_0 = \{&(1,2);(1,3);(1,4);(1,5);(1,6);(1,7);(1,8);(1,9);(2,3);(2,4);(2,5);(2,6);\\
&(2,7);(2,8);(2,9);(3,4);(3,5);(3,6);(3,7);(3,8);(3,9);(4,5);(4,6);(4,7);\\
&(4,8);(4,9);(5,6);(5,7);(5,8);(5,9);(6,7);(6,8);(6,9);(7,8);(7,9);(8,9)\}\\
\mathsf{R}_1 = \{&(1,6);(2,3);(2,7);(3,7);(4,5);(4,8);(4,9);(5,8);(5,9);(8,9)\}\\
\mathsf{R}_2 = \mathsf{R}&_1
\end{aligned}
$$

In the initial relation R_0 all states are equivalent because transitions are not taken into account. The following relations are built considering the transitions. In R_1 we discard every pair whose components can not mimic each other's transitions (e.g. $(2,8)$). Note that $\mathsf{R}_2 = \mathsf{R}_1$ and so we do not need to proceed. By taking the intersection of the relations above we obtain $\mathsf{R}^{\sim} = \mathsf{R}_1$, and, for example, since $(1,6) \in \mathsf{R}^{\sim}$ we can infer $1 \sim 6$.

Algorithms for stratification of bisimilarity work on finite LTSs to ensure that after a finite number of steps the partitions become stable, i.e., $R_k = R_{k-1}$ for a certain k, and hence $R^{\sim}$ can be always computed. Moreover, the resulting $R^{\sim}$ produces the correct bisimilarity (Proposition 4.3.9.2). However, from the efficiency point of view computing stratification as in Definition 4.3.8 is rather an inadequate process due to the associated burden of generating the first relations (e.g. R_0). Recall that each R_k forms an equivalence relation. Hence, a more sensible implementation for Definition 4.3.8 would consider equivalence classes within each R_k and construct R_{k+1} based on these equivalence classes. This also allow us to better handle the identities and the symmetric cases: compare each R_k given above with their respective representations as equivalence classes on the right side of Figure 4.5.

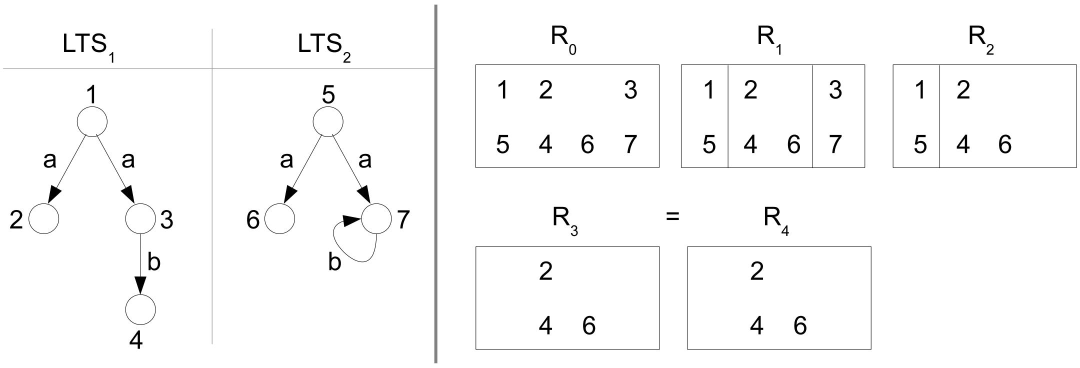


Figure 4.6: Example of stratification of bisimilarity: $1 \nsim 5$.

Example 4.3.3
For the LTSs on the left of Figure 4.6 we obtain the following bisimulation approximations:

$$\begin{aligned}
R_0 = \{&(1,2);(1,3);(1,4);(1,5);(1,6);(1,7);(2,3);(2,4);(2,5);(2,6);(2,7);(3,4);\\
&(3,5);(3,6);(3,7);(4,5);(4,6);(4,7);(5,6);(5,7);(6,7)\}\\
R_1 = \{&(1,5);(2,4);(2,6);(3,7);(4,6)\}\\
R_2 = \{&(1,5);(2,4);(2,6);\qquad\quad (4,6)\}\\
R_3 = \{&\qquad\quad (2,4);(2,6);\qquad\quad (4,6)\}\\
R_4 = R&_3
\end{aligned}$$

Observe that 4 and 7 can not mimic each other, and therefore they are no longer equivalent in R_1. During the construction of R_2 the pair $(3,7)$ performs b and lands on $(4,7)$, which is absent in R_1, and hence $(3,7)$ is not in R_2. Similarly, $(1,5)$ is not in R_3 due to the absence of $(3,7)$ in R_1. Now $R_4 = R_3$ and the intersection of these relations is $R^{\sim} = R_3$. Since $(1,5) \notin R^{\sim}$ then $1 \nsim 5$. On the right of Figure 4.6 each R_k is depicted as equivalence classes.

4.3.3.2 State Space Product and Bisimulations

In practice we often do not have the unfolded LTSs beforehand, and more importantly, if we are only interested in checking specific states (or graphs with interface) for bisimilarity it is unnecessary to build the bisimilarity relation since it is sufficient to find only one bisimulation relating these states.

The general principle behind Fernandez/Mounier's technique [FM91] consists in checking states for bisimilarity without having to construct the complete associated LTSs. They build on the fly an LTS called *state space product* (defined below) and show that two states are not bisimilar if and only if there exists an execution sequence in this product satisfying a specific criterion (see Proposition 4.3.11).

In Section 4.3.2 we have already described how transition labels are derived from graphs with interfaces and a set $\mathcal{P}$ of graph productions (see Definition 4.2.5 and Section 4.3.1). Furthermore, in Section 4.3.2 we defined how labels are matched in the borrowed context setting. A bisimulation checking procedure explores the state space product of two graphs with interface to be compared.

Definition 4.3.10 (**State Space Product and Failure**). *Let $\mathcal{P}$ be a set of graph productions p (with NACs). The* state space product *of two graphs P_0 and Q_0 is the labeled transition system generated from the initial state (P_0, Q_0) using the following inference rules:*

$$\textit{match1}: \quad \frac{P \xrightarrow{\mu}_{(d)} P' \quad Q \xrightarrow{\mu'} Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \textit{match2}: \quad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu'}_{(d)} Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \mu \cong \mu'$$

A sequence of the form $(P_0, Q_0) \overset{\mu_0}{\rightarrow} (P_1, Q_1) \overset{\mu_1}{\rightarrow} \cdots \overset{\mu_{k-1}}{\rightarrow} (P_k, Q_k)$ in a state space product is called execution sequence.

We say that a pair (P, Q) fails *to evolve whenever it holds:*

$$(P \xrightarrow{\mu}_{(d)} P' \wedge \nexists Q' \colon Q \xrightarrow{\mu'} Q' \;\; s.t. \;\; \mu \cong \mu') \vee (Q \xrightarrow{\mu'}_{(d)} Q' \wedge \nexists P' \colon P \xrightarrow{\mu} P' \;\; s.t. \;\; \mu \cong \mu').$$

Each transition $P \xrightarrow{\mu} P'$ and $Q \xrightarrow{\mu'} Q'$ is generated by Definition 4.2.5. Moreover, whenever the underlying production p of a transition has no NACs then the associated label is dependent (as in Definition 4.2.7) and we write $\rightarrow_d$. Otherwise, the label is arbitrary: $\rightarrow$.

The successors of (P, Q) are all (P', Q') such that P' and Q' respectively correspond to evolutions of P and Q along an equal (isomorphic) label μ. The rules *match1* and *match2* cover the situation when one dependent label (indicated with $\rightarrow_d$) is answered (i.e. matched) by either a dependent or independent label. Whenever the underlying production p has NACs we have to consider arbitrary labels μ and μ' (see discussion in Section 3.4), and hence the rules *match1* and *match2* coincide. If one graph can not answer, we say that the pair fails to evolve, i.e., we can infer immediately that P and Q are not bisimilar.

The following proposition provides the means to express whether P and Q are not equivalent in $R^{\sim}$ in terms of execution sequences in the state space product.

Proposition 4.3.11. *Let LTS^{sp} be the state space product generated from P and Q (as in Definition 4.3.10). Then $(P, Q) \notin R^{\sim}$ (stratification of bisimilarity as in Definition 4.3.8) if and only if there exists an execution sequence seq in LTS^{sp} such that:*

(i) *seq*: $(P, Q) = (P_0, Q_0) \overset{\mu_0}{\rightarrow} (P_1, Q_1) \overset{\mu_1}{\rightarrow} \cdots \overset{\mu_{k-1}}{\rightarrow} (P_k, Q_k)$*;*

(ii) (P_k, Q_k) *fails;*

(iii) *every* (P_i, Q_i) $(0 \leq i \leq k)$ *in seq is distinct;*

(iv) *for every* (P_i, Q_i) $(0 \leq i \leq k)$ *in seq it holds:* $(P_i, Q_i) \notin R_{k-i+1}$*.*

Proof. See Proposition 3.1 in [FM91]. □

According to Proposition 4.3.11 a pair $(P, Q) \notin R^{\sim}$ (and hence $P \nsim Q$ by item 2. of Proposition 4.3.9) if and only if we find a specific execution sequence *seq* in which: P_k fails to mimic Q_k, it contains no loop (Condition (iii)) and each $(P_i, Q_i) \notin R_{k-i+1}$.

The absence of loops in *seq* is necessary to infer the bisimilarity of a pair (P, Q) from the analysis of its successors (P', Q'), where $(P, Q) \overset{\mu}{\rightarrow} (P', Q')$. Put differently, based on the bisimilarity result of each successor (P', Q') we can determine whether (P, Q) is bisimilar.

Note that in Definition 4.3.8 (stratification) a pair in R_0 does not make its way to R_1 if and only if this pair fails. Hence, the failure of (P_k, Q_k) implies $(P_k, Q_k) \notin R_1$ in Condition (iv). The absence of (P_k, Q_k) in R_1 will determine that $(P_{k-1}, Q_{k-1}) \notin R_2$ and this happens "backwards" in *seq* until $(P_0, Q_0) \notin R_{k+1}$ because $(P_1, Q_1) \notin R_k$. In words, the bisimilarity of (P_k, Q_k) would have been required to infer bisimilarity for the other pairs in *seq*, but (P_k, Q_k) fails. This means that whenever Condition (iv) holds then *seq* is an explanation for the non-bisimilarity of P and Q.

On the left-side of Figure 4.7 we show the state space product for the example of Figure 4.5. This product has four possible execution sequences from $(1, 6)$, but none satisfies Condition (ii) of Proposition 4.3.11. Thus we can infer $(1, 6) \in \mathsf{R}^{\sim}$, and hence $1 \sim 6$.

Remark 4.3.12. *Every state space product is depicted fully unfolded in order to keep the associated text succinct. The results presented in this section also hold when the product is built on the fly, i.e., step by step.*

For the example of Figure 4.6 we give its space product on the right of Figure 4.7. There are four possible execution sequences from $(1, 5)$, but only the three rightmost are candidates for Proposition 4.3.11. This product illustrates that checking Condition (iv) is not an easy task because in order to find an execution sequence satisfying

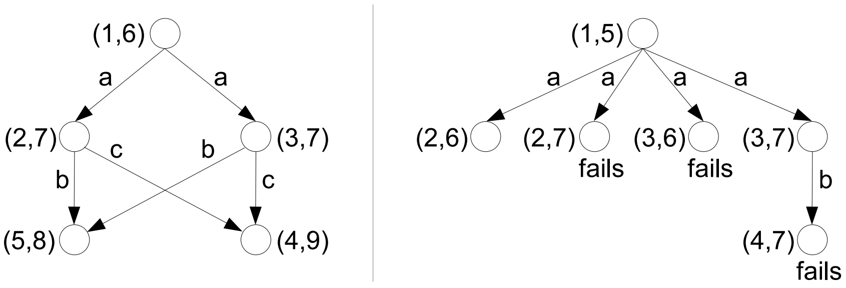


Figure 4.7: Examples of state space product.

it we also have to investigate other execution sequences. In Definition 4.3.8 (stratification) the bisimilarity result of a pair is given by the analysis of its successors. Remind the relations R_i ($\mathsf{i} = 0, 1, 2, 3$) given after Figure 4.6 and compare them with the product above. At the very beginning all pairs are equivalent in R_0. In R_1 the pairs $(2, 7)$, $(3, 6)$ and $(4, 7)$ are no longer present since they fail. The pair $(1, 5)$ is in R_1 because $(2, 6)$ and $(3, 7)$ are considered equivalent in R_0. However, $(3, 7)$ does not make its way to R_2 due to its direct dependence on $(4, 7)$. Hence, $(1, 5)$ turns out non-equivalent in R_3 because 3 and 7 have not found any matching partner. Finally, *seq*: $(1, 5) \xrightarrow{\mathsf{a}} (3, 7) \xrightarrow{\mathsf{b}} (4, 7)$ satisfies all conditions of Proposition 4.3.11 and thus $1 \nsim 5$. Notice that we had to take a look at the other execution sequences to determine that *seq* satisfies the required conditions.

Fernandez/Mounier's algorithm relies on Proposition 4.3.11 to check P and Q for bisimilarity. Each pair (P, Q) in the state space product is associated with a bit array, where each bit represents a successor of either P or Q. During the analysis of each successor (P', Q') of (P, Q) whenever it turns out that $P' \sim Q'$ we set P' and Q' to 1 in the bit array of (P, Q). Finally, after the analysis of all successors of (P, Q) if every bit in its bit array is 1 then $P \sim Q$, i.e., each successor has found a bisimilar partner.

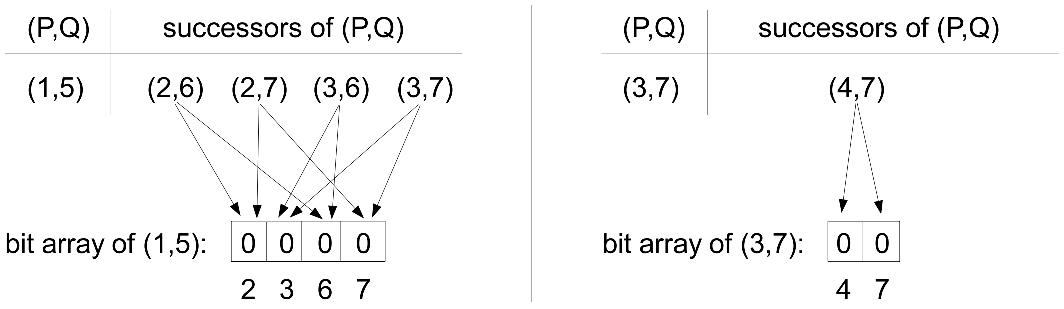


Figure 4.8: Bit arrays for $(1, 5)$ and $(3, 7)$.

In Figure 4.8 we show the initial bit arrays for $(1, 5)$ and $(3, 7)$ (of Figure 4.7), where every successor is set to 0. The pair $(4,7)$ fails, and hence 4 and 7 do not set their bits in the bit array of $(3,7)$ to 1. Since $(4,7)$ is the only successor of $(3,7)$ its analysis is finished and its bit array is 00, which implies that 3 and 7 are not equivalent. This result can be propagated to $(1,5)$, where neither 3 nor 7 sets any bit to 1. In order to determine the bisimilarity of $(1,5)$ we still have to analyze its other three pairs of successors. After the analysis of $(2,6)$ we find out that 2 and 6 properly mimic each other and we set its corresponding bits to 1 in the bit array of $(1,5)$, which is now 1010. The pairs $(2,7)$ and $(2,6)$ fail and so they do not set their bits to 1. At this moment, all successors of $(1,5)$ have been analyzed and its bit array is 1010. So we can infer that $1 \nsim 5$ because 3 and 7 have not found any matching partner.

In Figure 4.9 we give an example where LTS_1 and LTS_2 are isomorphic and so it trivially holds: $1 \sim 5$. With this example we want to illustrate that even though the root of the state space product is bisimilar it may contain execution sequences with pairs that fail. Figure 4.10 shows the initial bit array associated to $(1,5)$, which is 0000 representing $2,3,6,7$ (the successors of 1 and 5). The components in the pair $(2,6)$ are equivalent and so 2 and 6 set their corresponding bits to 1 in the bit array of $(1,5)$, which now is 1010. Then we analyze $(3,7)$, but we can only determine its equivalence after investigating $(4,8)$. Since $(4,8)$ turns out equivalent, so is $(3,7)$. This new result can be propagated to $(1,5)$, where 3 and 7 set their bits to 1 in the bit array of $(1,5)$. Note that $(2,7)$ and $(3,6)$ fail and they are not allowed to set any bit to 1. Finally, all successors of $(1,5)$ have been analyzed and its bit array is 1111, which implies $1 \sim 5$, i.e., all successors of 1 and 5 have found a matching partner.

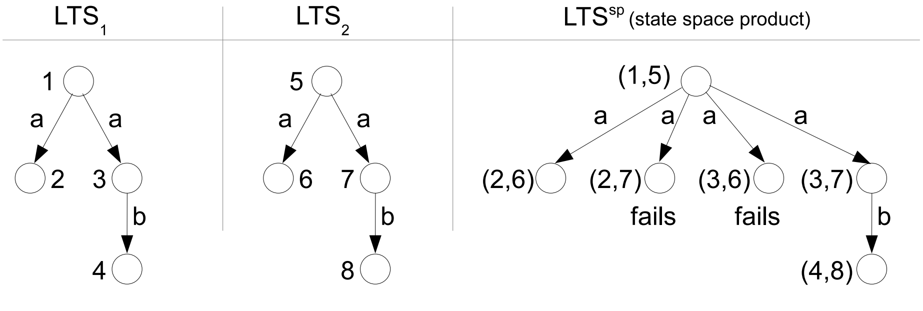


Figure 4.9: Example in which $1 \sim 5$.

Unfortunately, this strategy of deciding the bisimilarity of a pair after the complete analysis of all its successors may not work in general. There are situations in which it is possible to reach a pair in the state space product which has already been "visited" but not yet analyzed since this depends on the analysis of other pairs. Thus, the result of the analysis of such pair is unknown, i.e., it is not yet available. Figure 4.11 illustrates this problem. The dashed lines indicate the bisimilar states of LTS_1 and

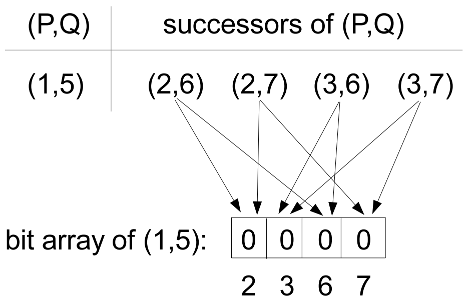


Figure 4.10: Pair $(1, 5)$: successors and its associated bit array.

LTS_2. The state space product from $(1, 3)$ is shown on the right. In Figure 4.12 we show the bit array of $(1, 4)$: the pair $(2, 5)$, which is bisimilar, has set its bits to 1. Two other successors, namely $(2, 3)$ and $(1, 5)$, fail and do not set any bit to 1. Now $(1, 4)$ depends on the analysis of $(1, 3)$. However, $(1, 3)$ also depends on $(1, 4)$, i.e., there exists a circular dependence.

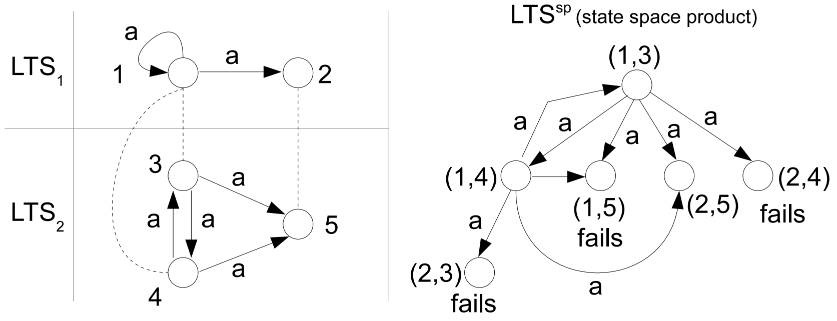


Figure 4.11: Using assumptions in bisimulation checking.

Fernandez and Mounier, and hence Hirschkoff, propose the following solution to this dependence problem. Consider we check P and Q for bisimilarity. Whenever we reach a pair in the state space product which has already been "visited" but not yet analyzed we assume it bisimilar. If this very pair turns out non-bisimilar after its analysis then an answer $P \sim Q$ is not reliable since we have made a wrong assumption. In this case, we have to run the algorithm once again, but keeping track of the pairs already known as non-bisimilar. On the other hand, an answer $P \nsim Q$ is always reliable.

Proposition 4.3.13. *Whenever the state space product of (P, Q) is finite then the bisimulation checking algorithm using assumptions (informally described above) will return that P and Q are bisimilar if and only if $P \sim Q$.*

Proof. See Proposition 4.1 in [FM91]. □

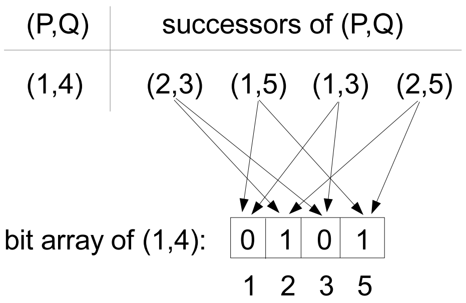


Figure 4.12: Bit array for the pair $(1, 4)$ of Figure 4.10.

In Proposition 4.3.13 Fernandez and Mounier showed that their algorithm terminates and is correct. We give here only its general idea. Let W be a set into which the algorithm stores the non-bisimilar pairs during the investigation of the state space product. Whenever a pair lands in W it does not need to be reanalyzed again. A finite state space product implies that the algorithm can only make a finite number of assumptions on bisimilarity. Whenever a pair (P', Q') assumed bisimilar turns out non-bisimilar, we can not trust on a positive answer (i.e. $P \sim Q$) of the algorithm. Thus we insert (P', Q') into W and restart the investigation of the state space product from the beginning, but keeping track of the non-bisimilar pairs already in W. This means that each wrong assumption implies that W strictly grows and the new exploration of the state space product is more constrained. After a finite number of executions of the algorithm, each due to wrong assumptions, it will be able to decide the bisimilarity of P and Q.

4.3.3.3 Algorithm for Bisimulation "On the Fly"

In the previous section we have revisited Fernandez and Mournier's technique [FM91] to check P and Q for bisimilarity using state space products. Here we present Hirschkoff's algorithm which also mechanizes this process. Additionally, we explicitly show which parts are specific tailored to the borrowed context setting.

The data structures used by the algorithm are:

- a structure **S** listing the pairs of states that still have to be inspected;
- a *Table* storing information about each pair of graphs (P, Q) under inspection in the state space product;
- a set V containing pairs that have already been visited;
- a set W containing pairs that are known to be non-bisimilar;

- a set R containing pairs that are known to be bisimilar.

By accessing **S** as stack (respectively queue) the algorithm performs a depth-first (respectively breadth-first) search on the state space product. The depth-first strategy lends itself better to give an explanation of the non-bisimilarity of P and Q because its nature, which closely follows the transitions, allows us to show an execution sequence satisfying the conditions of Proposition 4.3.11.

The main procedure is **bisimulation_check**, which calls: **succeeds**, **fails** and **propagate**. First off **bisimulation_check** initializes the sets W, R, V, inserts (P, Q) into **S** and *Table* (as described later on) and sets the variable *status* to *true*. This variable keeps track of the fidelity of our assumptions on the bisimilarity of pairs, i.e., whenever *status* = *false* and **bisimulation_check** returns *true* this result is not reliable because we have made a wrong assumption.

```
bisimulation_check$(P,Q)$ :=
  $W := \emptyset$;
  $(*)$ $R := \emptyset$; $V := \emptyset$;
  insert $(P,Q)$ into S and $Table$; $status := true$;
  while S $\neq \emptyset$ do
      take $(P_0,Q_0)$ from S;
      if succeeds$((P_0,Q_0))$
          then insert $(P_0,Q_0)$ into $R$;
                   propagate$((P_0,Q_0), true)$;
      else if fails $((P_0,Q_0))$
               then insert $(P_0,Q_0)$ into $W$;
                       propagate$((P_0,Q_0), false)$;
      else if $\boxed{(P_0,Q_0) \in V}$
               then move $(P_0,Q_0)$ from $V$ to $R$;
                       propagate$((P_0,Q_0), true)$;
      else if $\boxed{(P_0,Q_0) \in R}$
               then propagate$((P_0,Q_0), true)$;
      else {$pair\ (P_0,Q_0)\ is\ new$}
               insert $(P_0,Q_0)$ into $V$;
               $\{(P_0,Q_0) \xrightarrow{\mu} successor(P_0,Q_0)\}$
               insert successors of $(P_0,Q_0)$ into S;
               for each $successor(P_0,Q_0)$ do
                  if $successor(P_0,Q_0) \notin Table$
                    $\wedge\ successor(P_0,Q_0) \notin W \cup R \cup V$
                      then insert it into $Table$;
               end for
  end while
if $(P,Q) \notin R$
  then return $false$
  else if $status$ then return $true$ else loop back to $(*)$
```

```
succeeds$(P,Q)$ := $Table(P,Q).successors = \emptyset$
        $\wedge\ Table(P,Q).fails = false$

fails$(P,Q)$:=
  if $(P,Q) \in Table$
      then $Table(P,Q).fails \vee (P,Q) \in W$
      else $(P,Q) \in W$

propagate$((P,Q), success)$ :=
  if $(P,Q) \in R \wedge success = false$
      then $status := false$;
  if $(P,Q) \in Table$
      $\wedge\ Table(P,Q).successors\ is\ complete$[2]
      then remove $(P,Q)$ from $Table$;
  for each $(P_f,Q_f) \in Table$ with $(P_f,Q_f) \xrightarrow{\mu} (P,Q)$ do
      $Table(P_f,Q_f).successors(P,Q) := true$;
      if $success$
          then $Table(P_f,Q_f).m(P) := true$;
                 $Table(P_f,Q_f).m(Q) := true$;
      if $Table(P_f,Q_f).successors$ is complete
        then
          if $\exists j = false \in Table(P_f,Q_f).m$
            then
                insert $(P_f,Q_f)$ into $W$;
                propagate $((P_f,Q_f), false)$;
            else
                if $(P_f,Q_f) \in V$
                    then take it from $V$ to $R$;
                propagate $((P_f,Q_f), true)$;
  end for
```

The **while do** statement in **bisimulation_check** is in charge of building and exploring the state space product. It takes a pair (P_0, Q_0) from **S** and checks with **succeeds** whether it is immediately bisimilar (e.g. none of them is able to derive

[2] This means that either all successors of (P, Q) have already been analyzed (•) or $successors = \emptyset$.

further labels). If (P_0, Q_0) is not immediately bisimilar, **fails** checks if this pair fails to evolve or if it is already known as non-bisimilar (i.e., it is in W). If it fails we insert it into W and use **propagate** to update this new result in the state space product stored in *Table*, which can possibly lead to the discovery of new (non-)bisimilar pairs. If (P_0, Q_0) does not fail but has already been visited (i.e., it belongs to V) we assume it is bisimilar by moving it from V to R and only after the analysis of all its successors (where the notion of successor is specified in Definition 4.3.10) we are able to decide whether the pair is really bisimilar (as we assumed) or not. If by processing **S** we find a pair that is already in R we update *Table* using **propagate**. When none of the above conditions hold (P_0, Q_0) is a new pair and must be analyzed: we mark it as visited by inserting it into V, calculate its successors and insert them into **S** and *Table* (the latter only if the successor is not currently in *Table* under investigation and has not yet been analyzed). When all pairs of successors have already been analyzed ($\mathbf{S} = \emptyset$) the algorithm can determine the bisimilarity of (P, Q).

Figure 4.13 shows two small transition systems and their respective state space product. Figure 4.14 depicts the states under investigation in **S** and their current information in *Table*. The states 1–6 represent graphs with interfaces and a, b transition labels with borrowed contexts. Consider the pair $(1, 4)$ in *Table*. The entry *successors* shows the successors of $(1, 4)$ in the state space product together with a boolean value *true* (●) or *false* (○), indicating which pairs of successors have already been analyzed. The entry m lists the successor states (e.g. 3 and 6 with *false* [□], 2 and 5 with *true* [☒]), indicating which state has found a bisimilar partner. Whenever a pair of successors has been analyzed, if it turns out to be bisimilar both m-fields of the pair are set to *true* (☒). For example, successors $(2, 5)$ and $(2, 6)$ have been explored and only $(2, 5)$ is bisimilar. The m-field corresponds to the bit array of Fernandez and Mounier's technique in Section 4.3.3.2. Note that $(2, 5)$ and $(2, 6)$ are no longer in *Table* since the algorithm keeps in *Table* only the states under investigation. If all successors of (P, Q) have been analyzed (all are set to ●) and all fields of m are set to *true* (☒) then (P, Q) is bisimilar. If there is in m at least one graph with *false* (□) then (P, Q) is not bisimilar. The entry *fails* indicates if a pair fails to evolve (according to Definition 4.3.10). A non-bisimilar pair has always at least one successor leading to a failure, but as already shown in Section 4.3.3.2 a bisimilar pair might also have successors leading to a failure. In this example even though $(2, 6)$ and $(3, 5)$ fail, it is clear that $(1, 4)$ is bisimilar since LTS_1 and LTS_2 are isomorphic.

A new pair (P, Q) is inserted into *Table* as follows. Using Definition 4.3.10 (state space product and failure) we can determine if (P, Q) fails to evolve and if it has successors. We insert (P, Q) into *Table* with the following data in case of failure: *successors* $= \emptyset$, $m = \emptyset$ and *fails* $=$ *true*. If the pair does not fail we fill *successors* with the successors of (P, Q), m with the states of the successors of (P, Q) and all

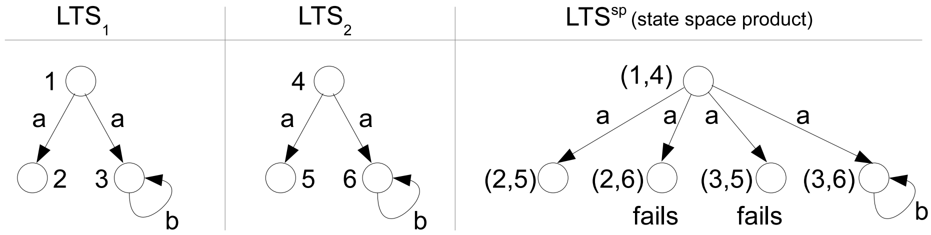


Figure 4.13: Two LTSs and their corresponding state space product.

S
(3,6)
(3,5)
(1,4)

Table			
(P,Q)	*successors*	m	*fails*
$(1,4)$	(2,5) (2,6) (3,5) (3,6) ● ● ○ ○	2 3 5 6 ☒□☒□	*false*
$(3,5)$			*true*
$(3,6)$	(3,6) ○	3 6 □□	*false*

Figure 4.14: Pairs still under analysis (in **S**) and their information in *Table*.

boolean values are set to *false*.

If the algorithm above is employed to check graphs with interfaces for bisimilarity the operators $\in$ and $\notin$ require isomorphism checks. Furthermore, note that the insertion of a pair (P, Q) of graphs with interface into *Table* requires the borrowed context machinery (in Definition 4.3.10) to not only derive transition labels from P and Q, but also to match them in order to obtain the pairs of successors to be inserted into the fields *successors* and m. At the beginning of Section 4.3.3.2 we describe how labels with borrowed contexts are derived and matched. In Section 6.5 we give additional details and algorithms to carry out these tasks.

The procedure **propagate** is in charge of updating the information in *Table* concerning the state space product analysis. The procedure **bisimulation_check** calls **propagate** to inform the space product under investigation whether a pair is bisimilar (*true*) or not (*false*). If we assumed (P, Q) bisimilar (i.e., it was moved to R) and it turned out non-bisimilar then the variable *status* is set to *false*, which means that the result of the current run of the algorithm is not reliable, i.e., we have made a wrong assumption about the bisimilarity of some pair. In this case when **S** is empty in **bisimulation_check** the algorithm is restarted but retaining the information in W about states which are already known to be non-bisimilar. We do not immediately restart the algorithm because we can still find other non-bisimilar pairs during the current execution. The procedure **propagate** keeps in *Table* only the pairs under analysis, removing the ones that have already been analyzed. When one pair is propagated, the predecessors of this pair should also be informed about the new results.

When a pair is propagated with *true* or *false* the algorithm always sets this pair as analyzed (*true* [•]) in the list of *successors* of each of its predecessors that are still under analysis (in *Table*). Only if the pair is propagated with *true* its respective graphs in m of its predecessors are set to *true* (☒). When the last successor of a given pair has been analyzed we can verify its bisimilarity. If the pair has at least one graph in m set to *false* it is non-bisimilar and we propagate this result. Otherwise it is bisimilar and we propagate this result.

In the example of Figure 4.13 the algorithm does not employ any assumption because there is no circular dependence in the state space product. Thus, 1 and 4 are found bisimilar within one single run of the algorithm. On the other hand, the state space product of Figure 4.15 has loops and by calling **bisimulation_check** to check $(1, 8)$ for bisimilarity assumptions are employed: the second time we visit the pair $(2, 12)$ (after performing b from $(3, 13)$) we assume it bisimilar (i.e. move it from V to R), but when the analysis of $(4, 14)$ is completed we find out that $(2, 12)$ is not bisimilar, which is a contradiction to the assumption. Analogously, it happens to $(5, 9)$. Finally, during the second run of the algorithm, since $(2, 12)$ and $(5, 9)$ have been inserted into W during the first run, the algorithm is able to conclude $1 \sim 8$.

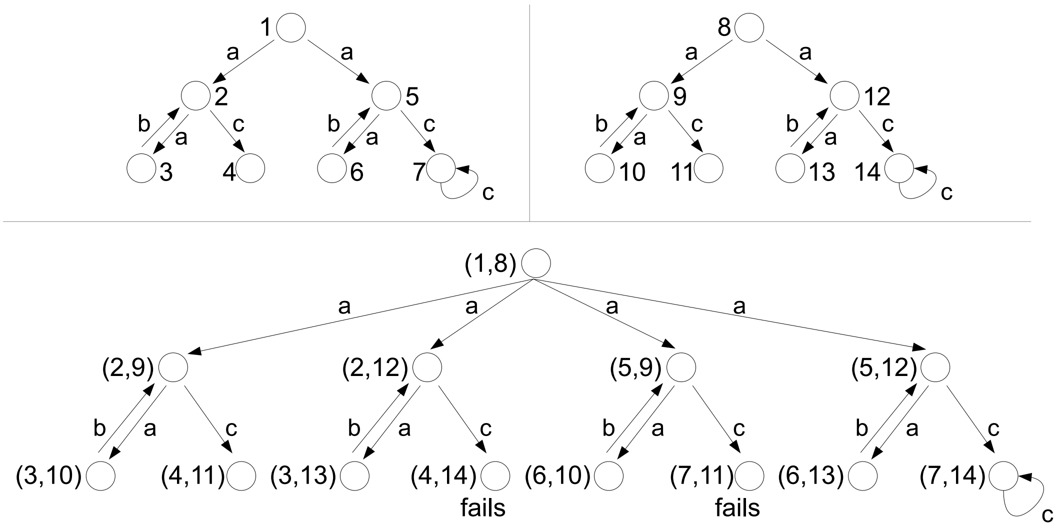


Figure 4.15: A more complex example in which wrong assumptions are made.

The examples presented here contain LTSs that are isomorphic, and hence they are trivially bisimilar. However, they are useful to illustrate how the algorithm works and also the issues on assumptions. Examples of bisimilarity and non-isomorphic LTSs are given in Sections 4.3.3.1 and 4.3.3.2.

Hirschkoff's algorithm, as opposed to Fernandez and Mounier's, also handles bisimulation up-to [San95]. These up-to techniques enable us to reduce the size of the relation needed to define a bisimulation. Moreover, they also provide the means to check bisimilarity with finite up-to relations in some cases where any bisimulation is infinite. If the algorithm has to handle bisimulation up-to we have only to replace in the procedure **bisimulation_check** $\boxed{(P_0, Q_0) \in V}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(V)}$ and $\boxed{(P_0, Q_0) \in R}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(R)}$, where $\mathcal{F}$ describes the up-to technique. For example, consider the up-to context technique $\mathcal{F}^C$ informally described at the end of Section 4.2. With this technique the algorithm is able to decide whether a pair (P_0, Q_0) can be obtained from a proper contextualization of another pair already equivalent in R. If this is the case then $(P_0, Q_0) \in \mathcal{F}^C(R)$, and we do not need to analyze (P_0, Q_0) since we can infer its bisimilarity from related pairs already analyzed. For the formal treatment of bisimulation up-to and also techniques for borrowed contexts we refer the reader to Section 3.4. In Section 6.6 we define algorithms to check whether a pair (P, Q) of graphs with interface belongs to $\mathcal{F}(\mathcal{R})$. For the refactoring examples proposed in this chapter (see Sections 4.4 and 4.5) we use the up-to context technique $\mathcal{F}^C$.

Our version of the bisimulation checking algorithm is very similar to Hirschkoff's. Our main contribution to the algorithm is the full specification of how *Table* is used to store and process the state space product investigation. A small efficiency improvement can be seen in the **for each** statement of **bisimulation_check**, where we added extra conditions in order to avoid reanalyzing pairs already investigated. Furthermore, we checked that the algorithm also works in our setting of borrowed contexts.

4.4 Example 1: Minimization of Finite Automata

We will now come back to our original motivation: showing that refactoring preserves behavior.

Inspired by the algorithm on minimization of automata by merging equivalent states, given in Hopcroft/Motwani/Ullman's classic book [HMU00] on automata theory, we define this process in terms of graph productions (see Figure 4.19). On one hand this enables us to perform minimization of deterministic finite automata (DFA) via graph transformations, but, on the other hand, we still do not have the means to verify whether minimized DFA and their original versions (given as graphs) accept the exact same language. We could manually check if both DFA are equivalent, but for complex DFA this can easily become time consuming. A reasonable solution is to use the borrowed context machinery to carry out this task, i.e., given a finite automaton and its refactored minimal version we check them for bisimilarity, which implies language equivalence for the regular language generated by the automata. However, to do so it is required a set of graph productions describing the operational semantics for DFA. As formally defined in [HMU00] an automaton can either consume symbols of an alphabet or accept the sequence of symbols previously consumed. Based on that we describe these operations as a set of graph productions in Figure 4.16.

Each *DFA* is described as a graph with interface, where unlabeled nodes are states and directed labeled edges are transitions (see DFA1 and DFA2 in Figure 4.16). An FS-loop marks a state as final. A W-node has an edge pointing to the current state and this edge points initially to the start state. The W-node is the interface, i.e., the only connection to the environment. So an automaton DFA is hidden in a "black box" whose observable part is only J.

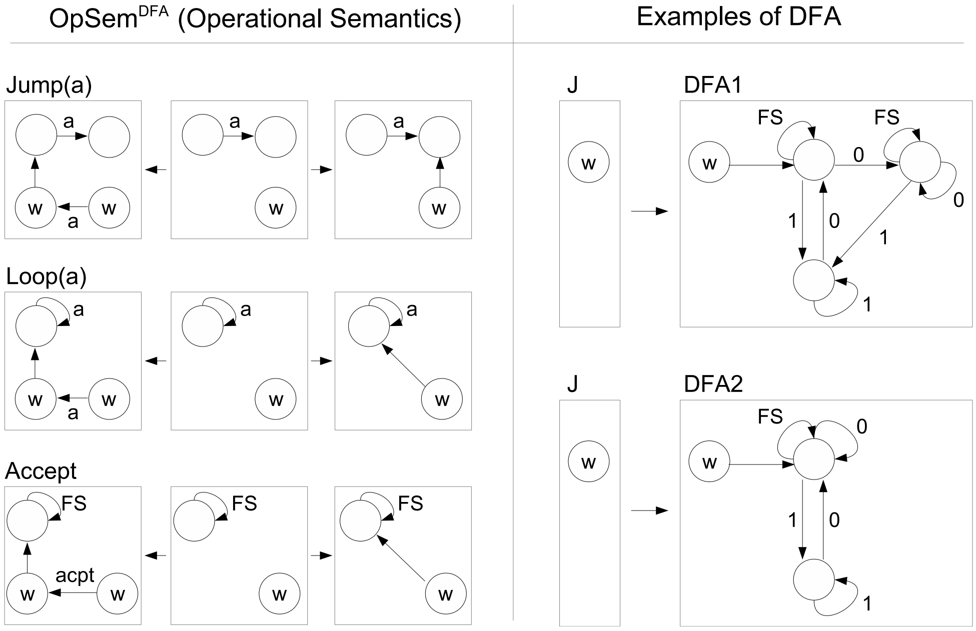


Figure 4.16: Operational semantics rules for *DFA* and two examples of *DFA*.

The operational semantics for *DFA* is given by a set OpSem$^{\text{DFA}}$ of rules containing Jump(a), Loop(a) and Accept depicted in Figure 4.16. The rules Jump(a), Loop(a) must be defined for each symbol $a \in \Lambda$, where Λ is a fixed alphabet. A *DFA* may change its state according to the rules in OpSem$^{\text{DFA}}$. The W-node receives a symbol (e.g. 'b') from the environment in form of a b-labeled edge connecting W-nodes, e.g., the string 'bc' is $\textcircled{w} \xleftarrow{b} \textcircled{w} \xleftarrow{c} \textcircled{w}$. An acpt-edge between W-nodes marks the end of a string. When such an edge is consumed by a *DFA*, the string previously processed is accepted. For example, whenever $\textcircled{w} \xleftarrow{0} \textcircled{w} \xleftarrow{0} \textcircled{w} \xleftarrow{1} \textcircled{w} \xleftarrow{0} \textcircled{w} \xleftarrow{\text{acpt}} \textcircled{w}$ is consumed by a *DFA* the string 0010 is accepted.

In Figure 4.17 we illustrate the automaton DFA1′, which is DFA1 of Figure 4.16 attached to the string 0010 previously depicted. This string is consumed by DFA1′ via a sequence of standard DPO transformations: $\text{DFA1}' \overset{\text{Jump(0)}}{\Longrightarrow} \text{DFA1}'' \overset{\text{Loop(0)}}{\Longrightarrow} \bullet \overset{\text{Jump(1)}}{\Longrightarrow} \bullet \overset{\text{Jump(0)}}{\Longrightarrow} \bullet \overset{\text{Accept}}{\Longrightarrow} \text{DFA1}$, whose first step is shown in Figure 4.17. In the last step the

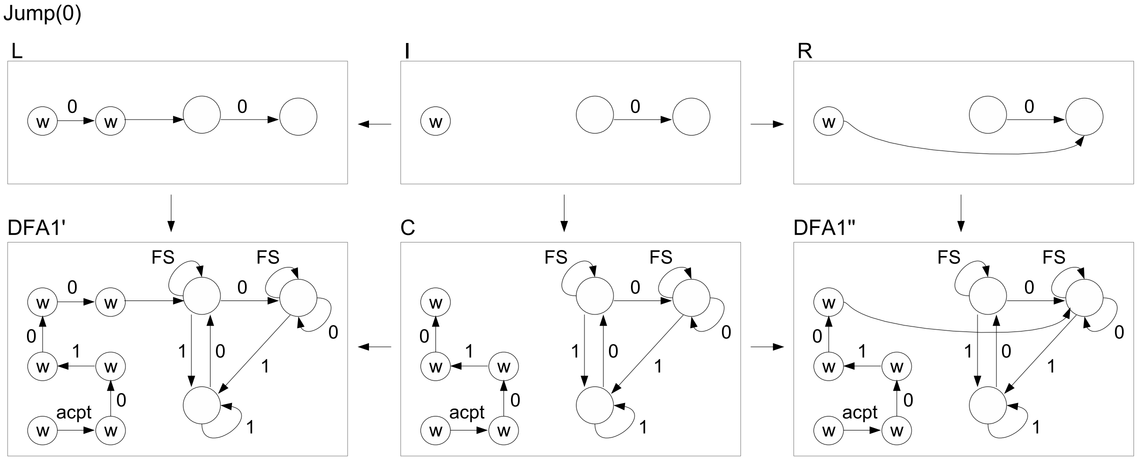


Figure 4.17: $\mathsf{DFA1}'$ consumes the symbol $\mathsf{0}$ via a standard DPO transformation.

automaton consumes the acpt-edge indicating that 0010 has been accepted and becomes DFA1. Analogously can be done for DFA2, which also accepts the same string via: $\mathsf{DFA2}' \overset{\mathsf{Loop(0)}}{\Longrightarrow} \bullet \overset{\mathsf{Loop(0)}}{\Longrightarrow} \bullet \overset{\mathsf{Jump(1)}}{\Longrightarrow} \bullet \overset{\mathsf{Jump(0)}}{\Longrightarrow} \bullet \overset{\mathsf{Accept}}{\Longrightarrow} \mathsf{DFA2}$. Even though we can manually demonstrate via DPO transformations that a specific string is accepted by two automata, it is still difficult to guarantee with standard DPO machinery that both automata recognize the exact same language, i.e., they are equivalent.

Our idea consists in employing the borrowed context (BC) technique to check automata (given as graphs with interface) for bisimilarity, which implies language equivalence. In standard DPO the consumption of strings resembles batch processing, whereas in the borrowed context framework strings are processed through interactions with the environment. In Figure 4.18 we show an example of BC step (as in Definition 4.2.5) from the automaton $\mathsf{J} \rightarrow \mathsf{DFA1}$. The environment provides $\mathsf{J} \rightarrow \mathsf{DFA1}$ with the borrowed context F (containing the symbol $\mathsf{0}$) in order to trigger the rewriting. None negative borrowed context is generated because $\mathsf{Jump(0)}$ does not have NACs. Analogously a BC step from $\mathsf{J} \rightarrow \mathsf{DFA2}$ via $\mathsf{Loop(0)}$ produces the exact same transition label $\mathsf{J} \rightarrow \mathsf{F} \leftarrow \mathsf{K}$ as in Figure 4.18. These transition labels represent the possible interactions one can make with each automaton, and therefore the bisimulation checking algorithm described in the previous section exploits them to show the bisimilarity of two automata.

Figure 4.19 depicts graph productions to minimize *DFA* by merging equivalent states. The idea of the algorithm is to identify the distinguishable states, followed by the merging of equivalent states. To the left of each rule we depict the negative application conditions (if any). The algorithm is defined by several graph productions spread over three layers, where each layer applies its rules as long as possible before the rules of the next layer can be applied. In practical terms, the transformation

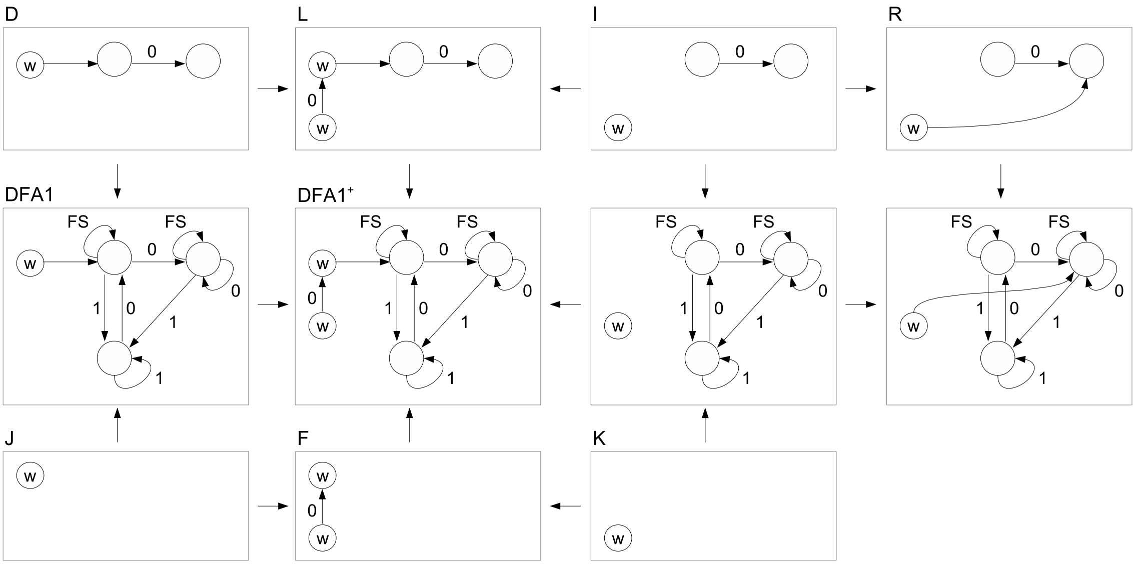


Figure 4.18: Borrowed context step from J → DFA1 via Jump(0).

begins with rules of Layer 0. If no more rule of Layer 0 is applicable, the rules of Layer 1 come into play. The rules in Layer 0 examine the transitions labels for every two states and determine if they are distinguishable. The rule in Layer 1 merges equivalent nodes, i.e., nodes without a dist-edge between them. Finally, the rules in Layer 2 remove all dist-edges and redundant transitions between two states.

In theory, one should prove that a refactoring algorithm is correct and complete. However, in practice this can be very difficult due to the complexity of the model under refactoring and the refactoring process. In our case, finite automata is a simple model, but showing these properties for the minimization algorithm of Figure 4.19 is far from trivial. Therefore, we have defined the graph productions of Figure 4.19 in AGG [AGG] and checked that transformations via these rules indeed produce the minimal versions of automata given as input. For our purpose here this is sufficient, since we are primarily interested in checking behavior preservation, i.e., that an automaton and its refactored minimal version present the same observable behavior.

Given an automaton and its minimal refactored version we can check them for bisimilarity with respect to the set $\mathsf{OpSem}^{\mathsf{DFA}}$ of rules defining the operational semantics of *DFA*. Note that for *DFA* borrowed context bisimilarity coincides with language equivalence. In other words, whenever we can show with the borrowed context technique that two automata (as graphs with interface) are bisimilar then both recognize the same language. What should be shown is that bisimilarity on automata seen as transition systems corresponds to the bisimilarity that we obtain via the borrowed context technique. This requires showing that transition labels with borrowed contexts and transitions of automata are equivalent. However, proving that these two

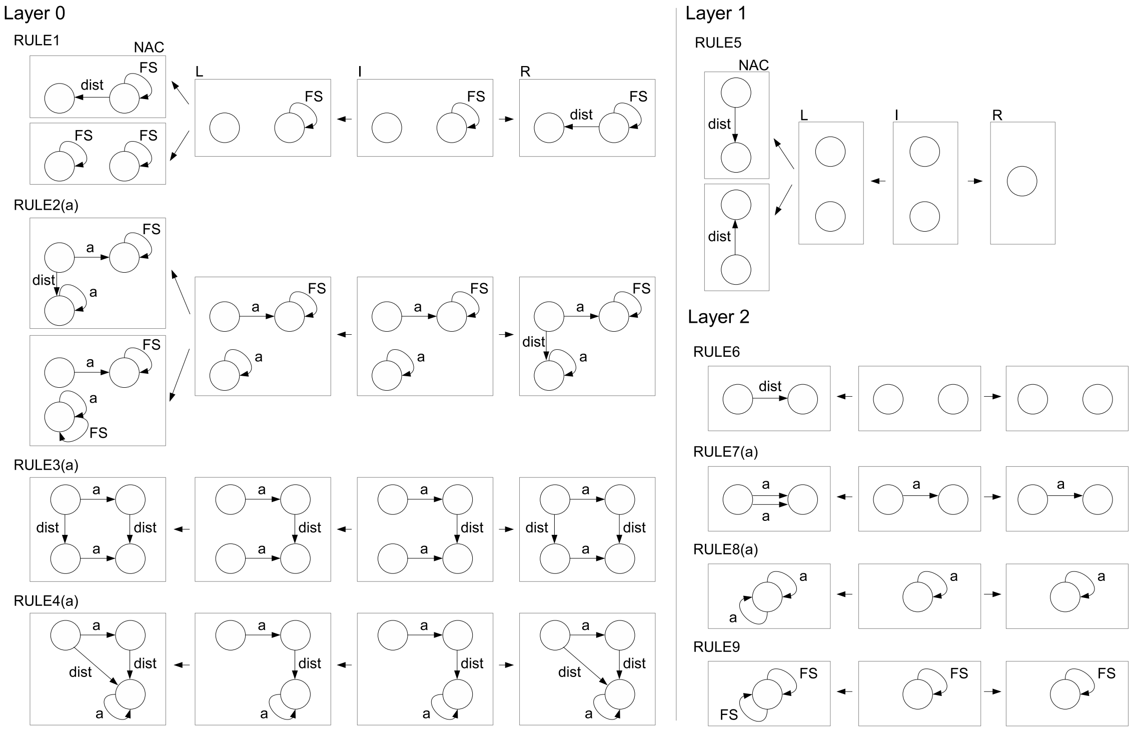


Figure 4.19: Productions for *DFA* minimization.

notions of bisimilarity coincide is not trivial, e.g., independent transition labels (as in Definition 4.2.7) in the BC framework do not have a direct meaning in the automata model. A more detailed investigation of this issue is part of our future work.

Consider J → DFA1 and J → DFA2 previously depicted in Figure 4.16. By applying the minimization algorithm on J → DFA1 we obtain J → DFA2 as its minimal version. Now we can use the procedure **bisimulation_check** to verify that these two automata are indeed bisimilar. By doing so we show behavior preservation for the refactoring of only one automaton. Proving behavior preservation for the refactoring process (i.e., all transformations preserve behavior) is a much more ambitious goal (see discussion in Chapter 5).

The general principle of **bisimulation_check** to verify J → DFA1 and J → DFA2 for bisimilarity is summarized as follows. In Figure 4.20 we give the resulting state space product fully unfolded, where the omitted interfaces of the graphs in the tuples contain only one node labeled W. First off we recall how the state space product is generated. From $(\mathsf{J} \to \mathsf{DFA1}, \mathsf{J} \to \mathsf{DFA2})$ (leftmost tuple) the algorithm calculates the partial matches between both graphs and the left-sides of the rules in $\mathcal{P} = \{\mathsf{Jump(a)}, \mathsf{Loop(a)}, \mathsf{Accept}\}$ $(\forall \mathsf{a} \in \Lambda)$ (as described in Section 4.3.1). Then it constructs the corresponding borrowed context steps as in Definition 4.2.5 (borrowed context rewriting). Now it checks whether the labels of one graph have a matching

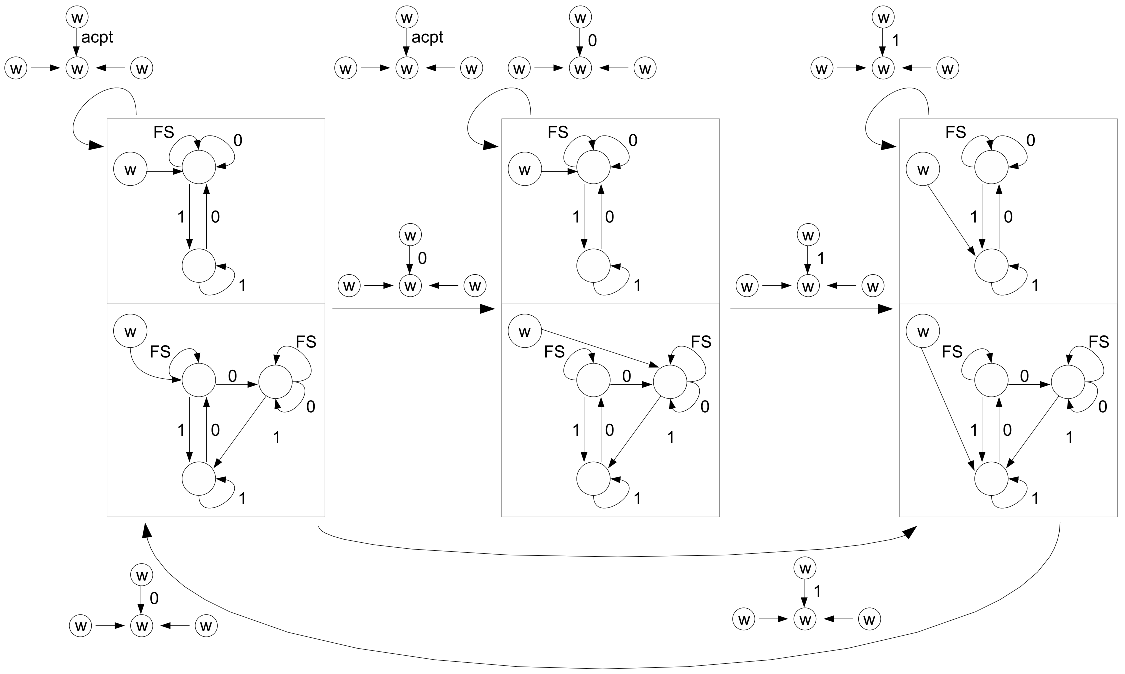


Figure 4.20: State space product for J → DFA1 and J → DFA2.

partner on the other graph (as in Definition 4.3.5). If there is a label from one graph that does not have a partner on the other graph then the graphs fail to mimic each other's move (Definition 4.3.10). Otherwise, we expand the state space product by adding the successor states and the transition labels.

The procedure **bisimulation_check** can not infer that the initial pair (J → DFA1, J → DFA2) is immediately (non)-bisimilar. Hence, the state space product is expanded by adding the pair in the middle of Figure 4.20, and only after the analysis of this successor the algorithm can infer the bisimilarity of (J → DFA1, J → DFA2). The middle pair is not immediately (non)-bisimilar as well and the state space product is once again expanded. The same occurs with the rightmost pair, whose 0-transition leads to the initial pair (J → DFA1, J → DFA2). Note that we are visiting the initial state for the second time, but its bisimilarity result is still unknown (their successors have not yet been analyzed). Thus, we assume it is bisimilar. This result is propagated to its predecessor and so we are also able to decide the bisimilarity of the rightmost pair since its successors have been analyzed. It turns out bisimilar and we propagate this result to the middle pair which turns out bisimilar as well. Finally, the bisimilarity of the middle pair implies that all successors of (J → DFA1, J → DFA2) have been analyzed and the algorithm concludes that the initial pair is bisimilar. Intuitively it is easy to check the bisimilarity of this pair since the state space product does not contain any pair which fails, and hence there does not exist an execution sequence satisfying Proposition 4.3.11.

The transition labels of Figure 4.20 are generated via partial matches with maximal overlaps between the graphs under rewriting and the left side of each rule (see Definition 4.2.5). These labels are exactly the dependent ones (Definition 4.2.7). Even though other partial matches exist with smaller objects D, they do not ensure the existence of the pushout complement objects F (borrowed context) which implies that the BC step is not feasible (no label is derived). Apart from the transition labels depicted in the state space product there exist other labels derived with D as the empty graph, but they lead to independent labels (which are omitted in Figure 4.20).

4.5 Example 2: Flattening of Statecharts

Statecharts [Har87] are finite state machines enriched with hierarchy on states and parallelism on transitions. A statechart with hierarchy has states within states, which breaks down the system under specification in smaller modules to help improve the comprehension, readability and maintenance of the specification. Any hierarchical statechart can be translated into a plain (flat) state machine, and hence every hierarchical statechart has a semantically equivalent flat statechart.

In this section we define a refactoring to flatten hierarchical statecharts. Our refactoring is inspired by the works of Minas/Hoffmann [MH08] and Geiger/Zündorf [GZ05]. Minas and Hoffmann take into account two kinds of hierarchical states, namely and- and or-states and describe the flattening process in pseudocode. Geiger and Zündorf tackle or-states and specify the refactoring in terms of graph transformation rules in Fujaba [Fuj].

First off, we specify the operational semantics of hierarchical and plain statecharts as graph productions. In addition we define graph rules to model the flattening of or-states. Our goal is to flatten hierarchical statecharts and use the borrowed context technique to check the behavior of hierarchical statecharts and their flat versions. Geiger and Zündorf can only execute transitions in plain statecharts since their operational semantics does not take hierarchical states into account.

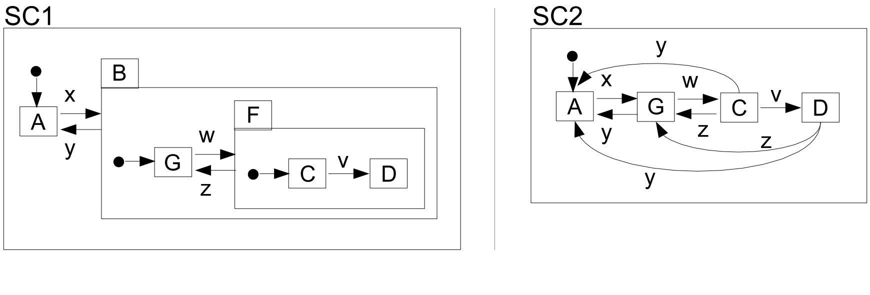


Figure 4.21: Examples of Statecharts.

Figure 4.21 depicts two statecharts: SC1 containing two or-states (B and F) and its corresponding flat version (SC2). Each or-state contains a complete statechart diagram and when an or-state is active then one of its contained states is active. Our simplified statecharts consist of a collection of plain states, non-empty or-states and initial pseudo states. Initial pseudo states cannot be active and whenever a statechart is activated then its initial pseudo state activates its connected state. An or-state has sub-states and exactly one of them may be active at a time. Final states and history states are not considered. The transitions take into account only events, so firing conditions and actions are not considered as well. Furthermore, transitions are not allowed to cross the borders of or-states.

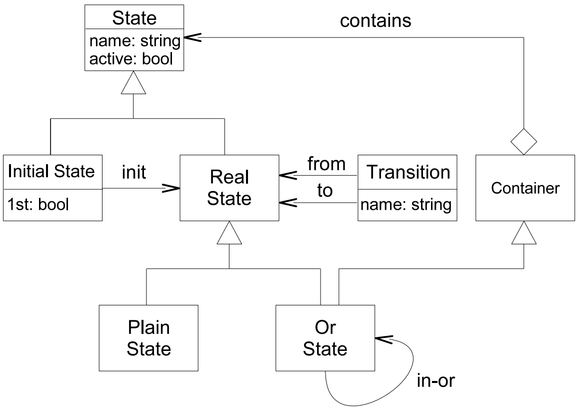


Figure 4.22: Metamodel for our simplified statechart diagrams.

The metamodel for our statechart diagrams is shown in Figure 4.22 as a class diagram. States are either initial pseudo or real states (plain and or-states). Each state has a name and a boolean value indicating whether it is active or not. Or-states contain the states of the contained state diagram. Nested or-states indicate their superstates with in-or-edges. An initial pseudo state indicates which real state is initial. Transitions are associated to event names and specify their respective states by using from- and to-edges.

Figure 4.23 shows SC1 and SC2 (of Figure 4.21) as graphs with interface. Nodes labeled st represent states and each kind of state is given by the loops: init, plain and or. When a state has an act-loop it is active. The t-nodes represent transition events. The names of states and transitions are given by the label on the node pointed by name-edges. Finally, the ct-edge stands for the "contains"-relation in the metamodel. The w-node has an edge connected to the initial pseudo state (marked with a 1st-loop). This w-node is the only part of the statechart that is observable (see J), i.e., the part the environment may interact with.

The operational semantics for our simplified statecharts follows a similar synchro-

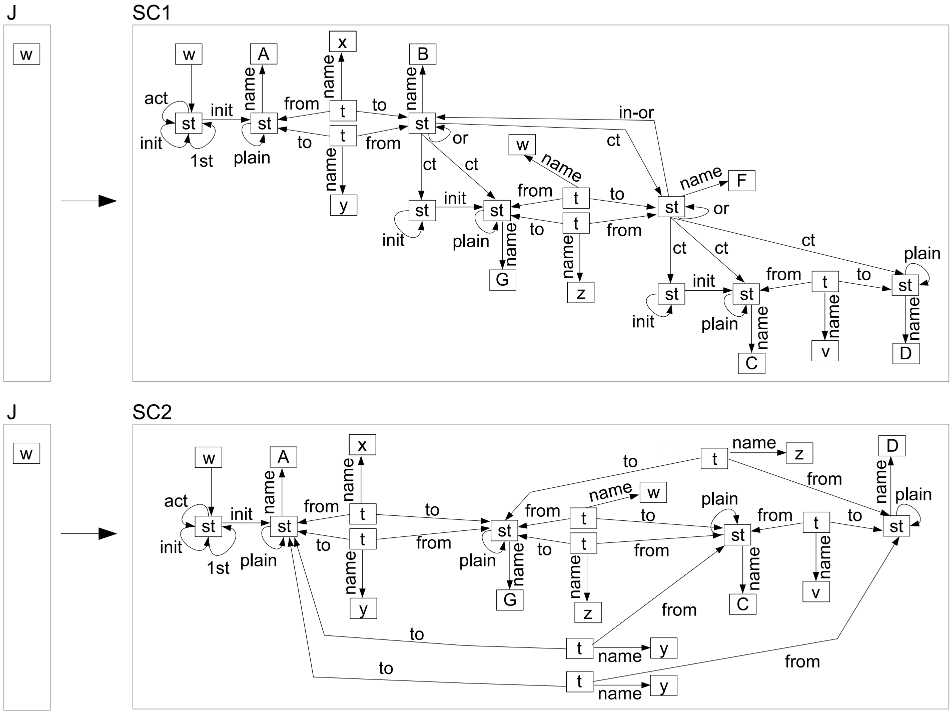


Figure 4.23: Statecharts as graphs with interface.

nization strategy as for finite automata, i.e., the environment has to provide a symbol which fires the corresponding transition in the statechart diagram. The current active state is marked with an act-loop. Whenever a transition t is fired the act-loop is transferred to the state activated by t. The rules in Figure 4.24 define this process. All rules must be instantiated for each event name a $\in$ *Events*, where *Events* is a set of event names. The firing of a transition is triggered by receiving from the environment the corresponding event name (as an edge connecting w-nodes) whenever the source state is active. Fire-plain(a) describes an event firing between plain states. Fire-or-in(a) triggers an event which activates an or-state, i.e., the state connected to its the pseudo state receives the act-loop. For example, firing the x-transition in J $\rightarrow$ SC1 activates the plain state G. An active state inside an or-state may activate a state connected outside the or-state by triggering the corresponding outgoing transition (see Fire-or-out(a)). For diagrams with nested or-states, i.e., or-states within or-states, an active state in a nested or-state may fire an outgoing transition of any of its superstates. For instance, whenever C is active it may fire the y-transition of B via Fire-nested-or-out(a). This rule works also for several levels of nesting because an or-state has an in-or-edge pointing to each superstate in the nested hierarchy. For example, consider a statechart with three nested or-states A, B, C, where A contains B and B contains C. Then we have $\begin{tikzcd} A & B \arrow[l, "\text{in-or}"'] & C \arrow[l, "\text{in-or}"'] \arrow[ll, bend left, "\text{in-or}"] \end{tikzcd}$ and an active substate of C

may fire outgoing transitions of B and A.

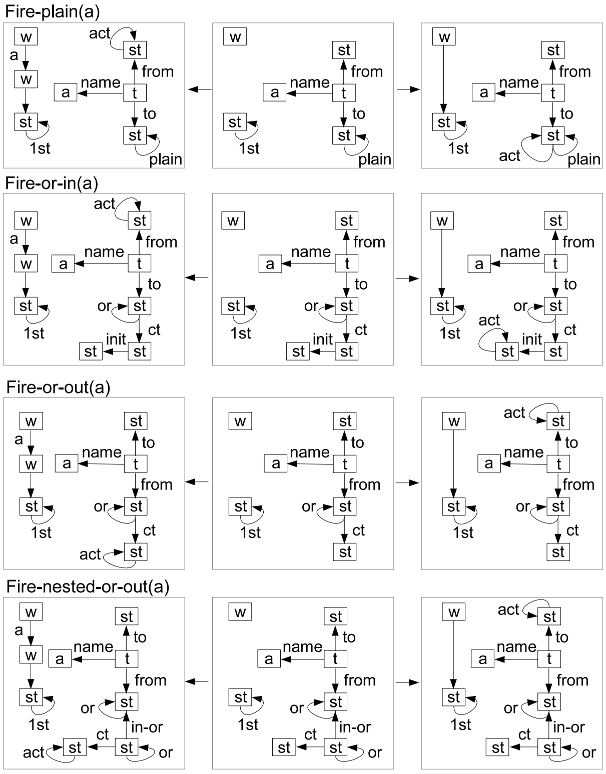


Figure 4.24: OpSem$^{\mathsf{SC}}$: operational semantics for our statecharts.

The equivalence of statechart diagrams can be established via labeled transition systems [MSPT96], i.e., two statecharts are equivalent if their induced labeled transition systems describe the exact same language. Labels are the event names which trigger the execution of transitions. Hence, by using a similar reasoning as for the previous finite automata example, bisimilarity on statecharts implies language equivalence.

Notice in Figures 4.21 and 4.23 that in order to flatten an or-state we have to shift the transitions which activate the or-state to the state connect to its pseudo state and, in addition, for each transition "leaving" the or-state we have to insert a new transition from each of its component real states. The rules in Figure 4.25 specify this process in three transformation layers. To the left of each rule we depict the negative application conditions (if they exist). One or-state is flattened at a time, i.e., if a statechart has n or-states the flattening process has to be executed n times.

In the first refactoring step an or-state is marked for deletion (see Mark). We use "deletion" and "flattening" of or-states as synonyms. Mark is applied to the innermost or-state within a nested-or hierarchy (see the first NAC). An or-state is marked only once for deletion (second NAC) and, finally, exactly one or-state is flattened at a

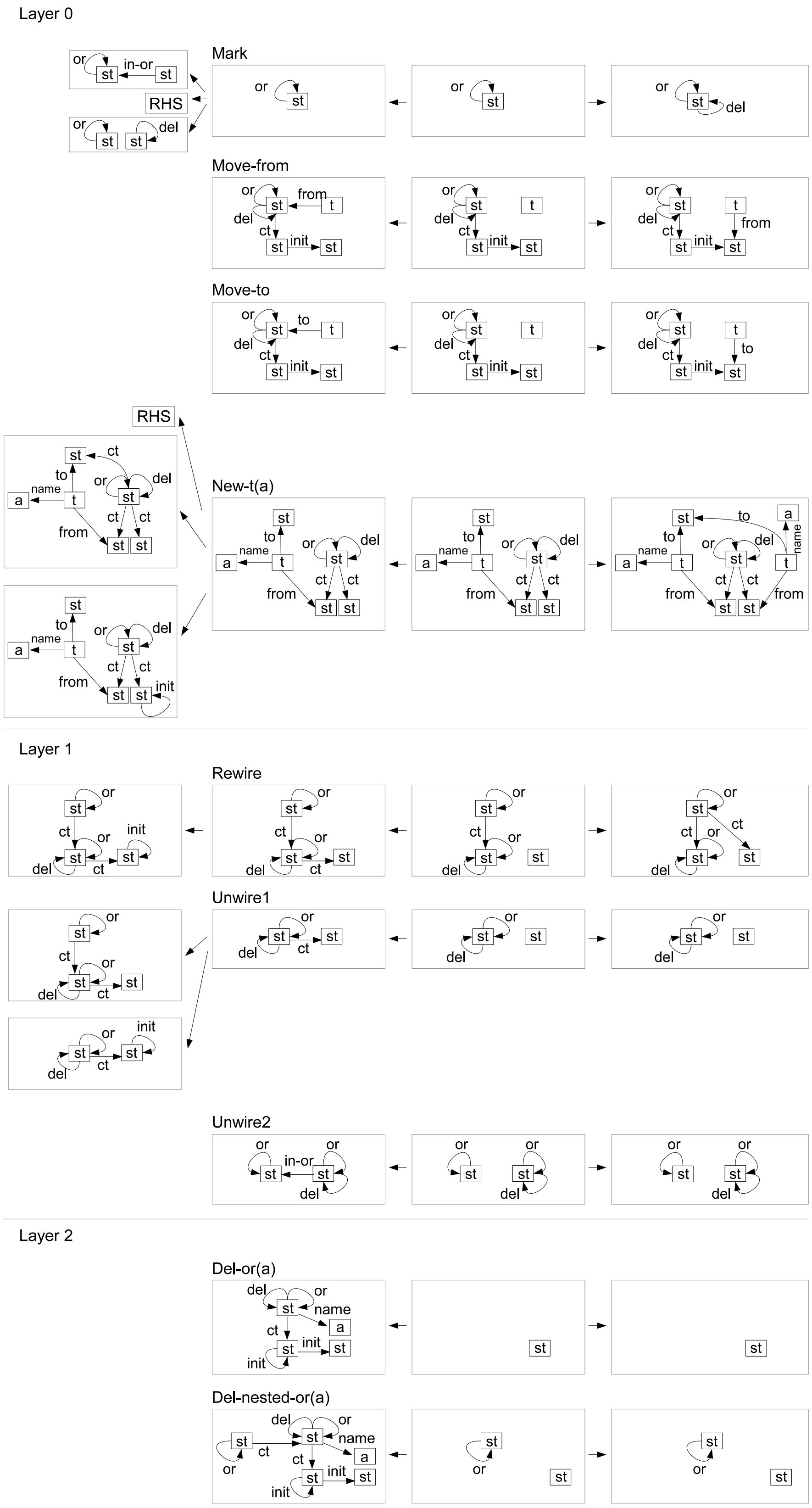


Figure 4.25: Flattening of or-states.

time (third NAC). Move-from and Move-to shift the transitions connected to the or-state to the state indicated by its pseudo state. Note that this makes a transition temporarily cross the border of an or-state. Recall that this is not allowed in our statechart diagrams, but it will help us identify which outgoing transitions (shifted via Move-from) will have to be additionally created to emulate the outgoing transitions of the or-state when it is finally deleted. Hence, New-t(a) creates new transitions to simulate the former outgoing transitions of the original or-state (see example in Figures 4.21 and 4.23). This rule has to be instantiated for every event name $\mathsf{a} \in \mathit{Events}$. Whenever the statechart diagram has nested or-states, Rewire comes into play to reconnect the real states from the or-state to be deleted to its parent or-state. Unwire1 disconnects the component states from the or-state to be deleted. The first NAC constrains Unwire1 to the cases where the or-state is not inside another or-state (in these situations Rewire must be applied). The second NAC says that the pseudo state should remain connected (it will be deleted later on). Unwire2 disconnects the or-state to be deleted from the nested hierarchy of or-states. Finally, Del-or(a) and Del-nested-or(a) remove the or-state and its pseudo state. These two rules have to be instantiated for each state name a employed by or-states. These refactoring rules have been defined in AGG [AGG] and given an hierarchical statechart as input they generate its flat version.

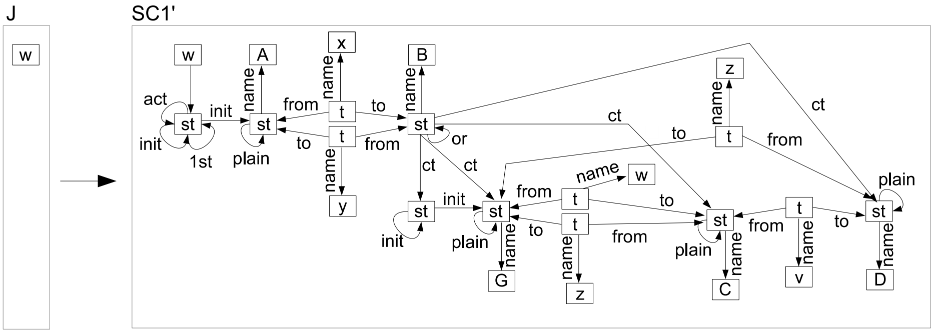


Figure 4.26: Statechart $\mathsf{J} \to \mathsf{SC1}'$.

By applying the refactoring rules to the statechart $\mathsf{J} \to \mathsf{SC1}$ of Figure 4.23 the or-state F is flattened, and hence we obtain $(\mathsf{J} \to \mathsf{SC1}) \Rightarrow^* (\mathsf{J} \to \mathsf{SC1}')$ (see Figure 4.26). Applying the refactoring rules to $\mathsf{J} \to \mathsf{SC1}'$ flattens the or-state B and gives rise to the transformation $(\mathsf{J} \to \mathsf{SC1}') \Rightarrow^* (\mathsf{J} \to \mathsf{SC2})$, where $\mathsf{J} \to \mathsf{SC2}$ is shown in Figure 4.23. The question is whether the entire refactoring transformation $(\mathsf{J} \to \mathsf{SC1}) \Rightarrow^* (\mathsf{J} \to \mathsf{SC2})$ preserved the behavior, i.e., the sequences of transition executions are equal for both statecharts. This is done by calling the procedure **bisimulation_check** to verify $\mathsf{J} \to \mathsf{SC1}$ and $\mathsf{J} \to \mathsf{SC2}$ for bisimilarity w.r.t. $\mathsf{OpSem}^{\mathsf{SC}}$. The algorithm performs similar steps as for the finite automata of the previous section. In Figure 4.27 we show the complete state space product, where the graphs with interface in the three rightmost tuples are not explicitly depicted. Finally, we conclude that $(\mathsf{J} \rightarrow \mathsf{SC1}) \sim_{\mathsf{OpSem}^{\mathsf{SC}}} (\mathsf{J} \rightarrow \mathsf{SC2})$.

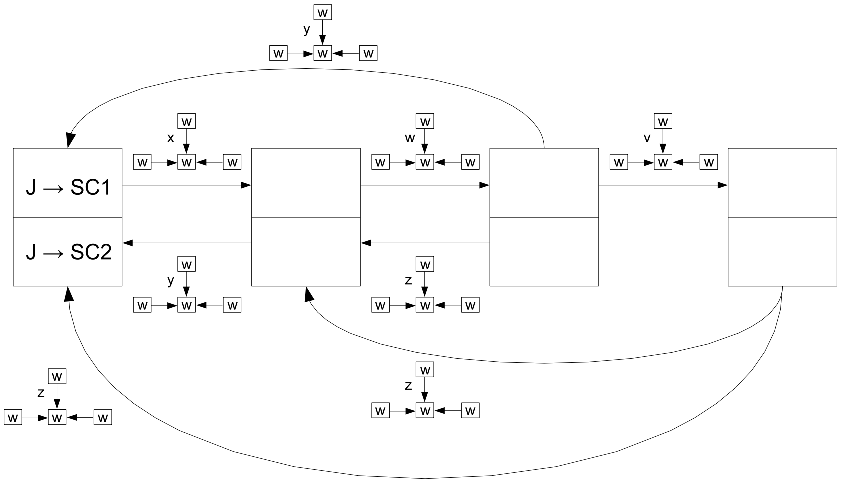


Figure 4.27: Space space product for $\mathsf{J} \rightarrow \mathsf{SC1}$ and $\mathsf{J} \rightarrow \mathsf{SC2}$.

4.6 Conclusions and Future Work

In this chapter we have proposed the algorithms required by the borrowed context machinery in order to check systems specified in terms of graphs for bisimilarity. Our extension of Hirschkoff's on-the-fly bisimulation checking algorithm to the borrowed context setting also handles operational semantics rules with negative application conditions (as in Chapter 3), but our examples given in Sections 4.4 and 4.5 (finite automata and statecharts) have operational rules without NACs.

Compared to Hirschkoff's original algorithm we have made some minor improvements to avoid certain redundant computations. In Chapter 6 we describe some procedures, which were described here at a high level of abstraction, in more details and also give suggestions to make the algorithm more efficient.

The contribution of this chapter is twofold. For borrowed context practitioners we provide a way to mechanize bisimulation proofs, and thus equivalence checking can in principle be carried out automatically. To people interested in model refactoring we have shown how the borrowed context technique can be used to reason about the behavior of instances of a metamodel. The main advantage of this approach is that for every metamodel whose operational semantics can be specified in terms of finite graph transformation productions, the bisimulation checking algorithm can be used to show bisimilarity between models which are instances of this metamodel. Two examples of refactorings were given, namely minimization of finite automata and flattening of hierarchical statecharts, and also checked with the AGG system. In Chapter 5 we develop more elaborate techniques based on borrowed contexts to analyze behavior preservation of refactoring rules.

For our statechart example in order to show the bisimilarity between hierarchical and flat diagrams we had to insert a special edge to the metamodel, namely the in-or-edge, to enable states inside an or-state to directly fire outgoing transitions of other or-states in the nested hierarchy with exactly one single borrowed context step. Recall that flat statecharts also fire their transition via one BC step. Another strategy would be to search the nested-or hierarchy for a particular transition and fire it. However, this could lead to more than one BC step since some "internal" processing would be required to search the transition and then a final step for the firing per se. In these terms, we would need weak instead of strong bisimulation. Note that by constructing a concurrent production [EEPT06, Lam07] p_c induced by this process (i.e. searching and firing) outgoing transitions in a nested-or hierarchy could be fired with one single BC step via p_c. Moreover, this could lead to a similar notion as weak bisimulation in the BC framework. This is an interesting direction for future research.

Last but not least, an interesting future work consists in investigating how close is the relation between bisimilarity on automata seen as labeled transition systems and the bisimilarity obtained via borrowed contexts. This can be done by finding correspondences between the labels of borrowed contexts and the transitions of automata. However this is not an easy task because there exist labels, e.g. independent labels, which does not have a straightforward interpretation in the automata model.

Chapter 5

Behavior Preservation in Model Refactoring

5.1 Motivation

Model transformation [MG06] is concerned with the automatic generation of models from other models according to a transformation procedure which describes how a model in the source language can be "translated" into a model in the target language. Model refactoring is a special case of model transformation where the source and target are instances of the same metamodel. Software refactoring is a modern software development activity, aimed at improving system quality with internal modifications of source code which do not change the observable behavior. In object-oriented programming usually the observable behavior of an object is given by a list of public (visible) properties and methods, while its internal behavior is given by its internal (non-visible) properties and methods.

Graph transformation systems (GTSs) are well-suited to model refactoring and, more generally, model transformation (see [MT04] for the correspondence between refactoring and GTSs). Model refactorings based on GTSs can be found in [RKE07, BEK$^+$06c, HJE06, MTR07]. The left part of Figure 5.1 describes schematically model refactoring via graph transformations. For a graph-based metamodel M, describing, e.g., deterministic finite automata or statecharts, the set $\mathit{Refactoring}^M$ of graph productions describes how to transform models which are instances of the metamodel M. A start graph G^M, which is an instance of the metamodel M, is transformed according to the productions in $\mathit{Refactoring}^M$ (using ordinary DPO transformations), thus producing a graph H^M which is the refactored version of G^M.

A crucial question that must be asked always is whether a given refactoring is behavior-preserving, which means that source and target models have the same ob-

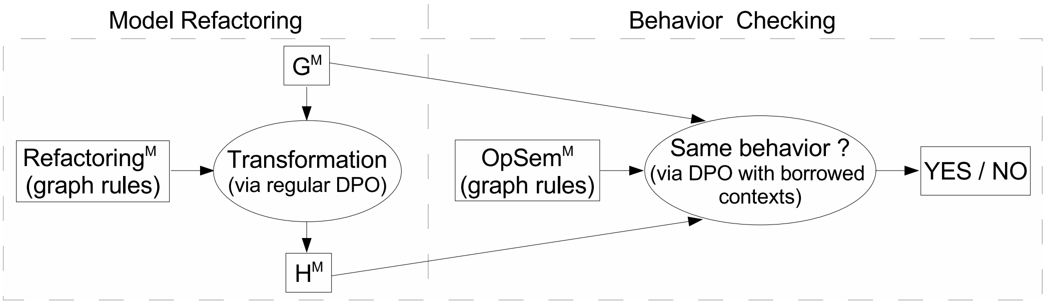

Model Refactoring
Behavior Checking
G^M
Refactoring^M (graph rules)
Transformation (via regular DPO)
H^M
OpSem^M (graph rules)
Same behavior ? (via DPO with borrowed contexts)
YES / NO

Figure 5.1: Model refactoring via graph transformations and behavior preservation.

servable behavior. In practice, proving behavior-preservation is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. On the other hand, formal approaches [vKCKB05, PC07, NK06, GSMD03] have been also employed. A common issue is that behavior preservation is checked only for a certain number of models and their refactored versions. It is difficult though to foresee which refactoring steps are behavior-preserving for all possible instances of the metamodel. Additionally, these approaches are usually tailored to specific metamodels and the transfer to other metamodels would require reconsidering several details. A more general technique is proposed in [BHE08] for analyzing the behavior of a graph production in terms of CSP processes [Hoa85] and trace semantics which guarantees that the traces of a model are a subset of the traces of its refactored version.

In Chapter 4 (and also in [RKE07]) we employed the general framework of borrowed contexts [EK04, EK06, RKE08a] to show that models are bisimilar to their refactored counterparts, which implies behavior preservation. The general idea is illustrated in the right-hand side of Figure 5.1. We define a set $OpSem^M$ of graph productions describing the operational semantics of the metamodel M and use the borrowed context technique to check whether the models G^M and H^M have the same behavior w.r.t. $OpSem^M$. In Chapter 4 we also tailored Hirschkoff's up-to bisimulation checking algorithm [Hir01] to the borrowed context setting and thus equivalence checking can in principle be carried out automatically. The main advantage of this approach is that for every metamodel whose operational semantics can be specified in terms of finite graph transformation productions, the bisimulation checking algorithm can be used to show bisimilarity between models which are instances of this metamodel. However, this technique is also limited to showing behavior preservation only for a fixed number of instances of a metamodel.

In this chapter we go a step further and employ the borrowed context framework in order to check refactoring productions for behavior preservation according to the operational semantics of the metamodel. We call a rule behavior-preserving when its left- and right-hand sides are bisimilar. Thanks to the fact that bisimilarity is a congruence, whenever all refactoring productions preserve behavior, so does every transformation via these rules. In this case, all model instances of the metamodel and their refactored versions exhibit the same behavior. However, refactorings very often involve non-behavior-preserving rules describing intermediate steps of the whole transformation. Given a transformation $G \overset{p_1}{\Rightarrow} H$ via a non-behavior-preserving rule p_1, the basic idea is then to check for the existence of a larger transformation $G \Rightarrow^* H'$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rule applications such that the concurrent production [EEPT06, Lam07] induced by seq is behavior-preserving. Since the concurrent production p_c performs exactly the same transformation $G \overset{p_c}{\Rightarrow} H'$ we can infer that G and H' have the same behavior.

This chapter is based upon our work published in [RLK$^+$08a, RLK$^+$08b], but here we allow operational semantics rules with negative application conditions (NACs). It is structured as follows. Section 5.2 briefly reviews how the DPO approach with borrowed contexts can be used to define the operational semantics of a metamodel. Section 5.3 defines the model refactorings we deal with. An example in the setting of finite automata is given in Section 5.4. In Section 5.5 we define a technique to check refactoring rules for behavior preservation and we discuss an extension to handle non-behavior-preserving rules in model refactoring. These techniques are then applied to the automata example. Finally, we also apply these techniques to the example describing the flattening of statecharts from Chapter 4.

5.2 Operational Semantics via Borrowed Contexts

Here we recall the DPO approach with borrowed contexts [EK04, EK06, RKE08a] and show how it can be used to define the operational semantics of a metamodel M. In this chapter we consider the category of labeled graphs, but the results would also hold for the category of typed graphs or, more generally, for adhesive categories in which pullbacks preserve epimorphisms (see Section 3.3.2 for details). This section is very similar to Section 4.2, however, with a slightly different notation to address borrowed contexts as operational semantics machinery.

In standard DPO [CMR$^+$97], productions rewrite graphs with no interaction with any other entity than the graph itself. In the DPO approach with borrowed contexts [EK06, RKE08a] graphs have interfaces, through which missing parts of left-hand sides can be borrowed from the environment. This leads to open systems which take into account interaction with the external environment.

Definition 5.2.1 (Graphs with Interfaces and Graph Contexts). *A* graph G with interface J *is a morphism* $J \to G$ *and a context consists of two morphisms*

$J \rightarrow E \leftarrow \overline{J}$. The embedding *of a graph with interface $J \rightarrow G$ into a context $J \rightarrow E \leftarrow \overline{J}$ is a graph with interface $\overline{J} \rightarrow \overline{G}$ which is obtained by constructing $\overline{G}$ as the pushout of $J \rightarrow G$ and $J \rightarrow E$.*

$$\begin{tikzcd} J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[dl, dotted] \\ G \arrow[r] & \overline{G} & \end{tikzcd}$$

Observe that the embedding is defined up to isomorphism since the pushout object is unique up to isomorphism.

Definition 5.2.2 (**Metamodel M and Model**)**.** *A* metamodel *M specifies a set of graphs with interface of the form $J \rightarrow G$ (as in Definition 5.2.1). An element of this set is called an instance of the metamodel M, or simply* model.

For example, the metamodel *DFA*, introduced in Section 5.4, describes deterministic finite automata. A model is an automaton $J \rightarrow G$, where G is the automaton and J specifies which parts of G may interact with the environment.

We define DPO rules with negative application conditions (NAC).

Definition 5.2.3 (**NAC and Rule with NAC**)**.** *A* negative application condition *$NAC(n)$ on L is an injective morphism $n \colon L \rightarrow NAC$. An injective match $m \colon L \rightarrow G$ satisfies $NAC(n)$ on L if and only if there is no injective morphism $q \colon NAC \rightarrow G$ with $q \circ n = m$.*

$$\begin{tikzcd} NAC \arrow[dr, "q"', "\shortmid" marking] & L \arrow[l, "n"'] \arrow[d, "m"] \arrow[dl, phantom, "="] \\ & G \end{tikzcd}$$

A negative application condition $NAC(n)$ is called satisfiable *if n is not an isomorphism.*

A rule *$L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ (l, r injective)* with NACs *is equipped with a finite set of negative application conditions $\{n_y \colon L \rightarrow NAC_y\}_{y \in Y}$.*

Note that if $NAC(n)$ is satisfiable then the identity match $id : L \rightarrow L$ satisfies $NAC(n)$. We will assume that for any rule with NACs, the corresponding negative application conditions are all satisfiable, so that the rule is applicable to at least one match (the identity match on its left-hand side).

Definition 5.2.4 (**Set of Operational Semantics Rules**)**.** *Given a metamodel M as in Definition 5.2.2, its operational semantics is defined by a set $OpSem^M$ of graph productions as in Definition 5.2.3.*

Below we define when a borrowed context step is NAC consistent, i.e., the NACs of an operational semantics rule do not forbid the BC step.

Definition 5.2.5 (NAC-Consistent Borrowed Context Step). *Assume that all morphisms are injective. Given $J \to G$ and a production $p\colon L \leftarrow I \to R; \{n_y\colon L \to NAC_y\}_{y\in Y}$ we say that a partial match $pm\colon G \leftarrow D \to L$ leads to a* NAC consistent BC step *with respect to $J \to G$ and p if for the pushout G^+ in the diagram below there is no $q_y\colon NAC_y \to G^+$ with $m = q_y \circ n_y$ for every $y \in Y$.*

$$\begin{tikzcd}
 & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[r, "n_y"] \arrow[d, "m"'] \arrow[dr, phantom, "=" near start] & NAC_y \arrow[dl, "q_y", "/" marking] \\
J \arrow[r] & G \arrow[r] & G^+ &
\end{tikzcd}$$

In the following we need the concept of a pair of jointly surjective morphisms in order to "cover" a graph with two other graphs. That is needed to find possible overlaps between the NACs and the graph G^+ which includes the borrowed context.

Definition 5.2.6 (Jointly Surjective Morphisms). *Two morphisms $f\colon A \to B$ and $g\colon C \to B$ are* jointly surjective *whenever for every pair of morphisms $a, b\colon B \to D$ such that $a \circ f = b \circ f$ and $a \circ g = b \circ g$ it holds that $a = b$.*

In a pushout square the generated morphisms are always jointly surjective. This is a straightforward consequence of the uniqueness of the mediating morphism.

Definition 5.2.7 (Borrowed Context Rewriting for Rules with NACs). *Let $OpSem^M$ be as in Definition 5.2.4. Given a model $J \to G$, a production $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y\in Y}$ ($p \in OpSem^M$) and a partial match $G \leftarrow D \to L$, we say that $J \to G$ reduces to $K \to H$ with transition label $J \to F \leftarrow K; \{F \to N_z\}_{z\in Z}$ if the following holds:*

(i) the BC step is NAC consistent (as in Definition 5.2.5);

(ii) there exist graphs G^+, C and additional injective morphisms such that Diagram (5.1) below commutes and the squares are either pushouts (PO) or pullbacks (PB);

(iii) the set $\{F \to N_z\}_{z\in Z}$ contains exactly the morphisms constructed via Diagram (5.2) (where all morphisms are injective). (That is, there exists a graph M_z such that all squares commute and are pushouts or morphisms are jointly surjective as indicated.)

$$\begin{tikzcd}
 & NAC_y & & \\
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[u, "n_y"'] \arrow[d, "m"] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] \arrow[dr, phantom, "PO"] & G^+ \arrow[dr, phantom, "PB"] & C \arrow[l] \arrow[r] & H \\
J \arrow[u] \arrow[r] & F \arrow[u] \arrow[d] & K \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & N_z & &
\end{tikzcd} \tag{5.1}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u, "n_y"] \arrow[r, "m"'] \arrow[ur, phantom, "\overset{j.surj.}{=}"] & G^+ \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[u]
\end{tikzcd} \tag{5.2}$$

In this case a borrowed context step *(BC step) is feasible and we write:* $(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H)$.

The construction of borrowed context steps is described in two situations: the simplest case is when the rule does not have NACs, whereas in the presence of NACs we have to deal with additional conditions.

When no NACs are present in Definition 5.2.7 Condition (ii) is sufficient, i.e., we can safely discard the other two conditions. In this case consider Diagram (5.1). The upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \rightarrow L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 5.2.7, namely C and F, may not exist. In this case, the rewriting step is not feasible. Note that due to the absence of NACs Diagram (5.1) has no $F \to N_z$.

By taking NACs into account, a BC step can only be executed when G^+ contains no forbidden structure of any negative application condition NAC_y at the match of L (Condition (i)). Additionally, enriched labels are generated (Condition (iii)). The morphisms $F \to N_z$ in Condition (iii) are also called *negative borrowed contexts* and each N_z represents the structures that should not be in G^+ in order to enable the BC step. This extra information in the label is of fundamental importance for the bisimulation game with NACs (Definition 5.2.8), where two graphs with interfaces must not only agree on the borrowed context which enables a transition but also on what should not be offered by the environment in order to perform the transition. The negative borrowed contexts $F \to N_z$ are obtained from $NAC_y \overset{n_y}{\leftarrow} L \overset{m}{\to} G^+ \leftarrow F$ of Diagram (5.1) via Diagram (5.2), where we create all possible overlaps M_z of G^+ and NAC_y in order to check which structures the environment should not provide in order to assure a NAC-consistent BC step. To consider all possible overlaps is necessary in order to take into account that parts of the NAC might already be present in the graph which is being rewritten. Due to the non-uniqueness of the jointly-surjective square one single negative application condition NAC_y may produce more than one negative borrowed context $F \to N_z$.

Whenever the pushout complement in Diagram (5.2) exists, the graph G^+ with borrowed context can be extended to M_z by attaching the negative borrowed context N_z via F. When the pushout complement does not exist, some parts of G^+ which are needed to perform the extension are not "visible" from the environment and no negative borrowed context is generated.

A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 5.2.8 (Bisimulation and Bisimilarity with NACs). *Let $OpSem^M$ be as in Definition 5.2.4 and $\mathcal{R}$ a symmetric relation containing pairs of models $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a* bisimulation with NACs *if, for every $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition*

$$(J \to G) \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H),$$

there exists a model $K \to H'$ and a transition

$$(J \to G') \xrightarrow{J \to F \leftarrow K; \{F \to N_z\}_{z \in Z}} (K \to H')$$

such that $(K \to H)\,\mathcal{R}\,(K \to H')$.

We write $(J \to G) \sim_{OpSem^M} (J \to G')$ (or $(J \to G) \sim (J \to G')$ if the operational semantics is obvious from the context) whenever there exists a bisimulation $\mathcal{R}$ that relates the two instances of the metamodel M. The relation $\sim_{OpSem^M}$ is called bisimilarity with NACs*.*

We often drop "with NACs" from bisimulation (bisimilarity) when it is clear from context.

When defining the operational semantics using the borrowed context framework, it should be kept in mind that the rewriting is based on interactions with the environment, i.e., the environment may provide some information via F to the graph G in order to trigger the rewriting step. For instance, in our finite automata example in Section 5.4 the environment provides a letter to trigger the corresponding transition of the automaton.

An advantage of the borrowed context technique is that the derived bisimilarity is a congruence, which means that whenever a graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace a component of the model with a bisimilar one, without altering its observable behavior.

Theorem 5.2.9 (Bisimilarity based on Productions with NACs is a Congruence). *The bisimilarity $\sim$ of Definition 5.2.8 is a congruence, i.e., it is preserved by embedding into contexts as given in Definition 5.2.1.*

Proof. See Theorem 3.3.10. □

In [EK06] a technique is defined to speed up bisimulation checking, which allows us to take into account only certain labels. A label is considered superfluous and called *independent* if we can add two morphisms $D \to J$ and $D \to I$ to the diagram in Definition 5.2.7 such that $D \to I \to L = D \to L$ and $D \to J \to G = D \to G$. That is, intuitively, the graph G to be rewritten and the left-hand side L overlap only in their interfaces. Transitions with independent labels can be ignored in the bisimulation game, since a matching transition always exists. However, this technique is limited to productions without NACs (see discussion in Section 3.4).

5.3 Refactoring Transformations

Here we define refactoring transformations using DPO rules with negative application conditions (NAC).

Definition 5.3.1 (Transformation). *A* direct transformation $G_0 \overset{p,m}{\Longrightarrow} G_1$ *via a rule p with NACs and an injective match $m\colon L \to G_0$ consists of the double pushout diagram below, where m satisfies all NACs of p.*

$$\begin{tikzcd}
NAC & L \arrow[l] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
 & G_0 & C_0 \arrow[l] \arrow[r] & G_1
\end{tikzcd}$$

Definition 5.3.2 (Layered Refactoring System and Refactoring Rule). *Let metamodel M be as in Definition 5.2.2. A* refactoring rule *is a graph rule as in Definition 5.2.3. A* layered refactoring system *$Refactoring^M$ for the metamodel M consists of k sets $Refactoring^M_i$ $(0 \leq i \leq k-1)$ of refactoring rules. Each set $Refactoring^M_i$ defines a transformation layer.*

Definition 5.3.3 (Refactoring Transformation). *Let Refactoring^M be as in Definition 5.3.2. A* refactoring transformation $t\colon (J \to G_0) \Rightarrow^* (J \to G_n)$ *is a sequence* $(J \to G_0) \overset{p_1}{\Rightarrow} (J \to G_1) \overset{p_2}{\Rightarrow} \cdots \overset{p_n}{\Rightarrow} (J \to G_n)$ *of direct transformations (as in Definition 5.3.1) such that* $p_i \in$ *Refactoring^M and* t *preserves the interface* J*, i.e., for each* i $(0 \leq i < n)$ *there exists an injective morphism* $J \to C_i$ *with* $J \to G_i = J \to C_i \to G_i$ *(see diagram below). Moreover, in* t *each layer applies its rules as long as possible before the rules of the next layer come into play.*

$$\begin{tikzcd}
NAC & L \arrow[l] \arrow[d] \arrow[rd, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[rd, phantom, "PO"] & R \arrow[d] \\
 & G_i & C_i \arrow[l] \arrow[r] & G_{i+1} \\
 & J \arrow[u] \arrow[ru, bend right] \arrow[u, phantom, shift left=4, "="] \arrow[rru, dotted, bend right] & &
\end{tikzcd}$$

Note that refactoring transformations operate only on the internal structure of G_i while keeping the original interface J unchanged.

5.4 Example: Deleting Unreachable States in *DFA*

In Section 4.4 we have presented an algorithm to minimize deterministic finite automata (*DFA*) by merging equivalent states. Here we give a procedure to delete unreachable states in *DFA*.

The metamodel *DFA* describes finite automata represented as graphs with interface as J → DFA1 and J → DFA2 in Figure 5.2, where unlabeled nodes are states and directed labeled edges are transitions. An FS-loop marks a state as final. A W-node has an edge pointing to the current state and this edge points initially to the start state. The W-node is the interface, i.e., the only connection to the environment.

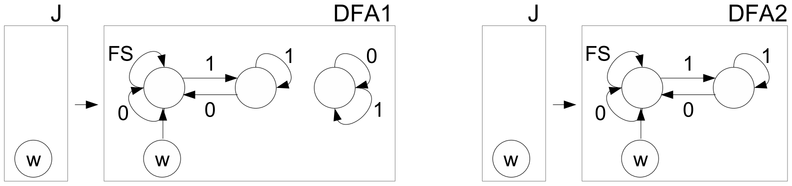
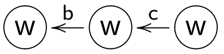


Figure 5.2: Examples of *DFA* as graphs with interface.

The operational semantics for *DFA* is given by a set OpSem$^{\mathsf{DFA}}$ of rules containing Jump(a), Loop(a) and Accept depicted in Figure 5.3. The rules Jump(a), Loop(a) must be defined for each symbol $a \in \Lambda$, where Λ is a fixed alphabet. A *DFA* may change its state according to the rules in $\mathsf{OpSem}^{\mathsf{DFA}}$. The W-node receives a symbol (e.g. 'b') from the environment in form of a b-labeled edge connecting W-nodes, e.g., the string 'bc' is . An acpt-edge between W-nodes marks the end of a string. When such an edge is consumed by a *DFA*, the string previously processed is accepted. An example of an automaton consuming a string is given in Section 4.4.

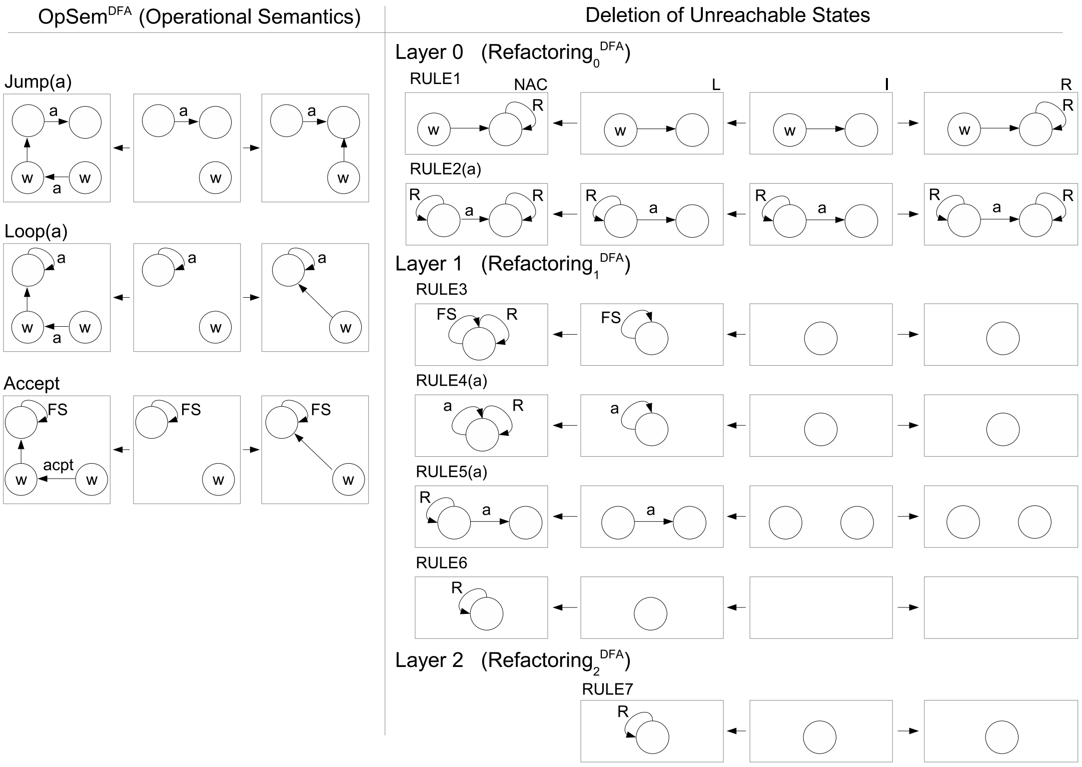

OpSem^DFA (Operational Semantics)
Deletion of Unreachable States
Jump(a)
a
w
Loop(a)
Accept
FS
acpt
Layer 0 (Refactoring_0^DFA)
RULE1
NAC
L
I
R
RULE2(a)
Layer 1 (Refactoring_1^DFA)
RULE3
RULE4(a)
RULE5(a)
RULE6
Layer 2 (Refactoring_2^DFA)
RULE7

Figure 5.3: Operational semantics and a refactoring for *DFA*.

A layered refactoring system for the deletion of unreachable states of an automaton is given in Figure 5.3 on the right. To the left of each rule we depict the NAC (if it exists). The rules are spread over three layers. Rule1 marks the initial state as reachable with an R-loop. Rule2(a) identifies all other states that can be reached from the start state via a-transitions. Layer 1 deletes the loops and the edges of the unreachable states and finally the unreachable states. Layer 2 removes the R-loops.

By applying the refactoring rules above to the automaton J → DFA1 we obtain J → DFA2, where the rightmost state was deleted. By using the bisimulation checking algorithm of Chapter 4 we conclude that J → DFA1 and J → DFA2 are bisimilar w.r.t. $\mathsf{OpSem}^{\mathsf{DFA}}$, which implies language equivalence.

5.5 Behavior Preservation in Model Refactoring

Here we introduce a notion of behavior preservation for refactoring rules and, building on this, we provide some techniques for ensuring behavior preservation in model refactoring.

5.5.1 Refactoring via Behavior-Preserving Rules

For a metamodel M as in Definition 5.2.2 we define behavior preservation as follows.

Definition 5.5.1 (Behavior-Preserving Transformation). *Let OpSem^M be as in Definition 5.2.4. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 5.3.3) is called behavior-preserving when $(J \to G) \sim_{\mathit{OpSem}^M} (J \to H)$.*

In order to check t for behavior preservation we can use Definition 5.2.7 (borrowed context rewriting) to derive transition labels from $J \to G$ and $J \to H$ w.r.t. the rules in OpSem^M.

Observe that behavior preservation in the sense of Definition 5.5.1 is limited to checking specific models. This process is fairly inefficient and, as behavior-preservation is checked for each specific transformation, it can never be exhaustive. A more efficient strategy consists in focussing on the behavior-preservation property at the level of refactoring rules. The general idea is to check for every $p \in \mathit{Refactoring}^M$ whether its left and right-hand sides, seen as graphs with interfaces, are bisimilar, i.e., $(I \to L) \sim (I \to R)$ w.r.t. OpSem^M. Whenever this happens, since bisimilarity is a congruence, any transformation $(J \to G) \overset{p}{\Rightarrow} (J \to H)$ via p preserves the behavior, i.e., $J \to G$ and $J \to H$ have the same behavior.

Definition 5.5.2 (Behavior-Preserving Refactoring Rule). *Let OpSem^M be as in Definition 5.2.4. A refactoring production $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ is* behavior-preserving *whenever $(I \to L) \sim (I \to R)$ w.r.t. OpSem^M.*

Now we can show a simple but important result that says that a rule is behavior-preserving if and only if every refactoring transformation generated by this rule is behavior-preserving.

Proposition 5.5.3. *Let OpSem^M be as in Definition 5.2.4. Then it holds: $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ is behavior-preserving w.r.t. OpSem^M if and only if any refactoring transformation $(J \to G) \overset{p}{\Rightarrow} (J \to H)$ (as in Definition 5.3.3) is behavior-preserving, i.e., $(J \to G) \sim_{\mathit{OpSem}^M} (J \to H)$.*

Proof. "$\Rightarrow$": Assume a refactoring transformation $(J \to G) \overset{p}{\Rightarrow} (J \to H)$ as depicted below. Since p is behavior-preserving we know that $(I \to L) \sim_{\mathit{OpSem}^M} (I \to R)$

(Definition 5.5.2). Observe that $J \rightarrow G$ and $J \rightarrow H$ are $I \rightarrow L$ and $I \rightarrow R$ respectively inserted into the context $I \rightarrow C \leftarrow J$, which implies $(J \rightarrow G) \sim_{OpSem^M} (J \rightarrow H)$ by Theorem 5.2.9 (bisimilarity is a congruence).

$$\begin{tikzcd}
NAC_y & L \arrow[l] \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
& G & C \arrow[l] \arrow[r] & H \\
& J \arrow[u] \arrow[ur, "="] \arrow[urr, dotted] & &
\end{tikzcd}$$

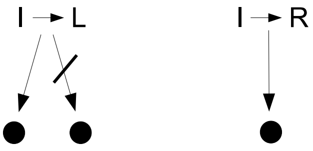
"$\Leftarrow$": Assume that any refactoring transformation via p is behavior-preserving. By assumption, for rule p all NACs in $\{L \rightarrow NAC_y\}_{y \in Y}$ are satisfiable (see Definition 5.2.3) and thus p is applicable to the identity match $id\colon L \rightarrow L$. As a result of the application of p to $I \rightarrow L$, we obtain $(I \rightarrow L) \overset{p}{\Rightarrow} (I \rightarrow R)$. Such refactoring is behavior-preserving, by hypothesis, and thus $(I \rightarrow L) \sim_{OpSem^M} (I \rightarrow R)$. Hence, by Definition 5.5.2, p is behavior-preserving.

□

Remark 5.5.4. *The fact that the previous proposition also holds for rules with NACs, even though Definition 5.5.2 does not take NACs into account for behavior-preservation purposes, of course does not imply that negative application conditions for refactoring rules are unnecessary in general. They are needed in order to constrain the applicability of rules, especially of those rules that are not behavior-preserving, or rather, are only behavior-preserving when applied in certain contexts. As a direction of future work, we plan to study congruence results for restricted classes of contexts. This will help to better handle refactoring rules with NACs.*

Remark 5.5.5. *This is a more technical version of Remark 5.5.4.*

In general the NACs of a refactoring rule $p\colon L \leftarrow I \rightarrow R; \{L \rightarrow NAC_y\}_{y \in Y}$ *may play a more prominent role in behavior preservation. Definition 5.5.2 takes into account only the morphisms* $I \rightarrow L$ *and* $I \rightarrow R$ *in order to determine the behavior-preservation property of p, i.e., whenever* $(I \rightarrow L) \sim_{OpSem^M} (I \rightarrow R)$ *holds then this is valid in any context. Recall that behavior preservation is obtained via Definitions 5.2.7 and 5.2.8 but it is important to make clear that the NACs present in Definition 5.2.7 belong to the operational semantics rule. The applicability of a refactoring rule p is limited to certain graphs which respect the NACs (of p), and hence it is not necessary to consider* $(I \rightarrow L) \sim_{OpSem^M} (I \rightarrow R)$ *valid in all contexts, but rather only in contexts that satisfy the NACs. By the current definition of borrowed contexts if there exists at least one context in which* $(I \rightarrow L) \sim_{OpSem^M} (I \rightarrow R)$ *does not hold then the refactoring rule p is not behavior preserving. In several cases, the* *exact contexts in which* $(I \to L) \sim_{OpSem^M} (I \to R)$ *does not hold are those that are ruled out by the NACs of p, and hence p preserves behavior in restricted classes of contexts. Below we schematically depict this situation, where a refactoring rule* $\mathsf{p}\colon \mathsf{L} \leftarrow \mathsf{I} \to \mathsf{R}; \{\mathsf{L} \to \mathsf{NAC_y}\}_{\mathsf{y} \in \mathsf{Y}}$ *is checked for behavior preservation.*



Two transition labels are derived from $\mathsf{I} \to \mathsf{L}$ *and only one from* $\mathsf{I} \to \mathsf{R}$*, and thus* p *is not behavior preserving because the label on the right-hand side of* $\mathsf{I} \to \mathsf{L}$ *does not have a matching partner in* $\mathsf{I} \to \mathsf{R}$*. In the following we consider a hypothetical borrowed context theory where graphs with interfaces are equipped with NACs which forbid BC steps whose contexts do not satisfy them. In the refactoring setting these NACs are the ones from a refactoring rule. Using this hypothetical BC rewriting the second label of* $\mathsf{I} \to \mathsf{L}$ *is not derivable because its borrowed context violates the NAC (of the graph* $\mathsf{I} \to \mathsf{L}$*). Thus, its left label and the label from* $\mathsf{I} \to \mathsf{R}$ *can be properly matched and we could infer that* p *is behavior preserving in the restricted class of contexts. An extension of the BC theory to cope with this kind of graphs with NACs is an interesting future work.*

Theorem 5.5.6 (**Refactoring via Behavior-Preserving Rules**). *Let* $OpSem^M$ *and* $Refactoring^M$ *be as in Definitions 5.2.4 and 5.3.2. If each rule in* $Refactoring^M$ *is behavior-preserving w.r.t.* $OpSem^M$ *then any refactoring transformation* $(J \to G_0) \Rightarrow^* (J \to G_n)$ *via these rules is behavior-preserving.*

Proof. Assume that $Refactoring^M$ contains only behavior-preserving rules with respect to $OpSem^M$. Let $t\colon (J \to G_0) \Rightarrow^* (J \to G_n)$ be a sequence of refactoring transformations as in Definition 5.3.3. So, for every i $(0 \leq i < n)$ we have a direct refactoring transformation $(J \to G_i) \overset{p}{\Rightarrow} (J \to G_{i+1})$ via a rule $p\colon L \leftarrow I \to R; \{L \to NAC_y\}_{y \in Y}$ in $Refactoring^M$. Since p is behavior-preserving we have: $(I \to L) \sim_{OpSem^M} (I \to R)$. From the first part of Proposition 5.5.3 we can infer that $(J \to G_i) \sim_{OpSem^M} (J \to G_{i+1})$. Furthermore, since bisimilarity $\sim$ is an equivalence relation, by transitivity $(J \to G_0) \sim_{OpSem^M} (J \to G_n)$.

□

Our examples in this chapter do not make use of NACs for the operational semantics rules, even though the techniques proposed here can handle them.

Example 5.5.1
We check the rules in $\mathsf{Refactoring}_\mathsf{i}^{DFA}$ ($\mathsf{i} = 0, 1, 2$) from Section 5.4 for behavior preservation. We begin with $\mathsf{Refactoring}_\mathsf{0}^{DFA}$ (Layer 0). For RULE1: $\mathsf{NAC} \leftarrow \mathsf{L} \leftarrow \mathsf{I} \rightarrow \mathsf{R}$ we derive transition labels from $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ w.r.t. $\mathsf{OpSem}^\mathsf{DFA}$ (where $\mathsf{L} \rightarrow \mathsf{NAC}$ plays no role). On the left-hand side of Figure 5.4 we schematically depict the first steps in their respective labeled transition systems (LTS), where each partner has three choices. Independent labels exist in both LTSs but are not illustrated below.

The derivation of label L_1 for $\mathsf{I} \rightarrow \mathsf{R}$ is shown on the right. Since $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ (and their successors) can properly mimic each other via a bisimulation we can conclude that $(\mathsf{I} \rightarrow \mathsf{L}) \sim_{\mathsf{OpSem}^\mathsf{DFA}} (\mathsf{I} \rightarrow \mathsf{R})$. The intuitive reason for this is that the R-loop, which is added by this rule, does not have any meaning in the operational semantics and is hence "ignored" by $\mathsf{OpSem}^\mathsf{DFA}$.

Analogously, RULE2(a) and the rule in Layer 2 are behavior-preserving as well. Hence, we can infer that every transformation via the rules of Layer 0 and Layer 2 preserves the behavior. On the other hand, all rules in Layer 1, except for RULE6, are not behavior-preserving. Note that RULE6 is only behavior-preserving because of the dangling condition. Thus, when a transformation is carried out via non-behavior-preserving rules of Layer 1 we cannot be sure whether the behavior is preserved.

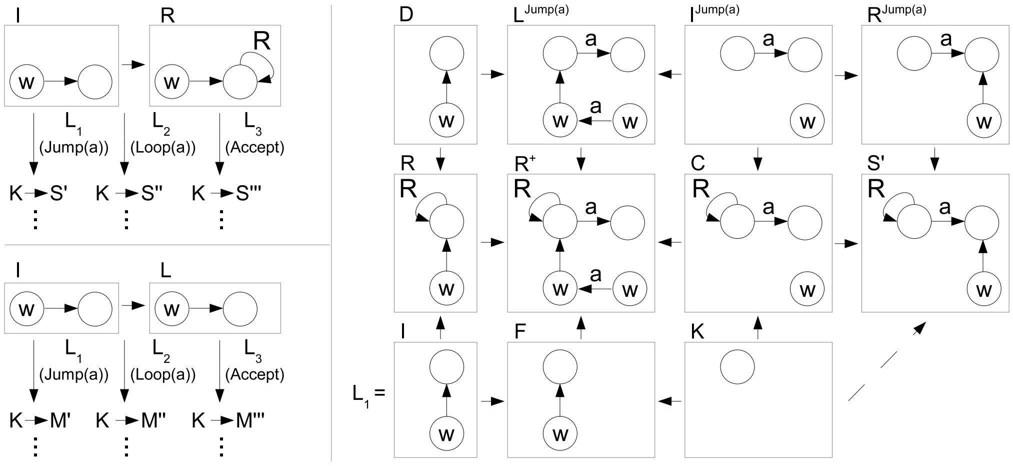


Figure 5.4: Labeled transition systems for RULE1 and a label derivation via Jump(a).

5.5.2 Handling Non-Behavior-Preserving Rules

For refactoring transformations based on non-behavior-preserving rules the technique of Section 5.5.1 does not allow to establish if the behavior is preserved.

Very often there are refactoring rules representing intermediate transformations that indeed are not behavior-preserving. Still, when considered together with neighboring rules, they could induce a concurrent production [EEPT06, Lam07] p_c, corresponding to a larger transformation, which preserves the behavior. For a transformation $t\colon (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ the concurrent production $p_c\colon L_c \leftarrow I_c \to R_c; \{L_c \to NAC_c^y\}_{y \in Y}$ induced by t performs exactly the same transformation $(J \to G) \overset{p_c}{\Rightarrow} (J \to H')$ in one step. Moreover, p_c can only be applied to $(J \to G)$ if the concurrent NACs in $\{L_c \to NAC_c^y\}_{y \in Y}$ are satisfied. This is the case if and only if every NAC of the rules in t is satisfied. The basic idea is now to check for a transformation $(J \to G) \overset{p_1}{\Rightarrow} (J \to H)$ based on a non-behavior-preserving rule p_1 whether there exists such a larger transformation $t : (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rules such that the concurrent production induced by t is behavior preserving. Then we can infer that $J \to G$ and $J \to H'$ have the same behavior.

This is made formal by the notion of safe transformation and the theorem below.

Definition 5.5.7 (**Safe Transformation**). *Let $OpSem^M$ be as in Definition 5.2.4. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 5.3.3) is called* safe *if it induces a behavior-preserving concurrent production w.r.t. $OpSem^M$.*

Theorem 5.5.8 (**Safe Transformations preserve Behavior**). *Let $OpSem^M$ and $Refactoring^M$ be as in Definitions 5.2.4 and 5.3.2, and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. If t is safe, then t is behavior-preserving, i.e., $(J \to G) \sim (J \to H)$.*

Proof. Let t be a safe transformation. By definition it induces a concurrent production $p_c : L_c \leftarrow I_c \to R_c; \{L_c \to NAC_c^y\}_{y \in Y}$ (see [EEPT06, Lam07] for the details of the construction) which is behavior-preserving, i.e., $(I_c \to L_c) \sim_{OpSem^M} (I_c \to R_c)$. Due to the Concurrency Theorem with NACs [Lam07] the concurrent production p_c induced by t is applicable to $J \to G$ with the same result $J \to H$. Since p_c preserves behavior it follows from Theorem 5.5.6 that $(J \to G) \sim_{OpSem^M} (J \to H)$.

□

In order to prove that a refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ is safe (and thus behavior-preserving), we can look for a split $t^{sp}\colon G \Rightarrow^* H_1 \Rightarrow^* \cdots \Rightarrow^* H_n \Rightarrow^* H$ (interfaces are omitted) of t where each step ($\Rightarrow^*$) induces a behavior-preserving concurrent production (see Definition 5.5.9). In fact, as shown below, if and only if such split exists we can guarantee that t preserves behavior (Theorem 5.5.10).

Definition 5.5.9 (**Safe Transformation Split**). *Let $OpSem^M$ be as in Definition 5.2.4 and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation (as in*

Definition 5.3.3). A split *of t is obtained by cutting t into a sequence of subtransformations $t^{sp}\colon (J \to G) \Rightarrow^* (J \to H_1) \Rightarrow^* \cdots \Rightarrow^* (J \to H_n) \Rightarrow^* (J \to H)$. A transformation split t^{sp} is* safe *if each step ($\Rightarrow^*$) is safe.*

In Section 5.5.3 we present a simple search strategy for safe splits. More elaborate ones are part of future work.

Theorem 5.5.10. *Let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. Then t is safe if and only if it admits a safe split.*

Proof. "$\Rightarrow$": The fact that t is safe implies a behavior-preserving concurrent production p_c such that $(J \to G) \overset{p_c}{\Rightarrow} (J \to H)$. In this case the split t^{sp} has only the step $(J \to G) \overset{p_c}{\Rightarrow} (J \to H)$ which is safe.

"$\Leftarrow$": if t admits a safe split then there exists a split $t^{sp}\colon G \Rightarrow^* H_1 \Rightarrow^* \cdots \Rightarrow^* H_n \Rightarrow^* H$ (the interfaces J are omitted) with $n+1$ steps such that the i-th step (for $1 \leq i \leq n+1$) is safe (Definition 5.5.9). Each of these steps induces a behavior-preserving concurrent production $p_c^i\colon L_c^i \leftarrow I_c^i \to R_c^i; \{L_c^i \to NAC_c^{y,i}\}_{y \in Y}$. We know that given two behavior-preserving rules p_1 and p_2 as depicted below the induced concurrent production p_c is also behavior-preserving since bisimilarity is a congruence (Theorem 5.2.9), i.e., $(I_1 \to L_1) \sim (I_1 \to R_1)$ implies $(I_c \to L_c) \sim (I_c \to E)$ and $(I_2 \to L_2) \sim (I_2 \to R_2)$ implies $(I_c \to E) \sim (I_c \to R_c)$ and so we have $(I_c \to L_c) \sim (I_c \to R_c)$.

$$\begin{tikzcd}
 & NAC_c^{y,1} & & & & NAC_c^{y,2} & & \\
 & L_1 \arrow[u] \arrow[d] \arrow[dr, phantom, "PO"] & I_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[dr] & & L_2 \arrow[u] \arrow[dl] & I_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R_2 \arrow[d] \\
NAC_c^y & L_c \arrow[l] & I_1' \arrow[l] \arrow[rr] & & E & & I_2' \arrow[ll] \arrow[r] & R_c \\
 & & & & I_c \arrow[ull] \arrow[urr] & & &
\arrow[from=2-3, to=3-5, phantom, "PO"]
\arrow[from=2-6, to=3-7, phantom, "PO"]
\arrow[from=3-5, to=4-5, phantom, "PB"]
\end{tikzcd}$$

Thus, since every p_c^i is behavior-preserving we can infer that the concurrent production induced by t^{sp} is also behavior-preserving which implies that t is safe.

□

Observe that, instead, the following does not hold in general: if $t\colon (J \to G) \Rightarrow^* (J \to H)$ and $(J \to G) \sim_{OpSem^M} (J \to H)$ then t is safe. Consider for instance RULE5(a) in Figure 5.3. As remarked, it is in general not behavior-preserving, but when, by coincidence, it removes a transition that is unreachable from the start state, the original automaton and its refactored version are behaviorally equivalent.

5.5.3 Ensuring Behavior Preservation

In this section we describe how the theory presented in this chapter can be applied. Note that our results would allow us to automatically prove behavior preservation only in special cases, while, in general, such mechanized proofs will be very difficult. Hence here we will suggest a "mixed strategy", which combines elements of automatic verification and the search for behavior-preserving rules, in order to properly guide refactorings.

More specifically, a given model $J \to G$ can be refactored by applying the rules in $\mathit{Refactoring}^M$ in an automatic way, where the machine chooses non-deterministically the rules to be applied, or in a user-driven way, where for each transformation the machine provides the user with a list of all applicable rules together with their respective matches and ultimately the user picks one of them. The main goal is then to tell the user whether the refactoring is behavior-preserving.

The straightforward strategy to accomplish the goal above is to transform $J \to G$ applying only behavior-preserving rules. This obviously guarantees that the refactoring preserves behavior. However if a non-behavior-preserving rule p is applied we can no longer guarantee behavior preservation. Still, by proceeding with the refactoring, namely by performing further transformations, we can do the following: for each new transformation added to the refactoring we compute the induced concurrent production for the transformation which involves the first non-behavior-preserving rule p and the subsequent ones. If this concurrent production is behavior-preserving we can again guarantee behavior preservation for the refactoring since the refactoring admits a safe split (see Theorem 5.5.10).

The strategy above is not complete since behavior preservation could be ensured by the existence of complex safe splits which the illustrated procedure is not able to find. We already have preliminary ideas for more sophisticated search strategies, but they are part of future work. Note however, that this strategy can reduce the proof obligations, since we do not have to show behavior preservation between the start and end graph of the refactoring sequence (which may be huge), but we only have to investigate local updates of the model.

Example 5.5.2
Consider the automaton $\mathsf{J} \to \mathsf{DFA1}$ of Section 5.4. By applying the behavior-preserving rules of $\mathsf{Refactoring}_0^{DFA}$ ($\mathsf{Layer\ 0}$) we obtain $\mathsf{J} \to \mathsf{DFA}_1^0$ depicted in Figure 5.5 (the interface J is omitted). Since $\mathsf{Refactoring}_0^{DFA}$ contains only behavior-preserving rules by Theorem 5.5.6 it holds that $(\mathsf{J} \to \mathsf{DFA1}) \Rightarrow^* (\mathsf{J} \to \mathsf{DFA}_1^0)$ preserves the behavior. No more rules in $\mathsf{Refactoring}_0^{DFA}$ can be applied, i.e., the computation of $\mathsf{Layer\ 0}$ terminates.

Now the rules of $\mathsf{Refactoring}_1^{DFA}$ ($\mathsf{Layer\ 1}$) come into play. Recall that all rules in $\mathsf{Refactoring}_1^{DFA}$ are non-behavior-preserving, except for $\mathsf{RULE6}$. This set contains $\mathsf{RULE4(0)}$ and $\mathsf{RULE4(1)}$ which are appropriate instantiations of $\mathsf{RULE4(a)}$. After the transformation $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1)$ we can no longer guarantee behavior-preservation since $\mathsf{RULE4(0)}$ has been applied. From now on we follow the strategy previously described to look for a behavior-preserving concurrent production. We perform the step $(\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$, build a concurrent production p_c induced by the transformation $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$ and, by checking p_c for behavior-preservation, we find out that it is not behavior-preserving. We then continue with $(\mathsf{J} \to \mathsf{DFA}_1^2) \overset{\mathsf{RULE6}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^3)$, build p'_c (Figure 5.6), induced by the transformation beginning at $\mathsf{J} \to \mathsf{DFA}_1^0$ and check it for behavior-preservation. By Definition 5.2.7 (borrowed context rewriting) we derive transition labels from $\mathsf{I_c} \to \mathsf{L_c}$ and $\mathsf{I_c} \to \mathsf{R_c}$ which are properly matched as in Definition 5.2.8 (bisimulation). Hence p'_c is behavior-preserving by Proposition 5.5.3 and so we can once again guarantee behavior preservation (Theorem 5.5.8).

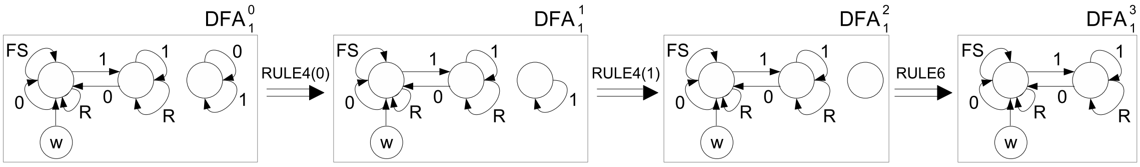


Figure 5.5: Refactoring transformation.

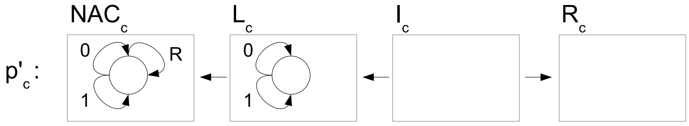


Figure 5.6: Concurrent production p'_c induced from Figure 5.5.

Finally, no more rules of $\mathsf{Refactoring}_1^{DFA}$ are applicable to $\mathsf{J} \to \mathsf{DFA}_1^3$. The behavior-preserving rule in $\mathsf{Refactoring}_2^{DFA}$ ($\mathsf{Layer\ 2}$) comes into play and performs a transformation $(\mathsf{J} \to \mathsf{DFA}_1^3) \overset{\mathsf{RULE7}}{\Longrightarrow}_2 (\mathsf{J} \to \mathsf{DFA2})$, where the final automaton is depicted in Section 5.4 ($\mathsf{DFA2}$). Concluding, since we have found a safe split given by $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{p}'_\mathsf{c}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^3)$ (Figure 5.7) for the transformation via non-behavior-preserving rules we can infer that $\mathsf{J} \to \mathsf{DFA1}$ and $\mathsf{J} \to \mathsf{DFA2}$ have the same behavior.

$$\begin{tikzcd}
(\mathsf{J} \to \mathsf{DFA1}) \arrow[r, Rightarrow, "*"] & (\mathsf{J} \to \mathsf{DFA}_1^0) \arrow[r, Rightarrow, "\mathsf{RULE4(0)}"] \arrow[rrr, Rightarrow, bend right, "\mathsf{p}'_\mathsf{c}"] & (\mathsf{J} \to \mathsf{DFA}_1^1) \arrow[r, Rightarrow, "\mathsf{RULE4(1)}"] & (\mathsf{J} \to \mathsf{DFA}_1^2) \arrow[r, Rightarrow, "\mathsf{RULE6}"] & (\mathsf{J} \to \mathsf{DFA}_1^3) \arrow[r, Rightarrow, "\mathsf{RULE7}"] & {}_2(\mathsf{J} \to \mathsf{DFA2}\ )
\end{tikzcd}$$

Figure 5.7: Transformation $(\mathsf{J} \to \mathsf{DFA_1}) \Rightarrow^* (\mathsf{J} \to \mathsf{DFA2})$ and safe splits (via p'_c) .

Intuitively, the concurrent production is behavior-preserving, since it deletes an entire connected component that is not linked to the rest of the automaton. Note that due to the size of the components involved it can be much simpler to check such transformation units rather than the entire refactoring sequence.

In addition, it would be useful if the procedure above could store the induced concurrent productions which are behavior-preserving into $\mathit{Refactoring}^M$ for later use. By doing so the user knows which combination of rules leads to behavior-preserving concurrent productions. Similarly, the user could also want to know which combination of rules leads to non-behavior-preserving concurrent productions. Of course, in the latter case the concurrent productions are just stored but do not engage in any refactoring transformation. It is important to observe that we store into $\mathit{Refactoring}^M$ only concurrent productions which are built with rules within the same layer (as in Example 5.5.2). For more complex refactorings, such as the flattening of hierarchical statecharts in Example 5.5.3, a behavior-preserving concurrent production p_c exists only when it is built from a transformation involving several layers. In this case, p_c is built and checked for behavior preservation but not stored for later use.

For the cases where a layer $\mathit{Refactoring}_i^M$ of $\mathit{Refactoring}^M$ is terminating and confluent it is then important to guarantee that adding concurrent productions to the refactoring layer does not affect these properties.

Theorem 5.5.11. *Let $\mathit{Refactoring}_i^M$ be as in Definition 5.3.2 and $R_i^{p_c}$ be a set containing concurrent productions p_c built from $p, q \in \mathit{Refactoring}_i^M \cup R_i^{p_c}$. Then whenever $\mathit{Refactoring}_i^M$ is confluent and terminating it holds that $\mathit{Refactoring}_i^M \cup R_i^{p_c}$ is also terminating and confluent.*

Proof. Suppose that an infinite graph transformation sequence $G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \dots$ exists via rules in $\mathit{Refactoring}_i^M \cup R_i^{p_c}$ (the interfaces J are omitted). Due to the analysis part of the Concurrency Theorem with NACs [Lam07] each direct transformation in this sequence via a concurrent rule $p_c \in R_i^{p_c}$ can be analyzed to a sequence of direct transformations with the same result via rules in $\mathit{Refactoring}_i^M$. Therefore, there would also exist an infinite graph transformation sequence via rules of $\mathit{Refactoring}_i^M$, but this is a contradiction since $\mathit{Refactoring}_i^M$ is terminating.

Now consider the transformation sequences $H_1 \overset{*}{\Leftarrow} G \overset{*}{\Rightarrow} H_2$ via productions in $\mathit{Refactoring}_i^M \cup R_i^{p_c}$. We can argue analogously. Due to the analysis part of the Concurrency Theorem with NACs [Lam07] each direct transformation in these sequences via a concurrent rule $p_c \in R_i^{p_c}$ can be analyzed to a sequence of direct transformations with the same result via rules in $\mathit{Refactoring}_i^M$. Thus, we obtain transformation sequences $H_1 \overset{\prime *}{\Leftarrow} G \overset{*\prime}{\Rightarrow} H_2$ via rules in $\mathit{Refactoring}_i^M$. Since $\mathit{Refactoring}_i^M$ is confluent there exist two transformation sequences $H_1 \overset{*}{\Rightarrow} X \overset{*}{\Leftarrow} H_2$ via rules in $\mathit{Refactoring}_i^M$. Hence, the original sequences $H_1 \overset{*}{\Leftarrow} G \overset{*}{\Rightarrow} H_2$ are also confluent. □

For the case where layer $Refactoring_i^M$ is terminating and confluent another interesting and useful fact holds: assume that we fix a start graph G_0 and we can show that *some* (terminating) transformation, beginning with G_0 allows a behavior-preserving split. Then clearly all transformations starting from G_0 are behavior-preserving since they result in the same final graph H.

In the following example we apply the techniques developed in this chapter to reason about the refactoring rules describing the flattening of statecharts from Section 4.5.

Example 5.5.3
Let $\mathsf{OpSem}^{\mathsf{SC}}$ be a set containing the rules of Figure 4.24 describing the operational semantics of statecharts diagrams. Each layer in Figure 4.25 forms a set of refactoring rules $\mathsf{Refactoring}_\mathsf{i}^{\mathsf{SC}}$ $(\mathsf{i} = 0, 1, 2)$. By checking these refactoring rules for behavior preservation we find out that all rules are not behavior-preserving, except for Mark, Del-or(a) and Del-nested-or(a).

In Figure 5.8 we depict a statechart with an or-state and its respective flat version. These statecharts are given as graphs with interface in Figure 5.9.

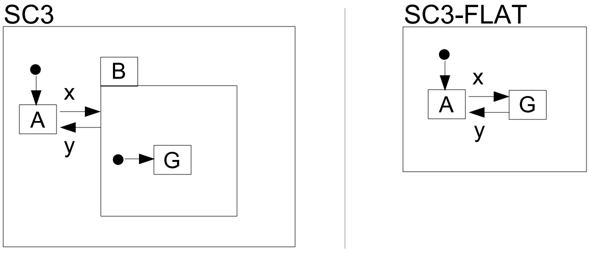


Figure 5.8: Statechart SC3 and its flattened version SC3-FLAT.

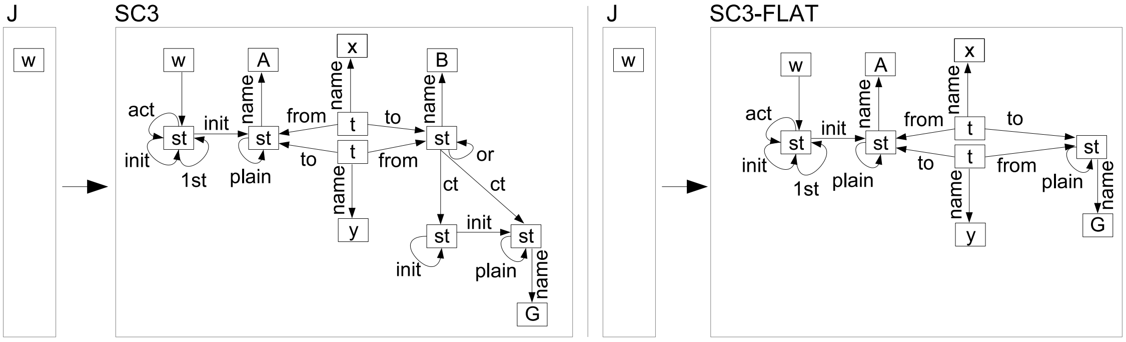


Figure 5.9: Statecharts as graphs with interface.

We then apply the rules in $\mathsf{Refactoring}^{\mathsf{SC}}$ to $\mathsf{J} \to \mathsf{SC3}$. The first transformation step $(\mathsf{J} \to \mathsf{SC3}) \overset{\mathsf{Mark}}{\Longrightarrow} (\mathsf{J} \to \mathsf{SC3}^1)$ (see Figure 5.10) is behavior-preserving since Mark preserves behavior. The second refactoring step is $(\mathsf{J} \to \mathsf{SC3}^1) \overset{\mathsf{Move\text{-}from}}{\Longrightarrow} (\mathsf{J} \to \mathsf{SC3}^2)$ and we can no longer guarantee behavior preservation since Move-from does not preserve behavior. From $\mathsf{J} \rightarrow \mathsf{SC3}^1$ on we will use the simple search strategy previously defined in order to find a safe split. Now we perform $(\mathsf{J} \rightarrow \mathsf{SC3}^2) \overset{\mathsf{Move\text{-}to}}{\Longrightarrow} (\mathsf{J} \rightarrow \mathsf{SC3}^3)$ and build the concurrent production p_c induced by $(\mathsf{J} \rightarrow \mathsf{SC3}^1) \Rightarrow^2 (\mathsf{J} \rightarrow \mathsf{SC3}^3)$. By checking p_c for behavior preservation we find that it does not preserve behavior. No more rule from Layer 0 is applicable and so the rules of Layer 1 come into play. The transformation $(\mathsf{J} \rightarrow \mathsf{SC3}^3) \overset{\mathsf{Unwire1}}{\Longrightarrow} (\mathsf{J} \rightarrow \mathsf{SC3}^4)$ takes place, we build a concurrent production including this last transformation, but it is not yet behavior preserving. Layer 1 terminates. The last transformation is $(\mathsf{J} \rightarrow \mathsf{SC3}^4) \overset{\mathsf{Del\text{-}or(B)}}{\Longrightarrow} (\mathsf{J} \rightarrow \mathsf{SC3\text{-}FLAT})$. We build the concurrent production p'_c (shown in Figure 5.11) induced by the transformation $(\mathsf{J} \rightarrow \mathsf{SC3}^1) \Rightarrow^4 (\mathsf{J} \rightarrow \mathsf{SC3\text{-}FLAT})$ and we find that p'_c is behavior preserving. Hence, we can guarantee again behavior preservation because we have found a safe split via p'_c for the refactoring transformation.

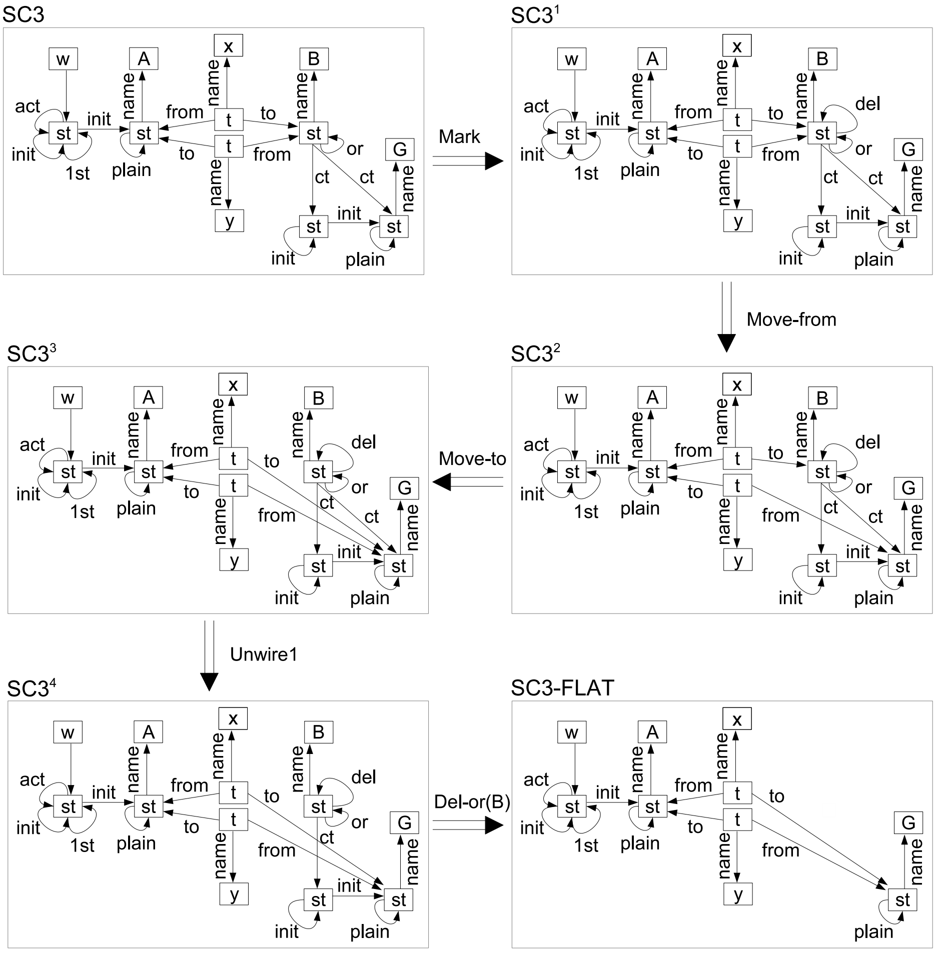


Figure 5.10: Sequence of refactoring transformations (interfaces J are omitted).

Since p'_c is built from rules of Layer 1 and Layer 2 we do not store it for later use.

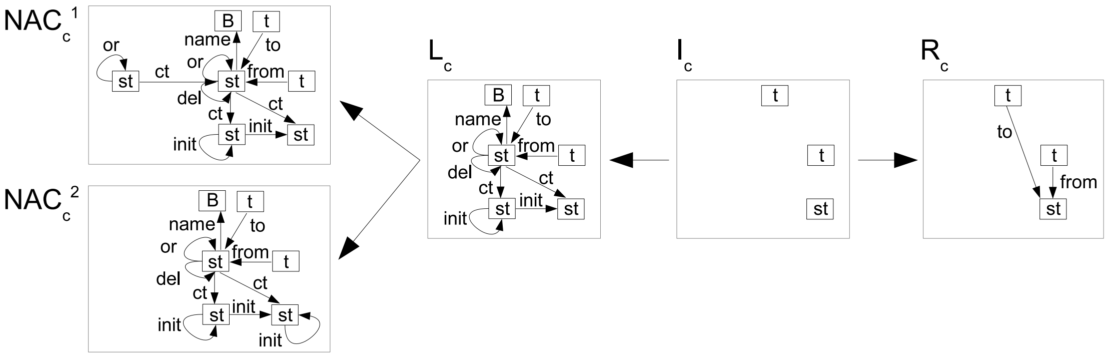


Figure 5.11: Induced concurrent production p'_c.

Note that the left and right sides of p'_c are much smaller than the statecharts in Figure 5.9 which leads to a more efficient behavior preservation checking.

5.6 Conclusions and Future Work

We have shown how the borrowed context technique can be used to reason about behavior-preservation of refactoring rules and refactoring transformations. In this way we shift the perspective from checking specific models to the investigation of the properties of the refactoring rules.

The formal techniques in related work [vKCKB05, PC07, NK06, GSMD03] address behavior preservation in model refactoring, but are in general tailored to a specific metamodel and limited to checking the behavior of a fixed number of models. Therefore, the transfer to different metamodels is, in general, quite difficult.

Hence, with this chapter we propose to use the borrowed context technique in order to consider any metamodel whose operational semantics can be given by graph productions. Furthermore, the bisimulation checking algorithm of Chapter 4 for borrowed contexts provides the means for checking models for behavior preservation. This can be done not only for a specific model and its refactored version, but also for the left-hand and right-hand sides of refactoring rules. Once we have shown that a given rule is behavior-preserving, i.e., its left- and right-hand sides are equivalent, we know that its application will always preserve the behavior, due to the congruence result. When rules are not behavior-preserving, they still can be combined into behavior-preserving concurrent productions. We believe that such a method will help the user to gain a better understanding of the refactoring rules since he or she can be told exactly which rules may modify the behavior during a transformation. An advantage of our technique over the one in [BHE08] is that we work directly with graph transformations and do not need any auxiliary encoding. Furthermore, with our technique we can guarantee that a model and its refactored version have exactly the same observable behavior, while in [BHE08] the refactored model “contains” the original model but may add extra behavior.

This work opens up several possible directions for future investigations. First, in some refactorings when non-behavior-preserving rules are applied, the search strategies for safe splits can become very complex. Here we defined only a simple search strategy, but it should be possible to come up with more elaborate ones.

Second, although we are working with refactoring rules with negative application conditions, these NACs do not play a prominent role in our automatic verification techniques, but of course they are a key to limiting the number of concurrent productions which can be built. The borrowed context framework and the congruence result already handle rules with NACs [RKE08a]. However, this applies only to negative application conditions in the operational semantics. It is, nevertheless, also important to have similar results for refactoring rules with NACs, which would lead to a “restricted” congruence result, where bisimilarity would only be preserved by certain contexts (see also the discussion in Remarks 5.5.4 and 5.5.5). Since model refactorings often use graphs with attributes it would be also useful to investigate whether the congruence results in [EK04, RKE08a] also hold for adhesive HLR categories (the category of attributed graphs is an instance thereof).

Chapter 6

Towards a Tool Support

6.1 Motivation

Experience shows that label derivation and bisimulation proofs demand a great amount of time even for small examples and when done by hand they are particularly susceptible to errors. This situation becomes even worse when dealing with behavior preservation on the level of refactoring rules, as shown in Chapter 5, where a refactoring rule $L \leftarrow I \rightarrow R; \{L \rightarrow NAC_y\}_{y \in Y}$ seen as graphs with interface, namely $I \rightarrow L$ and $I \rightarrow R$, may have an interface I of considerable size exposing several components of L and R to the environment which in turn makes the derivation of labels a nightmare when done manually. This is exactly the case for the flattening of statecharts in Example 5.5.3.

Metamodels whose operational semantics rules require negative application conditions may quickly increase the amount of work required to derive labels and match them via the bisimulation game. Transition labels with negative borrowed contexts are more complex and, additionally, we have to consider arbitrary labels because the technique based on (in)dependent labels does not scale up smoothly to this setting.

Originally, our initial idea to overcome these problems was to implement a prototype to carry out all the hurdle involving bisimulation checking. We began developing a prototype tool in Objective Caml (OCaml) [OCa], which is a functional language very appropriate for rapid prototyping. Due to the nature of graphs and the categorical operations required by the borrowed context technique we can easily implement them with a list-based functional programming language such as OCaml. We chose OCaml as a prototyping language because it is based on Lisp [WH89], and so widely known and used, which results in a variety of libraries and source code available. Furthermore, Ocaml is mainly interpreted for rapid prototyping purposes, but also possesses a code generator to C. These features allow a prototype to be developed in interpreted mode and afterwards compiled from the generated C code, giving rise to a faster program. Our prototype in OCaml uses directed labeled graphs and when we want to check two graphs for bisimilarity, we specify a set of graph productions and also the graphs with interface to be checked. We have implemented graphs with interfaces, graph productions, and all procedures required by the derivation of transition labels. OCaml is mainly textual, but for the sake of visualization, our graphs, rules and derived labels can be visualized with the package Graphviz [Gra].

By the time we extended the borrowed context framework to rules with NACs (Chapter 3) and then applied bisimulations to reason about behavior preservation in model refactoring we realized that our prototype would not be able to evolve to a fully functional tool. Ocaml's main drawback is the lack of graphical user interface (GUI) routines to implement features needed by a graphical editor to draw graphs and graph transformation rules. An alternative strategy would be to do the drawing in a existing tool with GUI resources such as AGG [AGG] and the bisimulation checking with our prototype. However, in practice switching among several small tools (not well integrated) can be very annoying and time consuming in the long run. Usually users tend to stick with solutions disposed in a smoothly integrated environment. For this reason our OCaml prototype is limited to the implementation of structures and operations required by label derivation from graphs and graph rules (without NACs).

The bisimulation checking procedure, the construction of concurrent productions with NACs and an elaborate GUI to support these tasks are part of our future work. We plan to implement the bisimulation checking algorithm and the behavior analysis techniques of Chapter 5 in a programming language such as Java and deploy them as an engine to be used by other tools, such as AGG for example. One of the advantages of equipping AGG with our engine is that the resulting tool would have a graphical editor to define graphs and graph rules, the required machinery to perform graph transformations (and also model refactorings) and the means to reason about behavior preservation of graph transformations and model refactorings.

The following sections complement Chapter 4. Here we describe additional ingredients needed to implement the bisimulation checking algorithm for the borrowed context technique to graph rewriting.

6.2 Graph, Graph Morphism and Match

Labeled graphs (as in Definition 2.3.1) can be easily encoded in an either functional or object-oriented programming language. First we have to declare Ω_V and Ω_E as sets containing labels for nodes and edges, respectively. A labeled graph can be implemented in a functional language as a tuple $G = (V, E, l_v, l_e)$, where:

- V is a set of node names, e.g., $V = \{“n1''; “n2''\}$;

- E is a set of edges. An edge is a tuple $(edge\_name, source, target)$ indicating the name of the edge, its source and target nodes, e.g., $(“e1'', “n1'', “n2'')$;

- l_v is a labeling function for nodes, defined as a set of pairs $(node\_name, type)$, where *type* is a label contained in Ω_V, e.g., $l_v = \{(“n1'', “A''); (“n2'', “B'')\}$;

- l_E is a labeling function for edges, defined as a set of pairs $(edge\_name, type)$, where *type* is a label contained in Ω_E, e.g., $l_E = \{(“e1'', “X'')\}$.

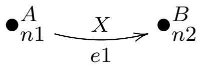


Figure 6.1: Example of a labeled graph.

A graph morphism $f\colon G \to H$ (as in Definition 2.3.1) consists of a function $f_V\colon G_v \to H_v$ for node mapping and another $f_e\colon G_E \to H_E$ for edge mapping. These functions can be encoded in a functional language as sets of tuples, where the first element of each tuple is a node (edge) name from G_V (G_E) and the second is a node (edge) from H_V (H_E).

Given two graphs G and H it is useful to know whether there exist graph morphisms of the form $G \to H$ (i.e., H preserves the structure of G). This is called graph matching and algorithms to find matches play a key role in any graph-based tool. The most straightforward algorithm to finding graph matches consists in creating for each node (edge) in G a list of nodes (edges) from H, where each element of these lists is a candidate for a mapping. The idea is then to generate all possible combinations of candidates and pick only the ones which preserve the structure of G in H, i.e., the ones that are indeed graph morphisms. This simple algorithm is easy to implement, but it has a major drawback in terms of performance due to its exponential worst-case complexity. Any serious graph-based tool implementation can not afford such a bottleneck.

More efficient algorithms take several optimizations into account in order to speed up match finding. One simple optimization is to populate the list of candidate nodes (edges) with nodes (edges) of H which have the same label as the node (edge) in G. A further optimization is to avoid generating all possible combinations of candidates since usually most of them do not lead to proper graph morphisms. A sensible algorithm may decrease the search space by generating the combinations of candidates on the fly with some kind of backtracking search strategy. Constraint satisfaction techniques also bring benefits when applied to graph matching search. Each node and each edge in G is assigned to constraints such as its type, its number of incoming and outgoing edges and functions to ensure source and target compatibility. In [Rud98] such techniques are described together with a backtracking search strategy. This algorithm has a good overall performance, as we observed by implementing it in our OCaml prototype. Another interesting optimization strategy can be found in [Bat06], where heuristics come into play to define search plans for matching strategies.

6.3 Categorical Constructions for Graphs

The borrowed context framework and also the behavioral analysis techniques presented in this thesis rely on basic categorical constructions, such as pushout, pushout complement, pullback and initial pushout, which can be mechanized. Their mathematical definitions are given in Appendix A.

In our OCaml prototype we have implemented the following constructions:

- **Pushout**: see Fact A.8 for the construction. Remark 2.18 in [EEPT06] presents an algorithm;
- **Pullback**: see Fact A.15;
- **Pushout Complement**: see Fact 2.3.9 for the construction. In general the pushout complement may not exist, but by implementing the gluing condition (Definition 2.3.7) we can check its existence;
- **Initial Pushout**: see Fact A.21.

The borrowed context extension to rules with negative application conditions in Chapter 3 requires an additional construction to build a commutative square with jointly surjective (epi) morphisms. Given a span $B \xleftarrow{f} A \xrightarrow{g} C$ of monomorphisms we want to build all possible cospans $B \xrightarrow{g'} D \xleftarrow{f'} C$ of monomorphisms such that $g' \circ f = f' \circ g$ and the morphisms f' and g' are jointly surjective (as in Definition 3.3.3). An algorithm to do so should basically build all possible overlaps D of B and C while preserving the elements of B and C that are identified in A.

Fact 6.1 (**Squares with Jointly Surjective Morphisms in Sets**). *Given a span $B \xleftarrow{f} A \xrightarrow{g} C$ of injective morphisms in* **Sets** *the cospans of injective morphisms $B \xrightarrow{g'} D \xleftarrow{f'} C$ with $g' \circ f = f' \circ g$ and f', g' jointly surjective are constructed as follows. First we build a set $B^{sub} = \{A^+ \in \mathcal{P}(B) \mid \exists\, a \colon A \to A^+ \ \wedge\ \exists f^+ \colon A^+ \to B$ such that (1) commutes$\}$, where $\mathcal{P}$ is a powerset function.*

$$\begin{tikzcd}
B \arrow[rr, "g'"] & & D \\
& A^+ \arrow[ul, "f^+"'] \arrow[dr, "g^+"] \arrow[l, phantom, "(1)"] \arrow[d, phantom, "(2)"] \arrow[ur, phantom, "PO"] & \\
A \arrow[uu, "f"] \arrow[ur, "a"] \arrow[rr, "g"'] & & C \arrow[uu, "f'"']
\end{tikzcd}$$

Then we do:

```
$\forall\ A^+ \in B^{sub}$ :
    $\forall\ g^+\colon A^+ \to C$ injective with (2) commutative :
        build pushout $B \overset{g'}{\to} D \overset{f'}{\leftarrow} C$ of $B \overset{f^+}{\leftarrow} A^+ \overset{g^+}{\to} C$.
```

Proof. Every graph A^+ in B^{sub} is a subgraph of B including A. Since f is injective and (1) commutes it holds that a is injective as well. The morphism f^+ is injective because it is an inclusion. Since g^+ is injective and (2) commutes it holds that A^+ is a subgraph of C containing A. These steps generate all spans $B \overset{f^+}{\leftarrow} A^+ \overset{g^+}{\to} C$ of injective morphisms between B and C, where each A^+ contains at least A.

By taking the pushout $B \overset{g'}{\to} D \overset{f'}{\leftarrow} C$ of f^+ and g^+ we can infer that g' and f' are injective due to the injectivity of g^+ and f^+, respectively. The pushout square implies that g' and f' are jointly surjective. It remains to show the required commutativity, namely $g' \circ f = f' \circ g$. Note that $g' \circ f = g' \circ f^+ \circ a = f' \circ g^+ \circ a = f' \circ g$.

□

Fact 6.1 can also be used in **Graphs** with a power function $\mathcal{P}$ producing subgraphs of B. Moreover, Fact 6.1 is employed to build the diagram below in Definition 3.3.4 (BC rewriting with NACs). Since the graph G^+ tends to be larger than NAC_y it is more efficient to build the cospans $NAC_y \to M_z \leftarrow G^+$ by taking the subgraphs of NAC_y (via $\mathcal{P}$) instead of G^+.

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "{j.surj. \atop =}"] & G^+ \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[l]
\end{tikzcd}$$

6.4 Isomorphism Checking and Graph Certificates

The bisimulation checking procedure for borrowed contexts in Section 4.3.3 heavily depends on isomorphism checks of graphs. Every operation $\in$ and $\notin$ to inspect relations and other data structures for a specific pair $(J \to G, J \to G')$ demands isomorphism checks.

Two graphs G and H are equal up to isomorphism, i.e., they are structurally identical, whenever for every morphism $f\colon G \to H$ there exists an inverse $f^{-1}\colon H \to G$ such that $f^{-1} \circ f = id_G$ and $f \circ f^{-1} = id_H$ (see Definition A.4 in Appendix A). Hence, a standard procedure to check G and H for isomorphism is to find such morphisms f and f^{-1}. Once we have a graph matching algorithm implemented one can easily check graphs for isomorphism. However, even a match finding algorithm with sophisticated techniques to improve its overall performance may very quickly become a bottleneck in a tool which frequently performs isomorphism checks.

A useful optimization consists in equipping graphs with invariants (also called certificates). Graph certificates are "signatures" that identify certain properties in graphs. Straightforward certificates take into account, for example, the number of nodes and edges with their respective labels. A more complex certificate would also store additional information such as the number of incoming and outgoing edges of every node. Graph certificates are intended to be computed very fast and, in addition, the comparison of graph certificates should also be done in no time. The idea is to define a function $c\colon G \to \mathit{Cert}$, where G is a graph and Cert is a certificate computed for G. Two isomorphic graphs end up having the same certificate, whereas the converse does not hold in general. Even though two graphs have the same certificate we still have to find an isomorphism between them to be on the safe side. Therefore, a certificate with high distinguishing power is very desirable. In practice, there are many situations in which it is useful to store graphs together with their respective certificates, and furthermore, each rewriting of a graph should force an update of its certificate. Just to give a flavor of how much impact graph certificates may have in the bisimulation checking algorithm consider the following situation: finding a specific pair of graphs $(J \to G, J \to G')$ inside a huge relation R. This is actually what the up-to isomorphism technique (Definition 3.4.6) and the operations with $\in$ and $\notin$ do.

In our Ocaml prototype we implemented a powerful graph certificate defined by Rensink in [Ren06]. These certificates are created by a function $c\colon G \to \mathbb{N}$ which takes a graph G as input and produces a natural number as certificate. The algorithm to calculate these certificates assigns for each node and edge a natural number, which is a special encoding of their corresponding labels to natural numbers, and iterate as follows: edges not only change their values based on the values of their adjacent nodes, but also modify the values of the nodes. For a finite graph after a finite number of iterations the values one nodes and edges become "stable" with respect to a halt condition. At the end the natural values of all nodes and edges are summed up to obtain a final certificate. Besides this sum the certificate is also composed of a list of natural values used by the edges and another list of the natural values used by the nodes. Thus, whenever two graphs turn out to have the same sum we are also able to compare their respective values in the lists of nodes and edges to ensure that both certificates are really the same. Only in this case we try to find an isomorphism between the two graphs.

6.5 Bisimulation Checking Algorithm

The bisimulation checking algorithm has been described in Section 4.3.3 with pseudo code which makes its presentation compact, but also leaves certain room for more details about some operations which have been only textually described. Here we will focus on clarifying these operations.

Given two graphs with interface we show an algorithm to construct their successor states in the state space product as in Definition 4.3.10. Based on this algorithm we will also be able to check whether these two graphs fails to mimic each other (Definition 4.3.10). These two operations allow us to give more details on how to insert a new pair of graphs with interface into *Table*, where the state space product under investigated is stored.

Table			
(P, Q)	*successors*	m	*fails*
$(1, 4)$	(2,5) (2,6) (3,5) (3,6) ● ● ○ ○	2 3 5 6 ☒□ ☒ □	*false*
$(3, 5)$			*true*
$(3, 6)$	(3,6) ○	3 6 □□	*false*

Figure 6.2: Pairs still under analysis in *Table*.

Figure 6.2 depicts an example of *Table*. We use P and Q as shortcuts for graphs with interface of the form $J \to G$. *Table* has four fields: (P, Q) indicating a pair of graphs with interface; a set called *successors*, where each successor is a tuple of the form $(P', Q', \mathit{boolean})$ whose boolean value indicates whether this successor has already been investigated; a set called m with tuples, where the first element is a successor and the second is a boolean value; *fails* is a boolean value indicating whether P and Q fail to mimic each other.

The procedure **find_successors** is in charge of expanding the state space product of a pair (P, Q) of graphs with interface. We need some auxiliary procedures. The procedure **build_transitions**(P) receives a graph with interface P as argument. First it uses the set $\mathcal{P}$ of rules to calculate partial matches leading to transition labels as described by the algorithm in Section 4.3.1. From these partial matches the procedure builds the corresponding borrowed context steps (Definition 4.2.5) and finally returns a set containing transitions of the form $P \xrightarrow{\mu}_{(d)} P'$ $(\mu = J \rightarrow F \leftarrow K; \{F \rightarrow N_z\}_{z \in Z})$, where the label μ is dependent only when its underlying rule has no NAC.

The procedure **derivable_label**(P, μ) receives a graph with interface P and a transition label μ in order to check whether μ is derivable from P and the set $\mathcal{P}$ of rules (as in Definition 4.3.6). This procedure returns either $P \xrightarrow{\mu} P'$ if μ is derivable from P or $\emptyset$, otherwise. We define three functions to access the elements of a transition $t = P \xrightarrow{\mu} P'$, namely $\mathbf{1st}(t) = P$, $\mathbf{2nd}(t) = P'$ and $\mathbf{label}(t) = \mu$.

```
find_successors(P, Q) :=
  succs := ∅;   m := ∅;
  T_P := build_transitions(P);
  T_Q := build_transitions(Q);
  for each t_P ∈ T_P do
     t_Q := derivable_label(Q, label(t_P));
     if t_Q ≠ ∅
        then insert (2nd(t_P), 2nd(t_Q), false) into succs;
               insert (2nd(t_P), false) into m;
               insert (2nd(t_Q), false) into m;
        else return ((P, Q), ∅, ∅, true);
  end for
  for each t_Q ∈ T_Q do
     t_P := derivable_label(P, label(t_Q));
     if t_P ≠ ∅
        then insert (2nd(t_P), 2nd(t_Q), false) into succs;
               insert (2nd(t_P), false) into m;
               insert (2nd(t_Q), false) into m;
        else return ((P, Q), ∅, ∅, true);
  end for
  return ((P, Q), succs, m, false);
```

In the procedure **bisimulation_check** we replace:

```
insert (P, Q) into S and Table; status := true;
```

by this code

```
insert (P, Q) into S;
insert find_successors(P, Q) into Table;
status := true;
```

and

```
{(P_0, Q_0) -μ-> successor(P_0, Q_0)}
insert successors of (P_0, Q_0) into S;
for each successor(P_0, Q_0) do
   if successor(P_0, Q_0) ∉ Table
            ∧ successor(P_0, Q_0) ∉ W ∪ R ∪ V
      then insert it into Table;
end for
```

by this code

```
succ_details :=find_successors(P_0, Q_0);
succs :=get_successors(succ_details);
for each succ ∈ succs do
    insert succ into S;
    if succ ∉ Table ∧ succ ∉ W ∪ R ∪ V
      then insert find_successors(succ) into Table;
end for
```

In the procedure **propagate** the following line

```
for each (P_f, Q_f) ∈ Table
                    with (P_f, Q_f) -μ-> (P, Q) do
```

should be replaced by

```
for each (P_f, Q_f) ∈ Table such that
                    (P, Q) ∈ Table.successors do
```

The procedure **find_successors** calculates the successors of P and Q whenever they can mimic each other. This procedure returns the information about (P, Q) to be stored into *Table* (see last line with *return*). First we generate the set T_P (T_Q) containing transitions of the form $P \xrightarrow{\mu}_{(d)} P'$ $(Q \xrightarrow{\mu}_{(d)} Q')$. Then we check whether each transition label from one graph is derivable from the other. If a label is derivable from the other graph they properly match and we insert the pair of successors into *succs* and each of its components into m. Whenever at least one label is not derivable from the other graph then P and Q fail to mimic each other and we return $((P, Q), \emptyset, \emptyset, \mathit{true})$.

The procedure **find_successors** can now be used in **bisimulation_check** of Section 4.3.3. We need to make two replacements in **bisimulation_check** as described above. First we replace how the successors of (P, Q) are inserted into *Table*. Note that in practice each field of *Table* should explicitly receive a field of the tuple returned by **find_successors**. The second replacement specifies how the successors of (P_0, Q_0) are constructed. The procedure **get_successors** returns the set of successors from the tuple *succ_details*. Then we insert each each successor into **S**, and in addition, if its information is not available in *Table* and it has not been already analyzed we insert the information about its successors into *Table*. The last replacement is within the procedure **propagate**, where we want to fetch the parents of (P, Q) in the state space product (stored in *Table*). This is done by looking for (P, Q) in every set *Table.successors* of each entry of *Table* and returning the associate pairs in *Table*.(P, Q).

6.6 Up-to Techniques

We show how the up-to techniques of Section 3.4 can be implemented. The general principle is as follows: given a relation $\mathcal{R} = \{(J \to G, J \to G'), ...\}$ and an up-to technique specified as a function $\mathcal{F}$ from relations to relations, we want to decide whether a pair of graphs with interface $(K \to H, K \to H')$ belongs to $\mathcal{F}(\mathcal{R})$.

6.6.1 Up-to Isomorphism

Since graphs are defined up to isomorphism the bisimulation checking algorithm of Section 4.3.3 requires at least the up-to isomorphism technique (Definition 3.4.6), which is given by:

$$\begin{aligned}\mathcal{F}^{iso}(\mathcal{R}) = \{(K \to H, K \to H') \mid \exists(J \to G, J \to G') \in \mathcal{R} \text{ and there exist} \\ \text{isomorphisms } K \xrightarrow{\sim} J, H \xrightarrow{\sim} G, H' \xrightarrow{\sim} G' \text{ such that } (1), (2) \text{ commute}\}\end{aligned}$$

$$\begin{tikzcd} K \arrow[r] \arrow[d, "\wr"'] \arrow[dr, phantom, "(1)"] & H \arrow[d, "\wr"'] \\ J \arrow[r] & G \end{tikzcd} \qquad \begin{tikzcd} K \arrow[r] \arrow[d, "\wr"'] \arrow[dr, phantom, "(2)"] & H' \arrow[d, "\wr"'] \\ J \arrow[r] & G' \end{tikzcd}$$

Let each element in $\mathcal{R}$ be a tuple of the form $(J \to G, J \to G', C_G, C_{G'})$, where C_G and $C_{G'}$ are graph certificates for $J \to G$ and $J \to G'$, respectively. The procedure **same_certificate** checks whether two graph certificates are equal and **isomorphic** checks if two graphs with interface are isomorphic. Thus, given two graphs and their respective certificates the procedure **is_in_upto_iso** tries to find an isomorphic pair in the relation $\mathcal{R}$. First we compare graph certificates, which is fast, and only when the certificates are equal we proceed by checking the graphs for isomorphism. Observe that the symmetry of $\mathcal{R}$ is implemented by code in order to keep this relation as small as possible.

```
is_in_upto_iso($K \to H, K \to H', C_H, C_{H'}, \mathcal{R}$) :=
  for each ($J \to G, J \to G', C_G, C_{G'}$) $\in \mathcal{R}$ do
    if same_certificate($C_H, C_G$) $\wedge$ same_certificate($C_{H'}, C_{G'}$)
       then if isomorphic($K \to H, J \to G$) $\wedge$ isomorphic($K \to H', J \to G'$)
            then return $true$;
       else if same_certificate($C_H, C_{G'}$) $\wedge$ same_certificate($C_{H'}, C_G$)
           then if isomorphic($K \to H, J \to G'$) $\wedge$ isomorphic($K \to H', J \to G$)
           then return $true$;
  end for
return $false$;
```

In the procedure **is_in_upto_iso** above the use of graph certificates have a great impact in the amount of time required to decide the membership of a pair of graphs with interface in $\mathcal{F}^{iso}(\mathcal{R})$, because it is able to quickly skip several pairs in $\mathcal{R}$ whose certificates do not match with the certificates of $(K \to H, K \to H')$.

6.6.2 Up-to Context

A more powerful technique is the up-to context (Definition 3.4.7). Two graphs with interface $(K \to H, K \to H')$ are bisimilar up to context if after removal of identical contexts the resulting pair of graphs can be found in the relation $\mathcal{R}$. This technique is given by:

$$\mathcal{F}^C(\mathcal{R}) = \{(K \to H, K \to H') \mid \exists (J \to G, J \to G') \in \mathcal{R} \text{ and there exists a context } J \to E \leftarrow K \text{ inducing the diagrams below}\}$$

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] \arrow[dr, phantom, "="] & K \arrow[l] \arrow[dl] \\
G \arrow[r] & H &
\end{tikzcd}
\qquad
\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] \arrow[dr, phantom, "="] & K \arrow[l] \arrow[dl] \\
G' \arrow[r] & H' &
\end{tikzcd}$$

We need three auxiliary procedures in order to implement **is_in_upto_context** below. Given two graphs G and H as input the procedure **find_monomorphisms** returns a set containing injective morphisms $G \to H$, if they exist. The procedure **build_context** receives morphisms of the form $J \to G \to H \leftarrow K$ and returns a context $J \to E \leftarrow K$ whenever: the pushout complement $J \to E \to H$ exists and there exists an injective morphism $K \to E$ such that the corresponding triangle above commutes. If such a context does not exist **build_context** returns $\emptyset$. The procedure

check_context determines whether $K \rightarrow H$ can be indeed obtained by inserting $J \rightarrow G$ into the context $J \rightarrow E \leftarrow K$.

```
is_in_upto_context($K \rightarrow H, K \rightarrow H', \mathcal{R}$) :=
  for each ($J \rightarrow G, J \rightarrow G'$) $\in \mathcal{R}$ do
      $mH$ :=find_monomorphisms($G, H$);
      if $mH \neq \emptyset$
          then
              for each $mh \in mH$ do
                  $context$ :=build_context($J \rightarrow G, K \rightarrow H, mh$);
                  if $context \neq \emptyset$
                      then
                          if check_context($J \rightarrow G', K \rightarrow H', context$)
                              then return $true$;
              end for

      $mH'$ :=find_monomorphisms($G, H'$);
      if $mH' \neq \emptyset$
          then
              for each $mh' \in mH'$ do
                  $context$ :=build_context($J \rightarrow G, K \rightarrow H', mh'$);
                  if $context \neq \emptyset$
                      then
                          if check_context($J \rightarrow G', K \rightarrow H, context$)
                              then return $true$;
              end for
  end for
  return $false$;
```

Note that **is_in_upto_context** subsumes the procedure **is_in_upto_iso** of Section 6.6.1. However, this fact does not lessen the usefulness of **is_in_upto_iso**. In practice, it can be computed much faster than **is_in_upto_context** due to the use of graph certificates and its simpler machinery.

Chapter 7

Conclusion

Here we summarize the main achievements of this thesis, and then outline open problems and directions for future work.

7.1 Summary and Main Results

In this thesis we have focused on a framework to reason about behavioral aspects of graphs and graph transformations. The advent of the DPO extension to borrowed contexts provided the means to draw a clear line between internal and observable behavior of systems given by graph transformations. Borrowed contexts allow systems to be treated as *black boxes*, where users or other systems (formally defined as an environment) are only granted to interact with the visible part of a system, whereas the internal computations are hidden, and therefore not observable.

The work presented in this thesis has extended the borrowed context framework [EK04, EK06] of Ehrig and König in many directions. The results of this thesis are in essence a consequence of bouncing between theory and practice. Initially, the main drive of our research was more theory-oriented, but later on we found out a very interesting application area (model refactoring) which could profit a lot by exploiting the distinction between observable and internal (hidden) behavior and the main result of borrowed contexts (bisimilarity is a congruence). By this time, our interest was fundamentally to extend the borrowed context framework in ways with direct impact in practice.

Compared to the other approaches based on relative pushouts [LM00, SS03a] to derivation of labeled transition systems the borrowed context technique has several advantages, varying from its intuitive graph-based notation to its easier underlying theory and simpler machinery to label derivation. Furthermore, by using graph transformations one can analyze a wide range of problems: protocol verification, behavioral equivalences of process calculi, model transformation and model refactoring, just to cite a few.

Below we summarize the main results achieved in this thesis.

Bisimulation Congruences in the Presence of NACs: we investigated negative application conditions in transformations rules and their consequences to the bisimilarity result. We discussed which problems arise due to the introduction of NACs and how they can be overcome in order to guarantee that the derived bisimilarities are congruences. The extension, which is carried out for adhesive categories [LS04], required an enrichment of the transition labels which now do not only indicate the context that is provided by the observer, but also constrain further additional contexts that may (not) satisfy the negative application condition. That is, we do not only specify what must be borrowed, but also what should not be borrowed. We proved that the main result of [EK06] (bisimilarity is a congruence) still holds for our extension. Furthermore, as a straightforward consequence of a technique based on initial pushouts proposed in [BGK06a] we defined the notion of gluing condition for borrowed context rewriting. We have illustrated the theory via two examples: a simple example based on processing tasks on servers and a more elaborate one in terms of blade server systems.

Up-to Techniques: we have extended the up-to context technique of [EK06] to the setting with NACs. Moreover, we defined two additional up-to techniques which help reduce the size of the relation needed to define a bisimulation. They are: up-to isomorphism and up-to bisimilarity. Because graphs in the DPO approach are defined up to isomorphism the up-to isomorphism technique can be considered the minimal requirement to enable bisimulation checks in the borrowed context framework. Even though this technique turns out to be subsumed by the more powerful up-to context it is still very useful in practice since it can be computed much faster with help of graph certificates.

Bisimulation Checking Algorithms: we have defined the essential algorithms required by the development of a tool support to check graphs for bisimilarity. The main algorithms are: partial match finding, label matching, up-to isomorphism, up-to context and the extension of Hirschkoff's on-the-fly bisimulation checking algorithm [Hir01] to borrowed contexts. We have also outlined the role of graph certificates [Ren06] and how they can be used to speed up some bottlenecks caused by isomorphism checks during bisimulation proofs. We have also developed a prototype in OCaml to derive transitions labels from graph with interfaces and graph rules.

Behavioral Analysis Techniques for Model Refactoring: we have shown that the borrowed context technique is a useful and suitable instrument to analyze the behavior of systems. One application area that has profited from the results of this thesis is model refactoring. New techniques to check model refactorings for behavior preservation, which is always a crucial aspect of every refactoring transformation, have been defined using the borrowed context machinery. The simplest technique checks instances of a metamodel for bisimilarity w.r.t. to a set of productions defining the operational semantics of the metamodel. Bisimilarity implies behavior preservation. A more elaborate technique exploits the fact that observational equivalence is a congruence and hence we show how to check refactoring rules for behavior preservation. When rules are behavior-preserving, their application will never change behavior, i.e., every model and its refactored version will have the same behavior. For the cases where non behavior-preserving refactoring rules are applied we defined a procedure to combine refactoring rules to behavior-preserving concurrent productions in order to ensure behavior preservation. The advantage of this second technique lies in its capability to reduce proof obligations, since we do not have to show behavior preservation between the start and end graph of the refactoring sequence (which may be huge), but we only have to investigate local updates of the model. Both techniques were applied to examples of minimization of deterministic finite automata and flattening of hierarchical statecharts. We believe that the second technique will help the user to gain a better understanding of the refactoring rules since he or she can be told exactly which rules may modify the behavior during a transformation. One of the advantages of our techniques over the ones in related work [vKCKB05, PC07, NK06, GSMD03] is that they are not tied to specific metamodels, but they can handle any metamodel whose operational semantics can be given by graph productions. For the behavioral analysis of refactoring productions an advantage of our techniques over the one in [BHE08] is that we work directly with graph transformations and do not need any auxiliary encoding. Furthermore, with our technique we can guarantee that a model and its refactored version have exactly the same observable behavior, while in [BHE08] the refactored model “contains” the original model but may add extra behavior.

Application to Examples: in this thesis the borrowed context technique has shown its applicability in a variety of examples: blade server systems, minimization of deterministic finite automata by merging of equivalent states, deletion of unreachable states in finite automata and flattening of hierarchical statecharts into plain state machines.

7.2 Open Problems and Future Work

The work presented in this thesis opens up several possible directions for future investigations.

Bisimulation Congruences for Rules with NACs

In our extension of the borrowed context to rules with negative application conditions we have obtained a finer congruence than the usual one. This is due to the construction of negative borrowed contexts via the jointly epi square which ends up in some cases generating extra information. One possible solution to be investigated consists in reducing the number of these possible contexts (for instance by forbidding contexts that contain certain patterns or subobjects) which would lead to coarser congruences, i.e., more objects would be equivalent. Studying such congruences is an interesting future work to be pursued.

Furthermore, a natural question to ask is whether there are other extensions to the DPO approach that, when carried over to the borrowed context framework, would require the modification of transition labels. One such candidate are generalized application conditions, so-called graph conditions [Ren04], which are equivalent to first-order logic and of which NACs are a special case. Such conditions would lead to fairly complex labels.

Bisimulation checks may be in some cases very inefficient. In order to improve the overall performance of the bisimulation checking algorithm we need further speed-up techniques such as additional up-to techniques and methods for downsizing the transition system, such as the elimination of independent labels. We discussed in Section 3.4 that the proof technique to eliminate independent labels [EK04, EK06] (or non-engaged labels as they are called in [Mil06]) does not carry over straightforwardly to the setting with NACs, but it can still be useful. This needs to be studied further.

Some open questions remain for the moment. In the categorical setting it would be good to know whether pullbacks always preserve epis in adhesive categories. This question is currently open, as far as we know. Moreover, it is unclear where the congruence is located in the lattice of congruences that respect rewriting steps with NACs. As for IPO bisimilarity it is probably not the coarsest such congruence, since saturated bisimilarity is in general coarser [BKM06]. So it would be desirable to characterize such a congruence in terms of barbs [RSS07].

It is not clear to us at the moment how NACs could be integrated directly into reactive systems and how the corresponding notion of IPO would look like. In our opinion this would lead to fairly complex notions, for instance one would have to establish a concept similar to that of jointly epi arrows.

Another topic that deserves further analysis is the relation between the bisimilarities we obtain via the borrowed context technique and the standard bisimilarity of finite automata seen as transition system. Proving that both notions coincide is not so trivial since via borrowed contexts we may derive certain transition labels, e.g. independent labels, which do not have a direct meaning in the automata model.

Finally, it would be also interesting to investigate the theoretical implications of merging a sequence of borrowed context steps into single steps via concurrent productions [EEPT06, Lam07]. This could lead to a similar notion as weak bisimulation in the BC framework, where one graph with interface would be able to perform "internal" processing in order to match other transitions.

Algorithms and Tool Support

In our extension of Hirschkoff's bisimulation checking algorithm to borrowed contexts we have already made some minor efficiency improvements to avoid certain redundant computations, but the algorithm can still be further improved. Hence, a full investigation from the efficiency point of view should be carried out.

The algorithms defined in this thesis are also due to improvements. For example, the partial match finding algorithm could benefit from the technique based on dependent and independent labels. An initial idea is to first calculate for a graph with interface and a production without NACs the partial match which is the threshold for independent labels and then we could test "bigger" partial matches. This strategy may lead to the generation of fewer candidates for partial matches.

During the development of this thesis we have implemented a prototype in OCaml [OCa] to derive transition labels. Our initial plan was to implement the bisimulation checking algorithm in OCaml as well, but we have decided to postpone the implementation to future work since our prototype would not be able to fulfill the expectations of potential users (e.g. from the model refactoring field) in terms of graphical user interface (GUI) and usability. Our plan is to implement the bisimulation checking algorithm in a programming language such as Java and deploy it as an engine which can be later reused by a variety of tools. A second step would consist in equipping AGG [AGG] with our engine for bisimulation checking in order to reason about behavior preservation in model refactoring.

Behavior Preservation in Model Refactoring

In some refactorings whenever non-behavior-preserving rules are applied, the search strategies for safe splits, through which we can again guarantee behavior preservation, can become very complex. In this thesis we defined only a simple search strategy, but it should be possible to come up with more elaborate ones.

Although we have worked with refactoring rules with negative application conditions, these NACs have not played a prominent role in our automatic verification techniques, but of course they are a key to limiting the number of concurrent productions which can be built. In [RKE08a] the borrowed context framework and the congruence result has been extended to handle rules with NACs. However, this applies only to negative application conditions in the operational semantics. It is, nevertheless, also important to have similar results for refactoring rules with NACs, which would lead to a “restricted” congruence result, where bisimilarity would only be preserved by certain contexts. Since model refactorings often use graphs with attributes it would be useful to investigate whether the congruence results in [EK04, RKE08a] also hold for adhesive HLR categories [EEPT06] (the category of attributed graphs is an instance thereof).

Beyond model refactoring an interesting work consists in generalizing our behavior analysis techniques to model transformations between different metamodels. In this case one aspect that should be kept in mind is that transition labels from different metamodels may be different, which would demand translations in order to properly match them.

Appendix A

Categorical Concepts

We briefly recall some categorical concepts that are used throughout this thesis. The text presented here is originally from [EEPT06]. For more comprehensive introductions to the world of category theory we refer the reader to [Pie91, BW95, Mar95].

A category is a mathematical structure that has objects and morphisms, with an associative composition operation on the morphisms and an identity morphism for each object.

Definition A.1 (Category). *A category* $\mathbf{C} = (Ob_C, Mor_C, \circ, id)$ *is defined by*

- *a class* Ob_C *of* objects*,*
- *a set* $Mor_C(A, B)$ *of* morphisms *for each pair of objects* $A, B \in Ob_C$*,*
- *a* composition *operation* $\circ_{(A,B,C)} \colon Mor_C(B, C) \times Mor_C(A, B) \to Mor_C(A, C)$ *for all objects* $A, B, C \in Ob_C$ *and*
- *an* identity *morphism* $id_A \in Mor_C(A, A)$ *for each object* $A \in Ob_C$*,*

such that the following conditions hold:

1. *Associativity: for all objects* $A, B, C, D \in Ob_C$ *and morphisms* $f \colon A \to B$, $g \colon B \to C$ *and* $h \colon C \to D$ *it holds:* $(h \circ g) \circ f = h \circ (g \circ f)$*;*
2. *Identity: for all objects* $A, B \in Ob_C$ *and morphisms* $f \colon A \to B$ *it holds:* $f \circ id_A = f$ *and* $id_B \circ f = f$*.*

Remark A.2. *Instead of* $f \in Mor_C(A, B)$ *we write* $f \colon A \to B$ *and leave out the index for the composition operation, since it is clear which one to use. For such a morphism* f*,* A *is called its domain and* B *its codomain.*

Example A.3
We show two examples of structures that form categories:

1. Category **Sets**, where the object class contains all sets and the morphisms are all functions $f\colon A \to B$. The composition for $f\colon A \to B$ and $g\colon B \to C$ is defined by $(g \circ f)(x) = g(f(x))$ for all $x \in A$, and the identity is the identical mapping $id_A\colon A \to A\colon x \mapsto x$.

2. The class of all graphs (as in Definition 2.3.1) as objects and of all graph morphism (as in Definition 2.3.1) forms the category **Graphs**, with the composition given in Fact 2.3.2, and the identities as the pairwise identities on nodes and edges.

Definition A.4 (**Mono-, Epi- and Isomorphism**). *Given a category* **C**, *a morphism $m\colon B \to C$ is a* monomorphism *if for all morphisms $f, g\colon A \to B \in Mor_C$ it holds: $m \circ f = m \circ g \Rightarrow f = g$.*

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "m"] & C \end{tikzcd}$$

A morphism $e\colon A \to B \in Mor_C$ is an epimorphism *if for all morphisms $f, g\colon B \to C$ it holds: $f \circ e = g \circ e \Rightarrow f = g$.*

$$\begin{tikzcd} A \arrow[r, "e"] & B \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & C \end{tikzcd}$$

A morphism $i\colon A \to B$ is an isomorphism *if there exists a morphism $i^{-1}\colon B \to A$ such that $i \circ i^{-1} = id_B$ and $i^{-1} \circ i = id_A$.*

$$\begin{tikzcd} A \arrow[r, shift right, "i"'] & B \arrow[l, shift right, "i^{-1}"'] \end{tikzcd}$$

Remark A.5. *An isomorphism i is both a monomorphism and an epimorphism. For every isomorphism i the inverse morphism i^{-1} is unique.*

Fact A.6. *In* **Sets** *and* **Graphs** *the monomorphisms (epimorphisms, isomorphisms) are exactly those morphisms that are injective (surjective, bijective).*

Proof. See Fact 2.15 in [EEPT06] (page 27). □

A.1 Pushouts as Gluing Construction

For the application of a graph transformation rule to a graph we need a technique to glue graphs together along a common subgraph. Intuitively the resulting graph consists of the common subgraph glued to all other nodes and edges from both graphs. The concept of pushouts generalizes this gluing construction to categorical terms, i.e., a pushout object emerges from gluing two objects along a common subobject.

Definition A.7 (Pushout). *Given the morphisms $f\colon A \to B$ and $g\colon A \to C$ in a category $\mathbf{C}$, a* pushout *(D, f', g') over f and g is defined by*

- *a pushout object D, and*
- *morphisms $f'\colon C \to D$ and $g'\colon B \to D$ with $f' \circ g = g' \circ f$*

such that the following universal property is fulfilled: for all objects X and morphisms $h\colon B \to X$ and $k\colon C \to X$ with $k \circ g = h \circ f$ there exists a unique morphism $x\colon D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "="] & B \arrow[d, "g'"] \arrow[ddr, bend left, "h"] & \\
C \arrow[r, "f'"'] \arrow[drr, bend right, "k"'] & D \arrow[dr, dotted, "x"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
& & X
\end{tikzcd}$$

We often use the abbreviation "PO" for pushout.

Fact A.8 (PO in Sets and Graphs). *In* **Sets** *the pushout object over $f\colon A \to B$ and $g\colon A \to C$ can be constructed as the quotient $B \mathbin{\dot\cup} C|_{\equiv}$, where $\equiv$ is the smallest equivalence relation with $(f(a), g(a)) \in {\equiv}$ for all $a \in A$. The morphisms f' and g' are defined by $f'(c) = [c]$ for all $c \in C$ and $g'(b) = [b]$ for all $b \in B$.*

Moreover the following properties hold:

1. *If f is injective (surjective) then also f' is also injective (surjective).*
2. *The pair (f', g') is jointly surjective, i.e., for each $x \in D$ there is a preimage $b \in B$ with $g'(b) = x$ or $c \in C$ with $f'(c) = x$.*
3. *If $x \in D$ has preimages $b \in B$ and $c \in C$ with $g'(b) = f'(c) = x$ then there is a preimage $a \in A$ with $f(a) = b$ and $g(a) = c$. If f is injective then a is unique.*

4. If f is injective (and hence f' as well) then D is isomorphic to $D' = C \dot{\cup} B \setminus f(A)$.

In **Graphs** *pushouts can be constructed componentwise for nodes and edges in* **Sets**. *Furthermore, the properties 1-4 hold componentwise.*

Proof. See Fact 2.17 in [EEPT06] (page 29). □

Pushout squares can be decomposed if the first square is a pushout. Furthermore, pushout properties are preserved.

Fact A.9 (Uniqueness, Composition and Decomposition of POs). *Given a category* **C**, *we have:*

a) The pushout object D is unique up to isomorphism.

b) The composition and decomposition of pushouts result again in a pushout, i.e., given the following commutative diagram, then it holds:

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

- *Pushout composition: If* (1) *and* (2) *are pushouts then* (1)+(2) *is a pushout as well.*
- *Pushout decomposition: If* (1) *and* (1) + (2) *are pushouts then* (2) *is also a pushout.*

Proof. See Fact 2.20 in [EEPT06] (page 31). □

Fact A.10 (PO Splitting). *In any category* **C** *given the left diagram below as pushout then it splits into two pushouts as shown in the right diagram.*

$$\begin{tikzcd}
A \arrow[r] \arrow[d] & B \arrow[r] \arrow[d, phantom, "PO"] & E \arrow[d] \\
C \arrow[rr] & & F
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

Proof. This follows easily from pushout decomposition in Fact A.9. □

In a variety of situations we need the reverse construction of a pushout, which is called pushout complement.

Definition A.11 (**Pushout Complement**). *Given morphisms $f\colon A \to B$ and $g'\colon B \to D$, then $A \overset{g}{\to} C \overset{f'}{\to} D$ is the* pushout complement *of f and g', if* (1) *below is a pushout.*

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"] \\
C \arrow[r, "f'"'] & D
\end{tikzcd}$$

Fact A.12. *Pushout complements of monomorphisms (if they exist) are unique up to isomorphism.*

Proof. See item 4. of Theorem 4.26 in [EEPT06] (page 96). □

Fact A.13 (**Pushout Complement Splitting**). *Whenever the square below on the left is a pushout with monos, then it can be split into two pushouts with monos as depicted on the right.*

$$\begin{tikzcd}
A \arrow[rr] \arrow[d] \arrow[drr, phantom, "PO"] & & E \arrow[d] \\
C \arrow[r] & B \arrow[r] & F
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] \\
C \arrow[r] & B \arrow[r] & F
\end{tikzcd}$$

Proof. This follows easily from Fact A.19. □

A.2 Pullbacks

The dual construction of a pushout is called pullback. Pullbacks can be seen as a generalized intersection of objects over a common object in category theory.

Definition A.14 (**Pullback**). *Given the morphisms $f\colon C \to D$ and $g\colon B \to D$ in a category* **C**, *a* pullback (A, f', g') *over f and g is defined by*

- *a pullback object A, and*
- *morphisms $f'\colon A \to B$ and $g'\colon A \to C$ with $g \circ f' = f \circ g'$*

such that the following universal property is fulfilled: for all objects X with morphisms $h\colon X \to B$ and $k\colon X \to C$ with $f \circ k = g \circ h$ there is a unique morphism $x\colon X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$.

$$\begin{tikzcd}
X \arrow[rrd, bend left, "h"] \arrow[rd, dotted, "x"] \arrow[rdd, bend right, "k"'] \arrow[rrd, phantom, "="] \arrow[rdd, phantom, "="'] & & \\
& A \arrow[r, "f'"] \arrow[d, "g'"'] \arrow[rd, phantom, "="] & B \arrow[d, "g"] \\
& C \arrow[r, "f"'] & D
\end{tikzcd}$$

We often use the abbreviation "PB" for pullback.

Fact A.15 (PB in Sets and Graphs). *In* **Sets** *the pullback $C \xleftarrow{g'} A \xrightarrow{f'} B$ over morphisms $f\colon C \to D$ and $g\colon B \to D$ is constructed by $A = \bigcup_{d \in D} f^{-1}(d) \times g^{-1}(d) = \{(c,b) \mid f(c) = g(b)\} \subseteq C \times B$ with morphisms $f'\colon A \to B\colon (c,b) \mapsto b$ and $g'\colon A \to C\colon (c,b) \mapsto c$. Moreover, the following properties hold:*

1. *If f is injective (surjective), then f' is also injective (surjective).*
2. *f' and g' are jointly injective, i.e., for all $a_1, a_2 \in A$, $f'(a_1) = f'(a_2)$ and $g'(a_1) = g'(a_2)$ implies $a_1 = a_2$.*
3. *A commutative square, as in Definition A.14, is a pullback in* **Sets** *if and only if, for all $b \in B$, $c \in C$ with $g(b) = f(c)$, there is a unique $a \in A$ with $f'(a) = b$ and $g'(a) = c$.*

In **Graphs** *pullbacks can be constructed componentwise for nodes and edges in* **Sets***.*

Proof. See Fact 2.23 in [EEPT06] (page 34). □

Remark A.16. *In* **Sets** *and* **Graphs** *we have the interesting property that pushouts, where at least one of the given morphisms is injective, is already a pullback. For further details see Remark 2.25 in [EEPT06] (page 34).*

Fact A.17 (Uniqueness, Composition and Decomposition of PBs). *Given a category* **C***, we have:*

a) The pullback object A is unique up to isomorphism.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[rd, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[rd, phantom, "(2)"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

b) The composition and decomposition of pullbacks result again in a pullback:

- *Pullback composition: if* (1) *and* (2) *are pullbacks then* (1)+(2) *is a pullback as well.*
- *Pullback decomposition: if* (2) *and* $(1) + (2)$ *are pullbacks then* (1) *is also a pullback.*

Proof. See Fact 2.27 in [EEPT06] (page 35). □

Fact A.18 (**PB Splitting**). *In any category* **C** *given the left diagram below as pullback then it splits into two pullbacks as shown in the right diagram.*

$$\begin{tikzcd}
A \arrow[rr] \arrow[d] & \arrow[d, phantom, "PB"] & E \arrow[d] \\
C \arrow[r] & B \arrow[r] & F
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "PB"] & D \arrow[r] \arrow[d] \arrow[dr, phantom, "PB"] & E \arrow[d] \\
C \arrow[r] & B \arrow[r] & F
\end{tikzcd}$$

Proof. This follows easily from pullback decomposition in Fact A.17. □

Fact A.19 (**Special Pushout-Pullback Decomposition**). *Given the diagram below with monos, whenever* $(1) + (2)$ *is a pushout and* (2) *is a pullback then* (1) *and* (2) *are both pushouts.*

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

Proof. See item 2. of Theorem 4.26 in [EEPT06] (page 96). □

A.3 Initial Pushouts

Given a rule $L \leftarrow I \rightarrow R$ and a match $m\colon L \rightarrow G'$ as depicted below the gluing condition of Definition 2.3.7 describes an operational way to check the existence of the context graph C'.

$$\begin{tikzcd}
L \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] & R \\
G' & C' \arrow[l] &
\end{tikzcd}$$

This notion of gluing condition can generalized from **Graphs** to any category **C** possessing initial pushouts. First let us define the categorical concept of initial pushout.

Definition A.20 (Initial Pushout). *Given a morphism $f\colon A \to D$ in a category* $\mathbf{C}$*, a mono $b\colon B \to A$ is called* boundary *over f if there exists a pushout complement $B \to C \xrightarrow{c} D$ such that Diagram* (1) *is a pushout which is* initial *over f. Initiality of Diagram* (1) *over f means that for every other pushout as in Diagram* (2) *with b' mono there exist two unique monos $B \to A'$ and $C \to D'$ such that Diagram* (2) *commutes and* (3) *is a pushout. The object B is called* boundary object *and C the* context *with respect to f.*

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[dr, phantom, "PO"] & A \arrow[d, "f"] \\
C \arrow[r, "c"'] & D
\end{tikzcd} \qquad (1) \qquad\qquad
\begin{tikzcd}
B \arrow[rr, "b"] \arrow[dr, dotted] \arrow[ddd] & & A \arrow[ddd, "f"] \\
& A' \arrow[ur, "b'"'] \arrow[d, "(3)"', "PO"] & \\
& D' \arrow[dr] & \\
C \arrow[ur, dotted] \arrow[rr, "c"'] & & D
\end{tikzcd} \qquad (2)$$

Fact A.21 (Initial Pushouts in Graphs). *The boundary object B of an injective graph morphism $f\colon A \to D$ consists of all nodes $a \in A$ such that $f(a)$ is adjacent to an edge in $D \setminus f(A)$. These nodes are needed to glue A to the context graph $C = D \setminus f(A) \cup f(b(B))$ in order to obtain D as the gluing of A and C via B in the initial pushout.*

In the following the gluing condition is formulated in terms of initial pushouts.

Definition A.22 (Gluing Condition via Initial Pushout). *In a category* $\mathbf{C}$ *with initial pushouts, a match $m\colon L \to G'$ satisfies the gluing condition with respect to a rule $L \xleftarrow{l} I \xrightarrow{r} R$ (l mono) if, for the initial pushout* (1) *over m, there exists a mono $b^*\colon B \to I$ such that $b = l \circ b^*$.*

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[dr, phantom, "(1)"] \arrow[rr, bend left, "b^*"] & L \arrow[d, "m"] & I \arrow[l, "l"'] \arrow[r, "r"] & R \\
C \arrow[r, "c"'] & G' & &
\end{tikzcd}$$

Theorem A.23 (Existence and Uniqueness of Contexts). *Given a category* $\mathbf{C}$ *with initial pushouts a match $m\colon L \to G'$ satisfies the gluing condition (Definition A.22) with respect to a rule $L \xleftarrow{l} I \xrightarrow{r} R$ (l mono) if and only if the context object C' exists, i.e., there exists a pushout complement $I \to C' \to G'$ of l and m. Moreover, whenever the context object C' exists it is unique up to isomorphism.*

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[rr, bend left, "b^*"] & L \arrow[d, "m"] \arrow[dr, phantom, "PO"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \\
C \arrow[r, "c"'] & G' & C' \arrow[l]
\end{tikzcd}$$

Proof. See Theorem 6.4 in [EEPT06] (page 127). □

Remark A.24. *For* **Graphs** *Definition A.22 is a restricted version of Definition 2.3.7 (gluing condition) since the former only takes into account injective morphisms l. We can also let l be non-injective, but in this case the "$\Leftarrow$" part of Theorem A.23 does not hold in general.*

In the following we recall a lemma that is required to show Theorem 3.2.5 (satisfiability of the gluing condition for borrowed context steps).

Lemma A.25. *Given an initial pushout* (1) *over a mono $f\colon A \to D$ and a pushout* (2) *with m mono, then the composition* (1) + (2) *is an initial pushout over the mono n.*

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[dr, phantom, "(1)"] & A \arrow[r, "m"] \arrow[d, "f"'] \arrow[dr, phantom, "(2)"] & E \arrow[d, "n"'] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

Proof. See item 2. of Lemma 6.5 in [EEPT06] (page 127). □

Appendix B

BC with NACs – Additional Information

Here we give further details about our extension of the borrowed context framework to negative application conditions, defined in Chapter 3.

B.1 Proofs of Lemmas

In this section we present the lemmas required by the proofs of Chapter 3. Lemmas B.1.2, B.1.4, B.1.5, B.1.6 and B.1.7 require the category to be adhesive. Lemma B.1.2 (and hence Lemma B.1.6, which is based on Lemma B.1.2) additionally requires that pullbacks preserve epis.

Lemma B.1.1. *Given the commuting diagram below, where the arrows c and d are jointly epi, the arrow e is an epi if and only if (f, g) is jointly epi.*

$$\begin{tikzcd}
 & B \arrow[d, "c"] \arrow[ddr, bend left, "f"] & \\
C \arrow[r, "d"'] \arrow[drr, bend right, "g"'] & D \arrow[dr, "e"] & = \\
 & = & E
\end{tikzcd}$$

Proof. We consider both directions. Take two arrows $x, y \colon E \to F$.

("$\Leftarrow$") Assume that (f, g) is jointly epi and $x \circ e = y \circ e$. Then we have: $x \circ e \circ c = y \circ e \circ c \Rightarrow x \circ f = y \circ f$ and $x \circ e \circ d = y \circ e \circ d \Rightarrow x \circ g = y \circ g$. Since (f, g) is jointly epi, $x \circ f = y \circ f$ and $x \circ g = y \circ g$ we have $x = y$. Therefore we can conclude that e is an epi.

("⇒") Let e be an epi, $x \circ f = y \circ f$ and $x \circ g = y \circ g$. Then we have: $x \circ e \circ c = x \circ f = y \circ f = y \circ e \circ c$ and $x \circ e \circ d = x \circ g = y \circ g = y \circ e \circ d$. Since (c, d) is jointly epi, $x \circ e \circ c = y \circ e \circ c$ and $x \circ e \circ d = y \circ e \circ d$ we have $x \circ e = y \circ e$. Our assumption that e is an epi implies $x = y$. Thus, we conclude that (f, g) is jointly epi.

□

Lemma B.1.2. *Given a commuting cube with all arrows mono and all lateral sides pushouts, then the pair of arrows (c, d) is jointly epi if and only if (p, q) is jointly epi.*

$$\begin{tikzcd}
& & A \arrow[dll, "a"'] \arrow[drr, "b"] \arrow[dd, "h"] & & \\
B \arrow[dr, "c"] \arrow[dd, "i"'] & & & & C \arrow[dlll, "d"'] \arrow[dd, "j"] \\
& D \arrow[dd, "k"] & F \arrow[dll, "m"'] \arrow[drr, "n"] & & \\
G \arrow[dr, "p"'] & & & & H \arrow[dlll, "q"] \\
& I & & &
\end{tikzcd}$$

Proof. We construct the pushout (r, s, J) of m and n and obtain t as an induced arrow such that $p = t \circ r$ and $q = t \circ s$ (see the upper left diagram below). We proceed analogously for (A, C, B, D) (see the lower left diagram below). Gluing these new monos to the cube gives rise to the cube below, except for $l \colon E \to J$.

$$\begin{tikzcd}
F \arrow[rr, "n"] \arrow[dd, "m"'] \arrow[dr, phantom, "PO"] & & H \arrow[dl, "s"'] \arrow[dd, "q"] \\
& J \arrow[dr, "t"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
G \arrow[rr, "p"'] \arrow[ur, "r"] & {} & I
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[rr, "b"] \arrow[dd, "a"'] \arrow[dr, phantom, "PO"] & & C \arrow[dl, "f"'] \arrow[dd, "d"] \\
& E \arrow[dr, "g"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
B \arrow[rr, "c"'] \arrow[ur, "e"] & {} & D
\end{tikzcd}$$

$$\begin{tikzcd}
& & & A \arrow[dlll, "a"'] \arrow[ddr, "b"] \arrow[dddd, "h"] & \\
B \arrow[drr, "e"] \arrow[ddr, "c"'] \arrow[dddd, "i"'] & & & & \\
& & E \arrow[dl, "g"'] \arrow[dddd, "l"] & & C \arrow[ll, "f"'] \arrow[dlll, "d"] \arrow[dddd, "j"] \\
& D \arrow[dddd, "k"] & & & \\
& & & F \arrow[dlll, "m"'] \arrow[ddr, "n"] & \\
G \arrow[drr, "r"] \arrow[ddr, "p"'] & & & & \\
& & J \arrow[dl, "t"'] & & H \arrow[ll, "s"'] \arrow[dlll, "q"] \\
& I & & &
\end{tikzcd}$$

$$\begin{tikzcd}
C \arrow[rr, "d"] \arrow[ddd, "j"'] \arrow[dr, "f"'] \arrow[ddr, phantom, "(1)"] & {} \arrow[d, phantom, "="] & D \arrow[ddd, "k"] \arrow[ddl, phantom, "(2)"] \\
& E \arrow[ur, "g"'] \arrow[d, "l"] & \\
& J \arrow[ddr, "t"] \arrow[d, phantom, "="] & \\
H \arrow[rr, "q"'] \arrow[uur, "s"] & {} & I
\end{tikzcd}$$

Note that $r \circ i \circ a = r \circ m \circ h = s \circ n \circ h = s \circ j \circ b$ since (A, F, B, G), (F, G, H, J) and (A, F, C, H) commute. So (A, B, C, E) as a pushout and $r \circ i \circ a = s \circ j \circ b$ imply a unique arrow $l \colon E \to J$ such that $l \circ e = r \circ i$ and $l \circ f = s \circ j$. From the inner cube we can infer that $(A, F, C, H) + (F, H, G, J)$ is a pushout and by the commutativity it implies $(A, C, B, E) + (B, E, G, J)$ as a pushout. By pushout decomposition, (B, E, G, J) is a pushout since (A, C, B, E) is a pushout. Analogously, (C, E, H, J) is also a pushout. The rightmost diagram is extracted from the cube: the outer square and (1) are pushouts. By pushout decomposition (2) is also a pushout. Since (1) is a pushout and j is mono then l is mono as well.

Here we will show: (c, d) jointly epi $\Rightarrow$ (p, q) jointly epi. Since (A, B, C, E) is a pushout and (c, d) is jointly epi (assumption), then by Lemma B.1.1 the arrow g is epi. The arrow t is also epi since (2) is a pushout. Finally, by Lemma B.1.1 we obtain that p and q are jointly epi.

The other direction ("$\Leftarrow$") is shown as follows. Since (F, G, H, J) is a pushout and (p, q) is jointly epi (assumption), then by Lemma B.1.1 the arrow t is epi. Since (2) is a pushout and l is mono, then (2) is also a pullback. The pullback (2) and t as epi imply that g is epi as well. So by Lemma B.1.1 (c, d) is jointly epi.

□

Lemma B.1.3 (**Composition of Jointly Epi Arrows**). *Whenever* (1) *is a commuting diagram and* (f, g) *and* (i, j) *are pairs of jointly epi arrows, then the composition* $(f, g \circ j)$ *is also jointly epi.*

$$\begin{tikzcd}
A \arrow[r, "f"] & B \\
C \arrow[u, "x"] \arrow[r, "i"'] \arrow[ur, phantom, "(1)"] & D \arrow[u, "g"'] \\
& E \arrow[u, "j"']
\end{tikzcd}$$

Proof. By Definition 3.3.3 (f, g) and (i, j) jointly epi means: *(i)* $\forall\, a, b \colon B \to F \colon\ a \circ f = b \circ f \,\wedge\, a \circ g = b \circ g \;\Rightarrow\; a = b$ and *(ii)* $\forall\, c, d \colon D \to G \colon\ c \circ i = d \circ i \,\wedge\, c \circ j = d \circ j \;\Rightarrow\; c = d$. We assume that $\forall\, a, b \colon B \to F :\ a \circ f = b \circ f \ \wedge\ a \circ (g \circ j) = b \circ (g \circ j)$ and we will show that this implies $a = b$. Observe that $a \circ g \circ i = a \circ f \circ x$ ((1) commutes) $=$ $b \circ f \circ x$ ($a \circ f = b \circ f$ by assumption) $=$ $b \circ g \circ i$ ((1) commutes). Then we have $a \circ g \circ i = b \circ g \circ i$ (previous calculation) and $a \circ g \circ j = b \circ g \circ j$ (assumption), which implies $a \circ g = b \circ g$ *(iii)*. By using *(i)* and*(iii)* ($a \circ f = b \circ f$ and $a \circ g = b \circ g$, respectively) together we have: $\forall\, a, b \colon B \to F : a \circ f = b \circ f \ \wedge\ a \circ g = b \circ g$, which by *(i)* implies that $a = b$.

□

Lemma B.1.4. *Given the diagram below, where all arrows are mono and* (c, d) *is jointly epi, whenever* b *is an iso then so is* c.

$$\begin{tikzcd}
C \arrow[r, "d"] & D \\
A \arrow[u, "b"] \arrow[r, "a"'] \arrow[ur, phantom, "\overset{j.epi}{=}"] & B \arrow[u, "c"']
\end{tikzcd}$$

Proof. We have to check that c is an iso. Since c is mono and we are working in an adhesive category it is enough to show that c is epi (cf. [LS05]). Assume that there are arrows $x, y\colon D \to X$ with $x \circ c = y \circ c$. Composing with a gives us $x \circ c \circ a = y \circ c \circ a$ and hence $x \circ d \circ b = y \circ d \circ b$. Since b is an iso it follows that $x \circ d = y \circ d$. Finally, since c and d are jointly epi we obtain $x = y$.

□

Lemma B.1.5. *In the diagram below, where all arrows are mono, it holds: whenever $\overline{F} \to \overline{N}$ is not iso then $F \to N$ is not iso as well.*

$$\begin{tikzcd}
N \arrow[r] & M' & \overline{N} \arrow[l] \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "\overset{j.epi}{=}"] & E_2 \arrow[u] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[u] \arrow[l]
\end{tikzcd}$$

Proof. We show the contrapositive: $F \to N$ iso $\Rightarrow$ $\overline{F} \to \overline{N}$ iso. The arrow $F \to N$ as an iso implies by Lemma B.1.4 that $E_2 \to M'$ is also an iso. The pushout along monos is also a pullback and $E_2 \to M'$ iso implies $\overline{F} \to \overline{N}$ iso.

□

Lemma B.1.6 (NAC Compatibility). *In the following let all arrows be mono and let Diagram (B.1) be given.*

If we have Diagram (B.3), then there exist objects M_z, N_z and M'_x such that Diagram (B.2)+(B.4) can be constructed as indicated. Furthermore, if we have Diagram (B.2)+ (B.4), then there exists an object $\overline{M}_x$ such that Diagram (B.3) can be constructed as indicated.

$$\begin{tikzcd}
L \arrow[d] \arrow[dr] & & \\
G^+ \arrow[r] \arrow[ur, phantom, "="] & \overline{G}^+ & \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "PO"] & E_2 \arrow[u] \arrow[ur, phantom, "="] & \overline{F} \arrow[l] \arrow[ul]
\end{tikzcd} \tag{B.1}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & \overline{M}_x & \overline{N}_x \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "\overset{j.epi}{=}"] & \overline{G}^+ \arrow[u] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[u] \arrow[l]
\end{tikzcd} \tag{B.3}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "\overset{j.epi}{=}"] & G^+ \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[l]
\end{tikzcd} \tag{B.2}$$

$$\begin{tikzcd}
N_z \arrow[r] & M'_x & \overline{N}_x \arrow[l] \\
F \arrow[u] \arrow[r] \arrow[ur, phantom, "\overset{j.epi}{=}"] & E_2 \arrow[u] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[u] \arrow[l]
\end{tikzcd} \tag{B.4}$$

Proof. The proof is split into two steps.

Step 1 *(Diagram (B.2)+(B.4) $\Rightarrow$ Diagram (B.3)).* We take the inner squares of Diagram (B.2)+(B.4) and build $\overline{G}^+$ as the pushout of $E_2 \leftarrow F \to G^+$ and $\overline{M}_x$ as the pushout of $M'_x \leftarrow N_z \rightarrow M_z$ (see Diagram (B.5)). Since *(left)* is a pushout and *(back)*, *(right)* and *(top)* commute, then there exists a unique arrow $\overline{G}^+ \rightarrow \overline{M}_x$ such that *(bottom)* and *(front)* commute. By pushout composition and then decomposition we find that *(front)* is a pushout. So $E_2 \rightarrow M'_x$ mono implies $\overline{G}^+ \rightarrow \overline{M}_x$ mono. The arrows $E_2 \rightarrow M'_x$ and $N_z \rightarrow M'_x$ are jointly epi which implies by Lemma B.1.2 that $\overline{G}^+ \rightarrow \overline{M}_x$ and $M_z \rightarrow \overline{M}_x$ are also jointly epi.

$$\begin{tikzcd}
F \arrow[rr] \arrow[dr] \arrow[dd] & & N_z \arrow[dr] \arrow[dd] & \\
& E_2 \arrow[rr] \arrow[dd] & & M'_x \arrow[dd] \\
G^+ \arrow[rr] \arrow[dr] & & M_z \arrow[dr] & \\
& \overline{G}^+ \arrow[rr] & & \overline{M}_x
\end{tikzcd} \quad (B.5)$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z \arrow[r] & \overline{M}_x & M'_x \arrow[l] & \overline{N}_x \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "="] & G^+ \arrow[u, "j.epi"] \arrow[r] \arrow[ur, phantom, "="] & \overline{G}^+ \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & E_2 \arrow[l] \arrow[u] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd}$$

Gluing the leftmost and rightmost squares of Diagram (B.2)+(B.4) to the bottom and front faces of Diagram (B.5) produces the diagram above on the right, which by Lemma B.1.3 and pushout composition is exactly Diagram (B.3). Note that for each choice of M_z and M'_x in Diagram (B.2)+(B.4), there is a unique $\overline{M}_x$ (up to iso) leading to Diagram (B.3).

Step 2 *(Diagram (B.3) ⇒ Diagram (B.2)+(B.4)).* Combining Diagrams (B.1) and (B.3) gives rise to Diagram (B.6). We take all possible factorizations $NAC_y \rightarrow M_z \rightarrow \overline{M}_x$ of $NAC_y \rightarrow \overline{M}_x$ such that there exists an arrow $G^+ \rightarrow M_z$ (see Diagram (B.7)) with (1), (2) commuting and jointly epi and all arrows are monos. At least one such M_z—which can be obtained as the pushout of $L \rightarrow NAC_y$ and $L \rightarrow G^+$—exists. By pushout splitting we find M'_x and both squares are pushouts along monos.

$$\begin{tikzcd}
NAC_y \arrow[rr] & & \overline{M}_x & & \overline{N}_x \arrow[ll] \\
L \arrow[u] \arrow[r] & G^+ \arrow[r] \arrow[u, phantom, "="] & \overline{G}^+ \arrow[u, "j.epi"] & E_2 \arrow[l] \arrow[u, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \quad (B.6)$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z \arrow[r] & \overline{M}_x & M'_x \arrow[l] & \overline{N}_x \arrow[l] \\
L \arrow[u] \arrow[r] \arrow[ur, phantom, "(1)"] & G^+ \arrow[u] \arrow[r] \arrow[ur, phantom, "(2)"] & \overline{G}^+ \arrow[u] \arrow[ur, phantom, "PO"] & E_2 \arrow[l] \arrow[u] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \quad (B.7)$$

Gluing the pushout of Diagram (B.1) to Diagram (B.7) produces Diagram (B.8), except for N_y and its arrows. So the fact that (*left*) + (*front*) is a pushout and the commutativity of *(bottom)* imply that the outer square over E_2 in Diagram (B.9) is a pushout. We construct the pullback $M_z \leftarrow N_z \rightarrow M'_x$ with monos of $M_z \rightarrow \overline{M}_x \leftarrow M'_x$, which induces $F \rightarrow N_z$ (due to the universal property of pullbacks) such that the top and back faces of the cube commute. Hence, $(left) + (front)$ is a pushout with monos and also a pullback. So (*back*) + (*right*) is a pullback as well, but (*right*) is already a pullback, which implies by pullback decomposition that (*back*) is a pullback. Since $G^+ \rightarrow M_z$ is mono, so is $F \rightarrow N_z$. Now (*back*) + (*right*) as a pushout and (*right*) as a pullback imply by special pushout-pullback decomposition that (*back*) and (*right*) are pushouts.

$$\begin{tikzcd}
 & & \overline{F} \arrow[rr] \arrow[dd] & & \overline{N}_x \arrow[dd] \\
 & F \arrow[rr] \arrow[dr] \arrow[ddd] \arrow[drr, phantom, "="] & & N_z \arrow[dr] \arrow[ddd] & \\
 & & E_2 \arrow[rr] \arrow[ddd] & & M'_x \arrow[ddd] \\
L \arrow[rr] \arrow[dr] \arrow[drr, phantom, "="] & & NAC_y \arrow[dr] & & \\
 & G^+ \arrow[rr, "j.epi"] \arrow[dr] \arrow[drr, phantom, "="] & & M_z \arrow[dr] & \\
 & & \overline{G}^+ \arrow[rr, "j.epi"] & & \overline{M}_x
\end{tikzcd} \tag{B.8}$$

$$\begin{tikzcd}
 & E_2 \arrow[dr, bend left] \arrow[d, phantom, "="] & \\
F \arrow[ur, bend left] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & N_z \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & M'_x \arrow[d] \\
G^+ \arrow[r] & M_z \arrow[r] & \overline{M}_x
\end{tikzcd} \tag{B.9}$$

$$\begin{tikzcd}
NAC_y \arrow[r] & M_z & N_z \arrow[l] \arrow[r] & M'_x & \overline{N}_x \arrow[l] \\
L \arrow[r] \arrow[u] \arrow[ur, phantom, "="] & G^+ \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & F \arrow[l] \arrow[r] \arrow[u] \arrow[ur, phantom, "="] & E_2 \arrow[u, "j.epi"] \arrow[ur, phantom, "PO"] & \overline{F} \arrow[l] \arrow[u]
\end{tikzcd} \tag{B.10}$$

Since all lateral sides of the cube are pushouts, the bottom and top faces commute and $\overline{G}^+ \rightarrow \overline{M}_x$ and $M_z \rightarrow \overline{M}_x$ are jointly epi we can infer by Lemma B.1.2 that $N_z \rightarrow M'_x$ and $E_2 \rightarrow M'_x$ are jointly epi as well. By taking the squares we are interested in, we obtain Diagram (B.10), which is Diagram (B.2)+(B.4). Observe that for each choice of $\overline{M}_x$ in Diagram (B.3) and each factor M_z there is a unique N_z leading to Diagram (B.2)+(B.4).

□

Lemma B.1.7. *A borrowed context step (as in Definition 3.3.4) is not NAC consistent whenever there exists a mono $q_y \colon NAC_y \rightarrow G^+$ such that $m = q_y \circ n_y$ (see Definition 3.3.2). This is equivalent to the situation, in which:*

(i) there exists a negative borrowed context $F \rightarrow N_z$ which is an iso;

or

(ii) there exists a mono $N_z \rightarrow G^+$ such that $F \rightarrow G^+ = F \rightarrow N_z \rightarrow G^+$.

Proof. We have to show: $\exists\, q_y \colon NAC_y \rightarrow G^+$ mono with $m = q_y \circ n_y \Leftrightarrow$ ($\exists$ negative borrowed context $F \rightarrow N_z$ which is iso) or ($\exists\, N_z \rightarrow G^+$ mono with $F \rightarrow G^+ = F \rightarrow N_z \rightarrow G^+$).

$$\begin{tikzcd}
NAC_y \arrow[r, "q_y"] \arrow[dr, phantom, "="] & G^+ \arrow[dr, phantom, "PO"] & F \arrow[l] & (1) \\
L \arrow[u, "n_y"] \arrow[r, "m"'] & G^+ \arrow[u, "j.epi", "\wr"'] & F \arrow[l] \arrow[u, "\wr"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
NAC_y \arrow[r, "q_y"] \arrow[dr, phantom, "="] & M_z \arrow[dr, phantom, "PO"] & N_z \arrow[l] & (2) \\
L \arrow[u, "n_y"] \arrow[r, "m"'] & G^+ \arrow[u, "j.epi\ m'", "\wr"'] & F \arrow[l] \arrow[u] &
\end{tikzcd}$$

("$\Rightarrow$"). Assume there exists a mono $q_y \colon NAC_y \rightarrow G^+$ with $m = q_y \circ n_y$ (left square of Diagram (1)). The arrows q_y and id_{G^+} are jointly epi. We show "(i)": a pushout along monos is also a pullback, and hence we can infer $F \rightarrow N_z$ is also an iso (id_F). Now we show "(ii)": $F \rightarrow G^+ \overset{id_{G^+}}{\rightarrow} G^+ = F \rightarrow N_z \rightarrow G^+$ (pushout). Let $N_z \rightarrow G^+ = N_z \rightarrow G^+ \overset{id_{G^+}^{-1}}{\rightarrow} G^+$. Then we have $F \rightarrow G^+ = F \rightarrow N_z \rightarrow G^+ \overset{id_{G^+}^{-1}}{\rightarrow} G^+$ which is $F \rightarrow G^+ = F \rightarrow N_z \rightarrow G^+$.

("$\Leftarrow$"). "(i)": Suppose $F \rightarrow N_z$ is an iso in Diagram (2). The pushout implies $m' \colon G^+ \rightarrow M_z$ iso, and therefore $m' \circ m = q_y \circ n_y$. Let $q_y = m'^{-1} \circ q_y$ then we have $m = q_y \circ n_y$. Observe that q_y mono and m' iso are clearly jointly epi. "(ii)": assume there exists a mono $N_z \rightarrow G^+$ with $F \rightarrow G^+ = F \rightarrow N_z \rightarrow G^+$. The pushout implies $m' \colon G^+ \rightarrow M_z$ iso. Let $q_y = m'^{-1} \circ q_y$. Then we obtain $m = q_y \circ n_y$.

□

B.2 Objects with Interfaces as Venn Diagrams

We depict some diagrams of Theorem 3.3.10 as Venn diagrams. The left-hand side L of a production, its interface I, the object G and its interface J are shown as circles (see Figure B.1). The NAC is the circle L together with the "boomerang"-shaped area. Figure B.1 shows typical overlaps between an object $J \rightarrow G$ and the left-hand side of a production that occur when a BC step takes place. Depending on the situation the NAC might have a bigger overlapping structure with G, which—in the picture—means that the NAC is rotated counterclockwise. On the right we show an overlapping when $J \rightarrow G$ is inserted into a context $J \rightarrow E \leftarrow \overline{J}$.

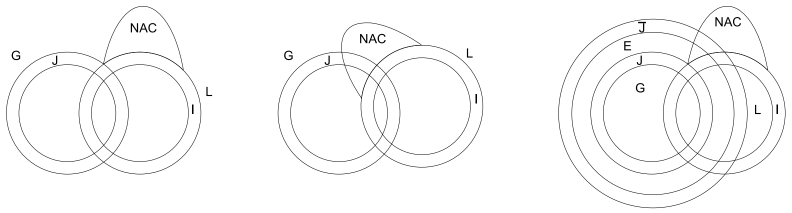


Figure B.1: Overlaps between $J \rightarrow G$ and a production $NAC \leftarrow L \leftarrow I \rightarrow R$

Figure B.2 shows graphical representations of Diagrams (3.12), (3.11) and (3.13) of Theorem 3.3.10. Non-empty areas, i.e., areas where items are present, are shaded gray.

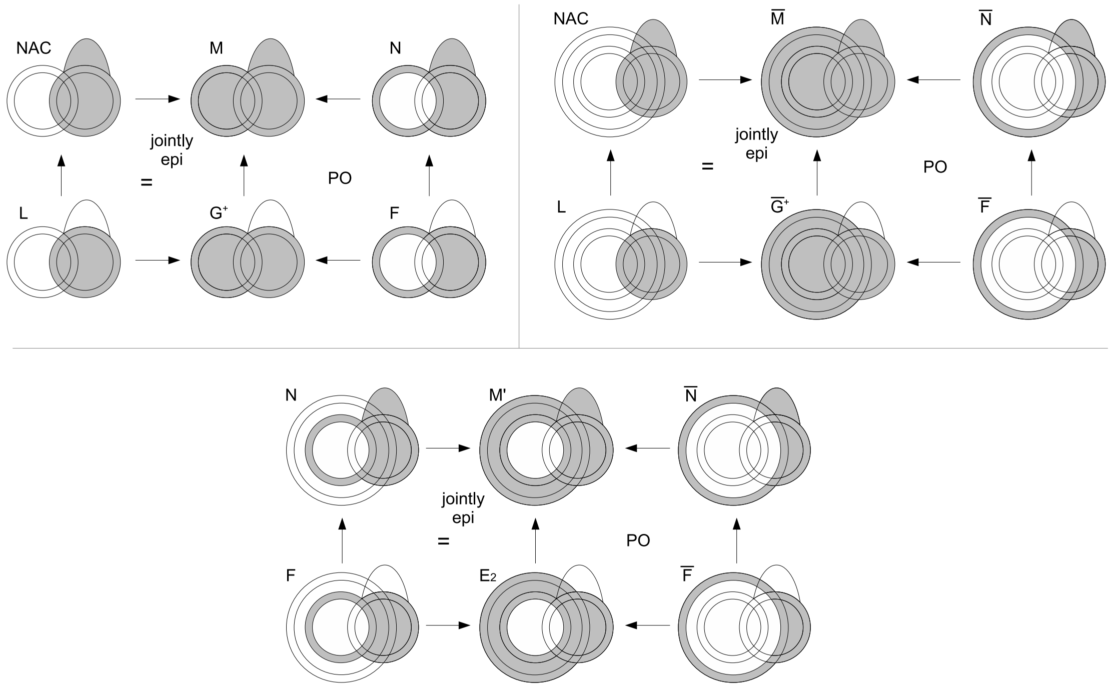


Figure B.2: Diagrams (3.12), (3.11) and (3.13) of Theorem 3.3.10.

The top-left diagram in Figure B.2 shows the construction of negative borrowed contexts of the form $F \rightarrow N$ for the BC step of $J \rightarrow G$. The objects G^+ and NAC overlap, giving rise to M. The object G^+ is G plus the borrowed context F. So N represents exactly what should not be further provided by the environment in order to guarantee the feasibility of the BC step. Note that N does not contain any information about G which is not "visible" from the interface J.

The top-right diagram in Figure B.2 depicts how the negative borrowed context $\overline{F} \to \overline{N}$ is built for the BC step of $\overline{J} \to \overline{G}$. Also note that $\overline{N}$ does not contain any information about $\overline{G}$ which is not already present in the interface $\overline{J}$.

Finally, the diagram in the second row of Figure B.2 represents the translation of a negative borrowed context with respect to a new context that is added. Observe that the translation is based on the context E_2, which does not depend on the contents of G, and therefore can be used for both parts of the proof of Theorem 3.3.10.

Bibliography

[AG97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AGG] AGG – the attributed graph grammar system. http://tfs.cs.tu-berlin.de/agg/.

[Bat06] G. V. Batz. An optimization technique for subgraph matching strategies. Internal Report 2006-7, Fakultät für Informatik, Universität Karlsruhe, September 2006.

[BEK06a] P. Baldan, H. Ehrig, and B. König. Composition and decomposition of DPO transformations with borrowed context. In *Proc. of ICGT '06 (International Conference on Graph Transformation)*, volume 4178 of *LNCS*, pages 153–167. Springer, 2006.

[BEK06b] P. Baldan, H. Ehrig, and B. König. Composition and decomposition of DPO transformations with borrowed context. Technical Report 2006-1, Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, October 2006.

[BEK$^+$06c] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF model refactoring based on graph transformation concepts. In *Proc. of SETra'06 (International Workshop on Software Evolution through Transformations)*, volume 3 of *Electronic Communications of the EASST*, 2006.

[BF01] K. Beck and M. Fowler. *Planning Extreme Programming.* Addison Wesley, 2001.

[BGH08] F. Bonchi, F. Gadducci, and T. Heindel. Parallel and sequential independence for borrowed contexts. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*, volume 5214 of *LNCS*, pages 226–241. Springer, 2008.

[BGK06a] F. Bonchi, F. Gadducci, and B. König. Process bisimulation via a graphical encoding. In *Proc. of ICGT '06 (International Conference on Graph Transformation)*, volume 4178 of *LNCS*, pages 168–183. Springer, 2006.

[BGK06b] F. Bonchi, F. Gadducci, and B. König. Process bisimulation via a graphical encoding. Technical Report 06-07, Department of Computer Science, University of Pisa, July 2006.

[BHE08] D. Bisztray, R. Heckel, and H. Ehrig. Verification of architectural refactorings by rule extraction. In *Proc. of FASE'08 (Fundamental Approaches to Software Engineering)*, volume 4961 of *LNCS*, pages 347–361. Springer, 2008.

[BKM06] F. Bonchi, B. König, and U. Montanari. Saturated semantics for reactive systems. In *Proc. of LICS '06 (Logic in Computer Science)*, pages 69–80. IEEE, 2006.

[Bro86] F. P. Brooks. No silver bullet — essence and accidents of software engineering. *Information Processing*, 86:1069–1076, 1986.

[BW95] M. Barr and C. Wells. *Category theory for computing science (2nd Edition).* Prentice Hall, 1995.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proc. of FoSSaCS'98 (Foundations of Software Science and Computation Structures)*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[CMR$^+$97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[EE06] H. Ehrig and K. Ehrig. Overview of formal concepts for model transformations based on typed attributed graph transformation. In *Proc. of GraMoT '05 (International Workshop on Graph and Model Transformation)*, volume 152 of *ENTCS*, pages 3–22. Elsevier Science, 2006.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: volume 2: applications, languages, and tools.* World Scientific, 1999.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer, 2006.

[EK04] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Proc. of FoSSaCS '04 (Foundations of Software Science and Computation Structures)*, volume 2987 of *LNCS*, pages 151–166, 2004.

[EK06] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. *Mathematical Structures in Computer Science*, 16(6):1133–1163, 2006.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: volume 3: concurrency, parallelism, and distribution.* World Scientific, 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification II: module specifications and constraints.* Springer, 1990.

[EW06] K. Ehrig and J. Winkelmann. Model transformation from visual OCL to OCL using graph transformation. In *Proc. of GraMoT '05 (International Workshop on Graph and Model Transformation)*, volume 152 of *ENTCS*, pages 23–37, 2006.

[FG96] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proc. of POPL '96 (Principles of Programming Languages)*, pages 372–385. ACM Press, 1996.

[FM91] J.-C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proc. of CAV'91 (International Conference on Computer Aided Verification)*, volume 757 of *LNCS*, pages 181–191. Springer, 1991.

[Fuj] Fujaba – tool suite. http://wwwcs.uni-paderborn.de/cs/fujaba/.

[Gib94] W. W. Gibbs. Software's chronic crisis. *Scientific American*, pages 86–95, 1994.

[Gla01] R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

[Gra] Graphviz - graph visualization software. http://www.graphviz.org.

[GSMD03] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In *Proc. of UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 144–158. Springer, 2003.

[GZ05] L. Geiger and A. Zündorf. Statechart modeling with fujaba. *ENTCS*, 127(1):37–49, 2005.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HC98] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design DeBugging.* Kluwer Academic Publishers, 1998.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[Hir01] D. Hirschkoff. Bisimulation verification using the up-to techniques. *International Journal on Software Tools for Technology Transfer*, 3(3):271–285, August 2001.

[HJE06] B. Hoffmann, D. Janssens, and N. Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. *ENTCS*, 152:53–67, 2006.

[HMU00] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[JGP99] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[Kik05] T. Kikuno. Why do software projects fail? reasons and a solution using a bayesian classifier to predict potential risk. In *Proc. of PRDC'05 (Pacific Rim International Symposium on Dependable Computing)*, page 4. IEEE Computer Society, 2005.

[Lam07] L. Lambers. Adhesive high-level replacement system with negative application conditions. Technical Report 2007/14, TU Berlin, 2007.

[Lei01] J. J. Leifer. *Operational Congruences for Reactive Systems.* PhD thesis, University of Cambridge Computer Laboratory, 2001.

[Lei02] J. J. Leifer. Synthesising labelled transitions and operational congruences in reactive systems, part 2. Technical Report RR-4395, INRIA Rocquencourt, 2002.

[LM00] J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *Proc. of CONCUR'00 (International Conference on Concurrency Theory)*, volume 1877 of *LNCS*, pages 243–258. Springer, 2000.

[LS04] S. Lack and P. Sobociński. Adhesive categories. In *Proc. of FoSSaCS'04 (Foundations of Software Science and Computation Structures)*, volume 2987 of *LNCS*, pages 273–288. Springer, 2004.

[LS05] S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO - Theoretical Informatics and Applications*, 39(2):522–546, 2005.

[Mar95] A. Martini. Elements of basic category theory. Technical Report 96-5, TU Berlin, 1995.

[MG06] T. Mens and P. Van Gorp. A taxonomy of model transformation. *ENTCS*, 152:125–142, 2006.

[MH08] M. Minas and B. Hoffmann. An example of cloning graph transformation rules for programming. *ENTCS*, 211:241–250, 2008.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[Mil93] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification.* Springer, 1993.

[Mil96] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[Mil01] R. Milner. Bigraphical reactive systems. In *Proc. of CONCUR'01 (International Conference on Concurrency Theory)*, pages 16–35. Springer, 2001.

[Mil06] R. Milner. Pure bigraphs: structure and dynamics. *Information and Computation*, 204(1):60–122, 2006.

[MP92] R. Milner and J. Parrow. A calculus for mobile process I. *Information and Computation*, 100:1–40, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus for mobile process II. *Information and Computation*, 100:41–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. of ICALP'92 (International Colloquium on Automata, Languages and Programming)*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[MSPT96] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *Proc. of CONCUR '96 (International Conference on Concurrency Theory)*, volume 1119 of *LNCS*, pages 687–702. Springer, 1996.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[MTR07] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.

[New01] M. Newborn. *Automated theorem proving: theory and practice.* Springer, 2001.

[NK06] A. Narayanan and G. Karsai. Towards verifying model transformations. In *Proc. of GT-VMT '06 (International Workshop on Graph Transformation and Visual Modeling Techniques)*, ENTCS, pages 185–194, 2006.

[OCa] Objective Caml. http://caml.inria.fr/ocaml/.

[PC07] J. Pérez and Y. Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In *Proc. of International ERCIM Workshop on Software Evolution*, pages 114–122. ERCIM, 2007.

[Pie91] B. C. Pierce. *Basic category theory for computer scientists.* MIT Press, 1991.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PV98] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS '98 (Logic in Computer Science)*. IEEE, Computer Society Press, 1998.

[Rei85] W. Reisig. *Petri nets.* Springer Verlag, 1985.

[Ren04] A. Rensink. Representing first-order logic using graphs. In *Proc. of ICGT '04 (International Conference on Graph Transformation)*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.

[Ren06] A. Rensink. Isomorphism checking in GROOVE. In *Proc. of GraBaTs'06 (International Workshop on Graph-Based Tools)*, volume 1 of *ENTCS*. Elsevier Science, 2006.

[RKE07] G. Rangel, B. König, and Hartmut Ehrig. Bisimulation verification for the DPO approach with borrowed contexts. In *Proc. of GT-VMT '07 (International Workshop on Graph Transformation and Visual Modeling Techniques)*, volume 6 of *Electronic Communications of the EASST*, 2007.

[RKE08a] G. Rangel, B. König, and H. Ehrig. Deriving bisimulation congruences in the presence of negative application conditions. In *Proc. of FoSSaCS '08 (Foundations of Software Science and Computation Structures)*, volume 4962 of *LNCS*, pages 413–427. Springer, 2008.

[RKE08b] G. Rangel, B. König, and H. Ehrig. Deriving bisimulation congruences in the presence of negative application conditions. Technical Report 2008-1, Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, 2008.

[RLK$^+$08a] G. Rangel, L. Lambers, B. König, H. Ehrig, and P. Baldan. Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*, volume 5214 of *LNCS*, pages 242–256. Springer, 2008.

[RLK$^+$08b] G. Rangel, L. Lambers, B. König, H. Ehrig, and P. Baldan. Behavior preservation in model refactoring using DPO transformations with borrowed contexts. Technical Report 12/08, TU Berlin, 2008.

[Roz97] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume 1. Foundations.* World Scientific, 1997.

[RSS07] J. Rathke, V. Sassone, and P. Sobociński. Semantic barbs and biorthogonality. In *Proc. of FoSSaCS '07 (Foundations of Software Science and Computation Structures)*, volume 4423 of *LNCS*, pages 302–316. Springer, 2007.

[Rud98] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Proc. of TAGT'98 (International Workshop on Theory and Application of Graph Transformations)*, volume 1764 of *LNCS*, pages 238–251. Springer, 1998.

[San95] D. Sangiorgi. On the proof method for bisimulation. In *Proc. of MFCS '95 (International Symposium on Mathematical Foundations of Computer Science)*, volume 969 of *LNCS*, pages 479–488. Springer, 1995.

[Sew98] P. Sewell. From rewrite to bisimulation congruences. In *Proc. of CONCUR'98 (International Conference on Concurrency Theory)*, volume 1466 of *LNCS*, pages 269–284. Springer, 1998.

[Sew02] P. Sewell. From rewrite rules to bisimulation congruences. *Theoretical Computer Science*, 274(1-2):183–230, 2002.

[Sob04] P. Sobociński. *Deriving process congruences from reaction rules.* PhD thesis, Department of Computer Science, University of Aarhus, 2004.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice.* John Wiley and Sons Ltd., 1993.

[SS03a] V. Sassone and P. Sobociński. Deriving bisimulation congruences: 2-categories vs precategories. In *Proc. of FoSSaCS '03 (Foundations of Software Science and Computation Structures)*, volume 2620 of *LNCS*, pages 409–424. Springer, 2003.

[SS03b] V. Sassone and P. Sobociński. Deriving bisimulation congruences using 2-categories. *Nordic Journal of Computing*, 10(2):163–183, 2003.

[SS05] V. Sassone and P. Sobociński. Reactive systems over cospans. In *Proc. of LICS '05 (Logic in Computer Science)*, pages 311–320. IEEE, 2005.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus - A Theory of Mobile Processes.* Cambridge University Press, 2001.

[vKCKB05] M. van Kempen, M. Chaudron, D. Kourie, and A. Boake. Towards proving preservation of behaviour of refactoring of UML models. In *Proc. of SAICSIT '05*, pages 252–259. South African Institute for Computer Scientists and Information Technologists, 2005.

[VVGE+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination analysis of model transformations by Petri nets. In *Proc. of ICGT '06 (International Conference on Graph Transformation)*, volume 4178 of *LNCS*, pages 260–274. Springer, 2006.

[Wes02] J. C. Westland. The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1):1–9, 2002.

[WH89] P. Winston and B. Horn. *LISP (3rd Edition)*. Addison Wesley, 1989.