Analysis and Verification of Systems with Dynamically Evolving Structure

Habilitationsschrift zur Erlangung der Venia Legendi
in Informatik

vorgelegt von
Dr. rer. nat. Barbara König

Institut für Formale Methoden der Informatik
Abteilung Sichere und Zuverlässige Softwaresysteme
Fakultät Informatik, Elektrotechnik und Informationstechnik
Universität Stuttgart

Dezember 2004

Abstract

This thesis is concerned with verification and analysis techniques for software systems characterized by dynamically evolving structure, such as dynamic creation and deletion of objects, mobility and variable topology. Examples for such systems are pointer structures, object-based systems and communication protocols in which the number of participants is not constant.

The approach taken here is based on graph transformation systems, an intuitive and—at the same time—powerful formalism for the modelling of distributed and mobile systems. So far there exists comparatively little research concerning the verification of graph rewriting.

We will—in the first part of this thesis—introduce graph transformations and give an overview of existing analysis and verification methods, with a focus on the verification of systems with dynamically evolving structure. Then we will describe three original lines of research: behavioural equivalences, type systems and approximation by Petri nets, all of them concerned with the analysis of graph transformation systems.

The second part consists of eight refereed research papers treating the previously introduced analysis and verification techniques in depth.

Contents

Part I

A General Introduction

Chapter 1

Introduction

Computers, networks and software are becoming increasingly more complex and interconnected. There exists even the notion of a *global ubiquitous computer* with and in which we will some day be living. In such a situation where computers and software systems determine and shape our lives, it is important not only to be able to *build* these systems, but also to *understand* existing systems, whether it is by testing, diagnosis, or verification and analysis.

Especially systems that are characterized by their dynamic evolution, mobility, creation and deletion of objects and their infinite-state space are difficult to handle with today's methods. Being able to cope with these systems is important for analyzing communication protocols, mobile systems, object-based systems, and also—on a somewhat smaller scale—programs handling complex pointer structures on the heap.

The approach taken in this thesis is to base verification and analysis methods on a natural and relatively simple modelling language that has been studied for quite some time: graph transformation systems. Graph transformations are a formalism which captures the essence of the problematic features, such as connectedness, mobility, topology and dynamics. States are represented by graphs, while system evolution is described by graph transformation rules. Research concerned with the verification and analysis of these systems started only fairly recently.

Established analysis and verification techniques such as bisimulations, type systems and abstract interpretation can—to some extent—be carried over to the field of graph rewriting and hence to systems with dynamically evolving structure. However, this transfer is not straightforward, since many new problems have to be addressed in this setting and hence new techniques and methods have to be developed. Furthermore it is possible to use concepts from concurrency theory such as unfoldings and partial order reductions.

Naturally, this is just a first step and several important questions will remain unanswered for the moment. But we hope that many interesting ideas and methods can be found in this thesis and that it is a step into the right direction towards being able to analyze complex dynamically evolving structures.

This habilitation thesis consists of two parts:

- *A General Introduction:* In this part we will give an overview of the contents of this thesis. This overview is meant to be informal, it will be a tutorial-style introduction concentrating on intuition rather than on formal definitions. The technical content is contained in various papers in the second part.

  We will first introduce graph transformation systems (Chapter 2) and then survey existing verification and analysis techniques, especially those which are dealing with dynamically evolving structures (Chapter 3). Afterwards we will introduce three different verification techniques for graph transformation systems: behavioural equivalences (Chapter 4), type systems (Chapter 5) and abstract interpretation by approximating graph transformation systems by Petri nets (Chapter 6). Finally we end this part with a short conclusion (Chapter 7).

- *Contributions:* The second part consists of eight research papers containing original contributions to this area. These papers have been accepted for conferences or journals. Compared to earlier versions, the formatting has been changed and in some places proofs that were missing in the conference versions have been added for completeness. A list of these eight papers can be found on page 66.

Hence the contents of this thesis can be summarized in the following table.

Chapter	Title	Remarks
	Part I: A General Introduction	
1	Introduction	
2	Modelling Dynamically Evolving Structures using Graph Rewriting	
3	A Short Summary of Methods and Techniques in Static Analysis and Verification	
3	Behavioural Equivalences for Graphs	Summarizes Chapters 8 and 9.
5	Assigning Types to Graphs	Summarizes Chapters 10 and 11.
6	Approximating Graphs by Petri Nets—An Unfolding-Based Approach	Summarizes Chapters 12, 13, 14, and 15.
7	Conclusion	
	Part II: Contributions	
8	Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting	Contains [EK04a].

9	Observational Equivalence for Synchronized Graph Rewriting with Mobility	Contains [KM01].
10	A General Framework for Types in Graph Rewriting	Contains [Kön00a].
11	Analysing Input/Output-Capabilities of Mobile Processes with a Generic Type System	Contains [Kön].
12	A Static Analysis Technique for Graph Transformation Systems	Contains [BCK01].
13	Approximating the Behaviour of Graph Transformation Systems	Contains [BK02].
14	A Logic for Analyzing Abstractions of Graph Transformation Systems	Contains [BKK03].
15	Verifying Finite-State Graph Grammars: an Unfolding-Based Approach	Contains [BCK04b].

Chapter 2

Modelling Dynamically Evolving Structures using Graph Rewriting

2.1 An Introduction to Graph Rewriting

Many structures in computer science can be described by graphs: computer networks, communicating processes, pointer structures on the heap, UML diagrams, and many others. These graphs are mostly static descriptions of system states. Adding dynamics requires some means to describe state changes, such as graph transformation rules. The theory of graph transformation systems has its origins in the 1970's and a rich theory is now available [Roz97]. It has found many applications in the modelling of concurrent systems [EKMR99] and in other areas such as software engineering, database design, VLSI layout, bioinformatics and visual languages [EEKR99].

Graph rewriting is well-suited for the specification of dynamically evolving structures, possessing features such as dynamic creation of objects, mobility and variable topology. Furthermore, graph transformation systems are a very intuitive, natural and general framework. This makes them suitable for an underlying specification language, on which fundamental methods for the verification of dynamically evolving structures can be based.

However, until recently there has been very little work on static analysis and verification of graph transformation systems. This is probably partly due to the fact that research in this area so far had a strong focus on semantic issues such as rewriting formalisms, expressiveness and concurrency. Furthermore research on verification techniques has only recently reached a point where it seems feasible to tackle systems of such an inherent complexity, which can model mobility and dynamically changing topologies. More details concerning related work in static analysis and verification can be found in Chapter 3.

In our setting we will usually use hypergraph rewriting. Hypergraphs are an extension of directed graphs where every edge has its own identity and is connected to an arbitrarily long sequence of nodes. Furthermore they are convenient for the specification of concurrent systems, for instance a process with m external ports can be modelled by an m-ary hyperedge. Rewriting rules for graphs are similar to productions in Chomsky grammars, consisting of a left-hand side (the graph to be rewritten) and a right-hand side (the graph to be created). Different from string or term rewriting, we additionally provide an interface graph, describing the parts of the original graph that are to be preserved. The interface is furthermore important for specifying how the right-hand side should be attached to the remaining original graph.

In the rest of this chapter we will first define hypergraphs and hypergraph morphisms, then we will formally introduce graph rewriting in two different ways (based on sets and based on category theory). In the end we will give some recipes describing how concurrent and other systems can be modelled in a convenient way using graph rewriting.

2.2 Hypergraphs and Hypergraph Morphisms

We define hypergraphs as a generalization of directed graphs. In a hypergraph, hyperedges sharing common nodes are considered as the central components.

We assume that there is a fixed set Λ of labels. Unless said otherwise we will only consider finite hypergraphs, i.e., hypergraphs with finite node and edges sets.

Definition 2.2.1 (Hypergraph) *Let Λ be a set of labels. A* hypergraph G *is a tuple (V_G, E_G, c_G, l_G), where V_G is a set of nodes, E_G is a set of edges, $c_G\colon E_G \to {V_G}^*$ is a connection function and $l_G\colon E_G \to \Lambda$ is the labelling function for edges satisfying $ar(l_G(e)) = |c_G(e)|$ for every $e \in E_G$. Nodes are unlabelled.*

As usual, morphisms are mappings that preserve the structure of the objects under consideration.

Definition 2.2.2 (Hypergraph morphisms) *Let G, G' be hypergraphs. A* hypergraph morphism $\varphi\colon G \to G'$ *consists of a pair of functions $(\varphi_V\colon V_G \to V_{G'},\ \varphi_E\colon E_G \to E_{G'})$ such that for every $e \in E_G$ it holds that $l_G(e) = l_{G'}(\varphi_E(e))$ and*[1] ${\varphi_V}^*(c_G(e)) = c_{G'}(\varphi_E(e))$.

A hypergraph morphism $\varphi\colon G \to G'$ is called isomorphism/injective morphism *whenever φ_V and φ_E are bijective/injective. In this case we say that G and G' are isomorphic (in symbols: $G \cong G'$).*

In the following we will usually drop the indices of the mappings φ_V, φ_E. Furthermore we will use the terms *graph* and *hypergraph* interchangeably. We are interested in graphs only up to isomorphism, i.e., we will often consider isomorphism classes of graphs.

Hyperedges will be depicted using rectangles with rounded corners containing its label, connected to nodes via so-called tentacles (see the left-hand side edge of Figure 2.1 for a hyperedge with label $A \in \Lambda$). Since the position of a

[1] By ${\varphi_V}^*(s)$ we denote the pointwise application of φ_V to every element of the string s.

node in the string $c_G(e)$ of nodes attached to an edge e matters, we will usually number the tentacles. The string $c_G(e)$ may contain duplicates. Whenever e is binary, i.e., $c_G(e) = v_1 v_2$, we depict the hyperedge by drawing an arrow from v_1 to v_2, as in the case of directed graphs (see the right-hand side edge of Figure 2.1).

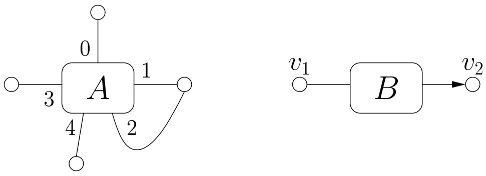


Figure 2.1: Two ways of drawing hyperedges.

2.3 A Set-Based Notion of Graph Rewriting

As described above, graph rewriting rules consist of a left-hand side, a right-hand side and an interface graph, which is embedded into the other graphs by graph morphisms.

Definition 2.3.1 (Rewriting rule) *A* graph rewriting rule r *is a tuple*

$$(L \stackrel{\varphi_L}{\leftarrow} I \stackrel{\varphi_R}{\rightarrow} R)$$

where $\varphi_L\colon I \to L$ and $\varphi_R\colon I \to R$ are injective graph morphisms.

A graph rewriting rule as defined above is often written $L \leftarrow I \rightarrow R$. It is depicted in Figure 2.2 in a schematic way. Concrete examples can be found in Figure 2.5.

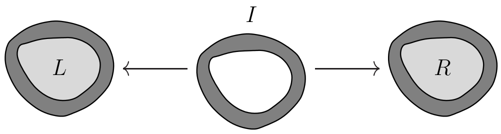


Figure 2.2: Schematic representation of a graph rewriting rule.

Whenever we detect a left-hand side L in a graph G we can apply the corresponding rule. We preserve its interface, but remove all other components of L from G. Then we attach the interface of R to the preserved interface in G. When doing this we have to make sure that we do not remove a node attached to an edge that is preserved, this is ensured with the so-called dangling edge condition. Otherwise the application of the rewriting rule is disallowed, since the resulting graph is not well-defined.

Definition 2.3.2 (Graph rewriting) *Let $r = (L \stackrel{\varphi_L}{\leftarrow} I \stackrel{\varphi_R}{\rightarrow} R)$ be a graph rewriting rule and let $\varphi\colon L \to G$ be an injective match of the left-hand side* *of r in G. Since φ, φ_L and φ_R are injective and we are interested in graphs only up to isomorphism, we can assume that all morphisms are set-theoretic inclusions, i.e., $\varphi(x) = x$, $\varphi_L(x) = x$ and $\varphi_R(x) = x$ for all items x of the respective graphs. Furthermore we assume that the sets of elements of G and R that are not in the range of $\varphi \circ \varphi_L$ respectively φ_R are disjoint.*

Furthermore it holds for every $e \in E_G - E_L$ that no node attached to e is contained in $V_L - V_R$ (dangling edge condition).

By applying r to the match φ we obtain a new graph H, written $G \stackrel{r}{\Rightarrow} H$ (or simply $G \Rightarrow H$), which is defined as follows:

$$V_H = (V_G - V_L) \uplus V_R \qquad E_H = (E_G - E_L) \uplus E_R$$

$$c_H = \begin{cases} c_G(e) & \text{if } e \in E_G - E_L \\ c_R(e) & \text{if } e \in E_R \end{cases} \qquad l_H(e) = \begin{cases} l_G(e) & \text{if } e \in E_G - E_L \\ l_R(e) & \text{if } e \in E_R \end{cases}$$

We denote by $\Rightarrow^*$ the transitive closure of the rewriting relation $\Rightarrow$.

In the schematic representation, the application of a rule $L \leftarrow I \rightarrow R$ to a graph G produces the rewriting step $G \Rightarrow H$ depicted in Figure 2.3.

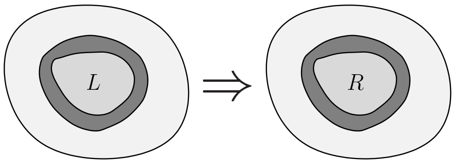


Figure 2.3: Schematic representation of a rewriting step.

Note that the theory has been extended to non-injective rule spans and non-injective matches.

We finally define the notion of a graph transformation system which consists of a set of rules and a start graph.

Definition 2.3.3 (Graph transformation system) *A* graph transformation system *(or* GTS*) $\mathcal{G} = (\mathcal{R}, G_0)$ consists of a set of rewriting rules $\mathcal{R}$ and a* start graph *G_0. We say that a graph G is generated by $\mathcal{G}$ whenever $G_0 \Rightarrow^* G$.*

Some of the verification techniques will require some restrictions on rules, for instance we may forbid to delete nodes or we may require that the interface is discrete, i.e., it consists of nodes only. In other cases it is convenient to consider an extension of the graph rewriting mechanism described above, by allowing non-injective matches of the left-hand side. Note that these modifications are of a rather technical nature and that all graph rewriting formalisms considered in this thesis are Turing-complete and thus hard to analyze.

Example 2.3.4 We will now introduce a running example being used in the rest of this part of the thesis. The example will also reappear in Chapter 14.

Consider a system where processes compete for resources R_1 and R_2. A process needs both resources in order to perform some task. The system is represented as a GTS Sys as follows: graph nodes are standing for ports or connection points used for attaching components. These components are modelled by hyperedges, where a hyperedge has tentacles to all ports it is connected with. In this specific example it is sufficient to consider binary edges only.

We consider edges labelled by R_1, R_2, R_1^f, R_2^f standing for assigned and free resources, respectively, and P_1, P_2 and P_3 denoting the states of a process waiting for resource R_1, a process waiting for resource R_2 and a process holding both resources, respectively. Furthermore, edges labelled by D_1 and D_2 represent the demand of a process and connect the target node of a process and the source node of a resource when the process is asking for the resource. When the target node of a resource coincides with the source node of a process, this means that the resource is assigned to the process. The initial scenario for Sys is represented in Figure 2.4, with a single process P_1 asking for both resources.

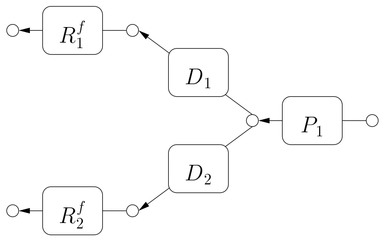


Figure 2.4: Start graph of Sys with one process and two resources.

Figure 2.5 shows the three rules of the graph transformation system. In this case, the interface graphs are always discrete. These nodes and their corresponding images in the left-hand and right-hand sides are numbered in order to indicate the graph morphisms.

Rule [AcquireResource$_{i,j}$] can be instantiated with $i = j = 1$ and $i = j = 2$ (we will consider different instantiations later) and shows how a free resource R_i^f is assigned to a process which subsequently changes its state either from P_1 to P_2 or from P_2 to P_3. A process being in possession of both resources can free them and switch to state P_1 (see rule [ReleaseResources]). The system described by these two rules has only finitely many states (up to isomorphism) and is thus not very challenging. In order to obtain a truly infinite-state system we add rule [CreateNewProcess] describing the forking of a process creating a new process having access to the same resources.

It is essential that we restrict ourselves to instances of rule [AcquireResource$_{i,j}$], where $i = j$. If we would also add rule [AcquireResource$_{1,2}$], where a process in state P_1 can acquire resource R_2, the initial graph could be rewritten to a deadlock situation, i.e., there is a vicious circle where processes are waiting for each other to release their resources (see Figure 2.6).

Note that this is only a toy example being used to illustrate the main concepts of this thesis. References to more elaborate case studies can be found in Section 6.8.

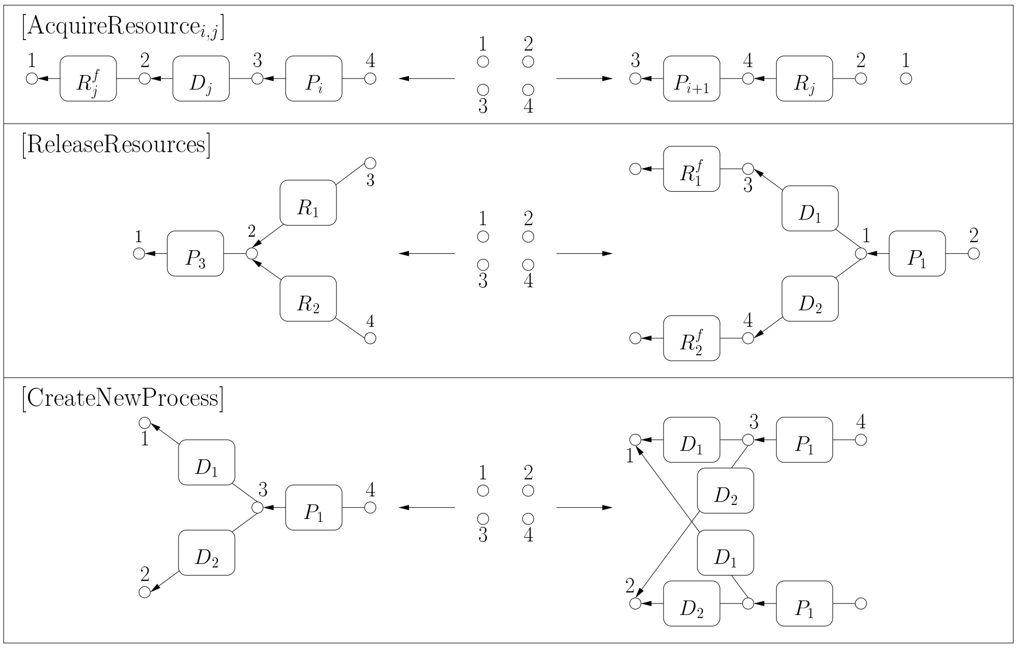


Figure 2.5: Rewriting rules for Sys.

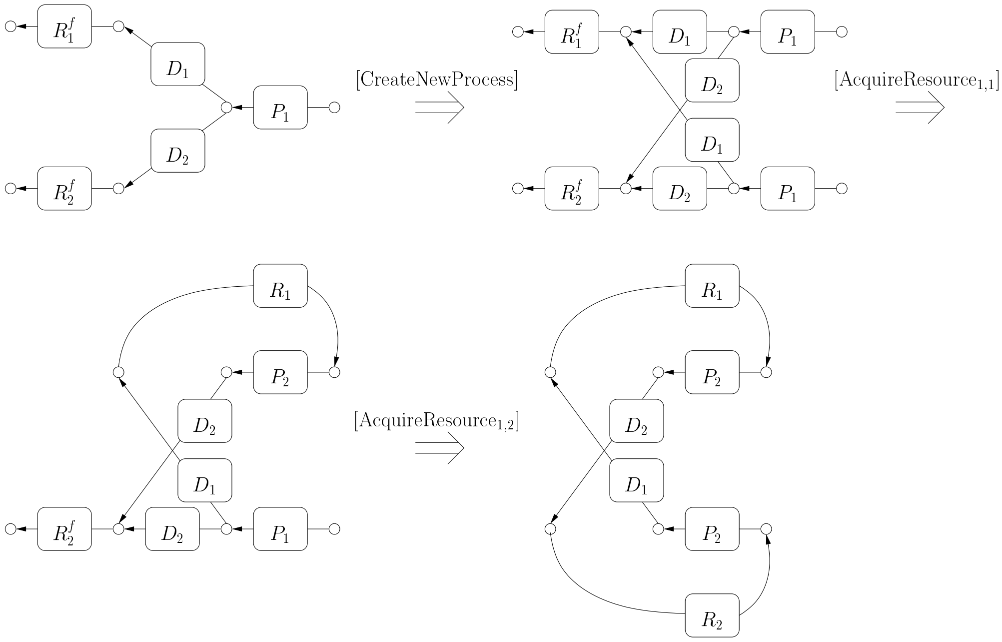


Figure 2.6: Creation of a vicious cycle representing a deadlock.

2.4 Category Theory for Graph Rewriting—The Double-Pushout Approach

The set-based notion of graph rewriting has several drawbacks, especially when used in proofs. It is very cumbersome to construct new graphs by taking union and intersection, by factoring through equivalences (in order to merge components) and also to make sure that appropriate disjointness criteria are met and that the dangling edge condition holds. Proofs in this notation can get very long and hard to decipher. It is usually easier and more elegant to characterize graphs via the relations between them, i.e., via morphisms. For instance, one could define the disjoint union of two graphs using a so-called universal properties as the "most general graph" (in a sense which will be made precise below) into which both graphs can be mapped via morphisms. Such universal properties are studied in category theory [Mac71, Pie91].

Categorical approaches also give us the possibility to speak in general about "graph-like structures" and obtain results that do not only hold for specific classes of graphs—such as directed graphs or hypergraphs. This is important since the notion of "graph" is far from fixed, different variants of graphs arise in different application areas.

We will, in the following present the double-pushout (DPO) approach [EPS73]—one of the standard frameworks for graph rewriting—which is based on categorical notions and gives an elegant definition of graph rewriting. A different approach, which is also frequently used but which will not be considered in this thesis, is the so-called single-pushout (SPO) approach. For an overview of these approaches see [Roz97].

The DPO (double-pushout) approach is based on a specific form of colimit in category theory [Mac71, Pie91], the so-called pushout. In the category of graphs and graph morphisms pushouts formalize a very intuitive concept: gluing together two graphs by merging a common subgraph. We will define pushouts only for graphs and graph morphisms, although the definition below can be easily generalized to arbitrary categories.

Definition 2.4.1 (Pushout) *Let $\varphi_1\colon G_0 \to G_1$ and $\varphi_2\colon G_0 \to G_2$ be two graph morphisms. The* pushout *of φ_1 and φ_2 consists of a graph G_3 and two graph morphisms $\psi_1\colon G_1 \to G_3$, $\psi_2\colon G_2 \to G_3$ such that $\psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2$ and for every other pair of morphisms $\psi_1'\colon G_1 \to G_3'$, $\psi_2'\colon G_2 \to G_3'$ such that $\psi_1' \circ \varphi_1 = \psi_2' \circ \varphi_2$ there exists a unique morphism $\psi\colon G_3 \to G_3'$ such that $\psi \circ \psi_1 = \psi_1'$ and $\psi \circ \psi_2 = \psi_2'$.*

$$\begin{tikzcd}
G_0 \arrow[r, "\varphi_2"] \arrow[d, "\varphi_1"'] & G_2 \arrow[d, "\psi_2"'] \arrow[ddr, "\psi_2'"] & \\
G_1 \arrow[r, "\psi_1"] \arrow[drr, "\psi_1'"'] & G_3 \arrow[dr, dotted, "\psi"] & \\
 & & G_3'
\end{tikzcd}$$

The morphism ψ in the definition above is also called mediating morphism. Observe that we can obtain this morphism "for free" whenever we have a pushout and another commuting square, i.e., a square for which the compositions of morphisms coincide. The existence of mediating morphisms is a central concept that is used in many proofs.

Pushouts always exist (in the category of graphs and graph morphisms) and are uniquely defined up to isomorphism. Furthermore the pushout of two injective morphisms consists again of two injective morphisms. Pushouts can be explicitly constructed using the following gluing operation.

Proposition 2.4.2 (Explicit construction of pushouts) *Let $\varphi_1\colon G_0 \to G_1$ and $\varphi_2\colon G_0 \to G_2$ be two graph morphisms and construct a graph G_3 as the disjoint union of G_1 and G_2 factored through the relation $\equiv$ on nodes and edges of G_1 and G_2, which is defined as the smallest equivalence satisfying:*

- $\varphi_1(e) \equiv \varphi_2(e)$ *for every edge* $e \in E_{G_0}$ *and*
- $\varphi_1(v) \equiv \varphi_2(v)$ *for every node* $v \in V_{G_0}$.

Furthermore let $\psi_1\colon G_1 \to G_3$, $\psi_2\colon G_2 \to G_3$ be morphisms mapping every edge and node to its equivalence class. Then G_3, ψ_1 and ψ_2 are well-defined and satisfy the definition of a pushout.

Having formally defined the notion of "gluing", we can now define a rewriting step in the DPO approach.

Definition 2.4.3 (DPO rewriting step) *A* rule $(L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R)$ *consisting of two (not necessarily injective) graph morphisms can be applied to a graph G, resulting in a graph H, if there is a match morphism $\varphi\colon L \to G$ and we can find a graph C and morphisms such that the two squares in the following diagram are both pushouts.*

$$\begin{tikzcd}
L \arrow[d, "\varphi"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[d] \\
G & C \arrow[l] \arrow[r] & H
\end{tikzcd}$$

The intuition behind this definition is the following: We first have to remove the left-hand side (apart from its interface) from the graph G, which is done by locating a graph C such that gluing L and C along the interfaces results in G. Then we glue together C and R resulting in the graph H. This is schematically depicted in Figure 2.7.

The graph C is usually called *pushout complement* and it might not always exist. Specifically if the removal of L would leave some edges without attached nodes (see also the dangling edge condition in Definition 2.3.2), there is no way to complete the diagram in order to obtain two pushouts.

Note also that if we start with three injective morphisms $\varphi_L, \varphi_R, \varphi$, the diagram can be completed using only injective morphisms. In this case we have exactly the situation described in Definition 2.3.2.

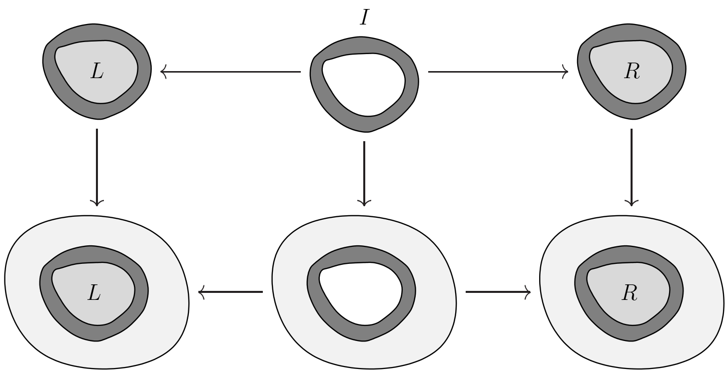


Figure 2.7: Schematic representation of graph rewriting in the DPO approach.

As mentioned earlier, the DPO approach can be used not only to model rewriting on graphs, but also on many other graph-like objects. There is a family of categories—adhesive categories [LS04]—that capture exactly the notion of structures for which it is possible to cut substructures out and to attach new structure. This general view on graph rewriting is also known as high-level replacement [EGPP99, EHPP04].

Coming back to the running example, we can depict a rewriting step corresponding to the application of rule [AcquireResource$_{1,1}$] in Figure 2.8. Again numbered nodes indicate how morphisms map nodes into the graphs.

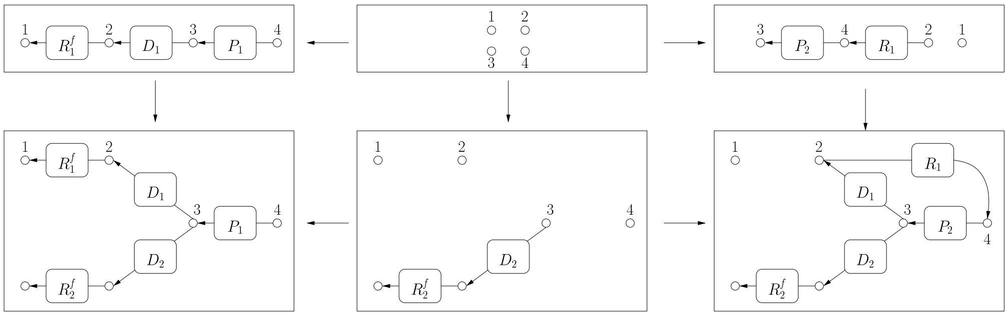


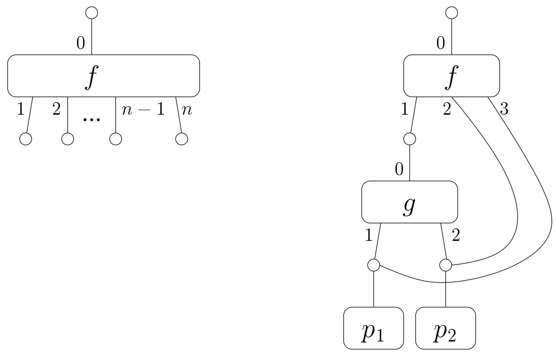
Figure 2.8: A DPO rewriting step in system Sys.

2.5 Modelling Dynamically Evolving Structures—Some Recipes

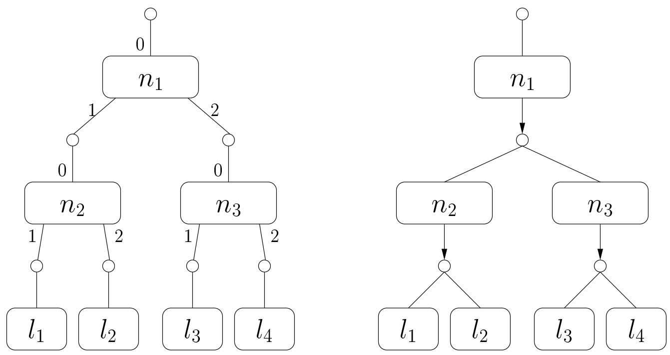
In the following we give some recipes describing how to model frequently occurring computational structures by graph rewriting. The list is by no means exhaustive, neither are these suggestions original. Its purpose is to give a general impression about how to model dynamic evolution using graphs and graph transformation. The presented encodings can be easily automated.

Functions: When modelling functions and functional programming, a function with n parameters can be depicted by an $n+1$-ary hyperedge with one node for its result and the remaining n nodes in order to attach the parameters. Using these edges one can either construct tree-like terms or—alternatively—acyclic graphs modelling the sharing of common subterms. The latter technique is used in efficient implementations for functional programming languages.

In this case, the right-hand graph stands for the term $f(g(p_1, p_2), p_2, p_1)$.



Ordered trees: In the same vein as above, one can build ordered trees. It is equally easy to model trees without any order on the children of a parent node using binary edges.

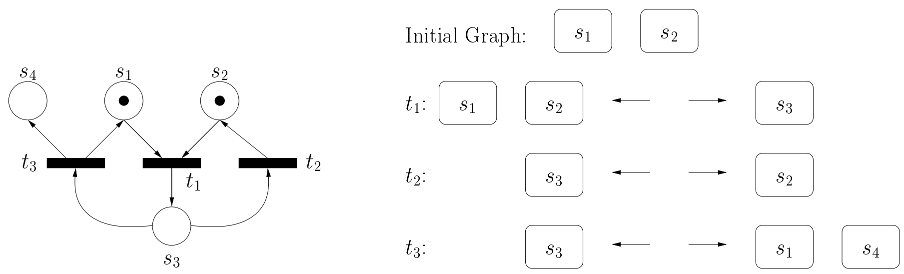


Processes: Hyperedges are especially convenient for the modelling of processes, where a process has a number of ports (= nodes) in order to interact with its environment. If we compare to process calculi such as the π-calculus [Mil99, SW01], nodes roughly correspond to the (channel) names of a process. There are several encodings of the π-calculus into graph rewriting based on that idea [MP95, Kön00c]. Although the term is not standard, one could describe these encodings as *hierarchical* encodings: They reflect the term structure of processes in the boundaries of hyperedges. Encoding a specific process gives us a graph representing this process and a set of rules describing its behaviour.

On the other hand there are a number of encodings [GM01, GM02] of a different type, that *flatten* the term structure and encode one process into a flat graph consisting mainly of the syntax tree of the process together with connections representing channels or shared ports. The set of reaction rules is finite and fixed. Encodings of λ-terms into acyclic graphs with sharing (see [Wad71, BvEG+87]) are also of this type. Although the fixed set of rules is a strong point in favour of such encodings, they usually have problems modelling replication and recursion. Furthermore they usually require rules merging nodes, leading to non-injective morphisms in graph rewriting rules, which might complicate the analysis of such systems.

Petri nets: Graph transformation systems are in some sense a generalization of Petri nets. While Petri nets describe rewriting on multi-sets, graph transformation systems model rewriting on more complex structures. There is a very simple and straightforward encoding of Petri nets into graph transformation systems: A token on place s is represented by a 0-ary hyperedge labelled s, while a transition t becomes a rewriting rule.

The following figure shows an example for such an encoding.



Control states: There are some approaches to add to graph transformation systems features especially useful for programming [Sch96, HP01]. Some features such as a control or program state can be easily added. For instance we might want to add a set of control states $\{\ell_1, \ldots, \ell_n\}$ and require that rules can be only applied in certain control states. This can be modelled by adding a 0-ary edge representing the control state to the initial graph and modifying all rules in such a way that they are only applicable if a suitable "control edge" is present. Furthermore they must replace the old control state with the new control state.

There are also more sophisticated approaches to programming with graph transformation systems, see for instance PROGRES [Sch96].

Operations on pointers: Verifying programs manipulating pointer structures on the heap is an important application area for our techniques. Pointer structures can naturally be considered as graphs, while program statements can be interpreted as graph transformation rules. Rather than giving a general encoding—which is possible—we will show with an example how to encode such programs. Similar examples can be found in [RV04]. Comparable problems arise when object-oriented languages are encoded into graph transformation systems [CDFR04, WG96].

We consider the following program deleting the last element of a linked list. The variable `start` points to the first element of the list, the remaining elements can be reached using the `next` selector. The end of the list is reached if we find the null pointer. For this program we assume that the list contains at least two elements.

```
1: x:=start;
2: y:=start->next;
3: while y->next≠null do
4:    x:=y;
5:    y:=y->next
   od;
6: x->next:=null
```

Figure 2.9 shows such a linked list, modelled as a hypergraph, where cells are generic elements of the list. The rules in Figure 2.10 specify the deletion of its last element according the program given above. Note that here we use 0-ary hyperedges indicating program states, as suggested above. Conditional branches in the program are modelled using rules that check whether a specific subgraph is present without deleting or creating anything. Some parts of the graph might get disconnected, for instance the last cell which is being deleted from the list. These disconnected parts can be considered as garbage, to be removed by a garbage collector.

For more convenience it might sometimes also be necessary to use rules with negative application conditions, i.e., additional conditions that say that a rule can only be applied whenever certain substructures are *not* present. These negative application conditions do not integrate well with most of the techniques presented in this thesis, their integration is an interesting and important direction of future research.

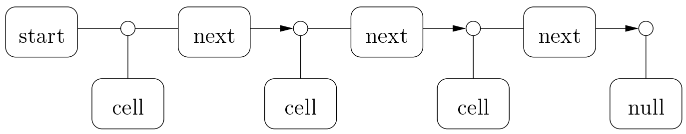


Figure 2.9: A linked list.

Graph transformation rules can easily model local transformations on graphs. It is possible in principle to work with graph transformation frameworks allowing non-local transformations using complex embedding rules, but these are not considered here. First, while at the most abstract level of computation, graph rewriting steps might take a non-local form, system models closer to an implementation will always be describable by local steps. Second, it seems highly unlikely to obtain satisfactory analysis techniques using such complex transformation rules.

There is one issue we have neglected so far and that is the integration of data values. Although the formalisms under consideration are Turing-powerful, for

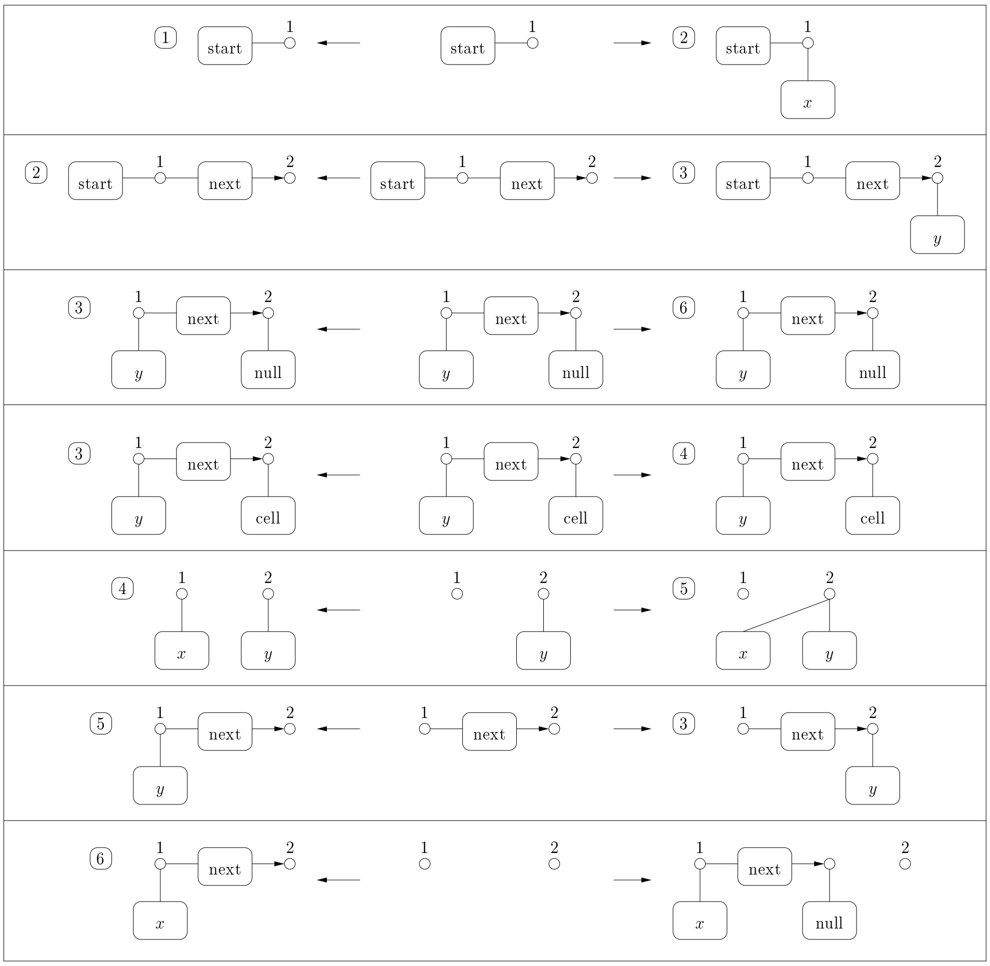


Figure 2.10: Rules describing the deletion of the last element of a linked list.

convenient modelling and the encoding of real-world programming languages, data values, such as integers and strings attached to nodes and edges, are usually needed. These data values attached to graph components are also called attributes and they can be useful for modelling counters, message contents, and other information. There is some work on attributed graph transformation systems [EPT04, BFK00, LKW93], especially in the framework of high-level graph rewriting [EHPP04, EGPP99]. Here, however, we will not consider attribution; the analysis task is sufficiently interesting and complex without them. Furthermore verification can usually take place at a more conceptual level where concrete attribute values are ignored. Finally, it seems fairly straightforward to integrate attributes into most of the approaches presented here, although it will notably complicate the notation. If we want to approximate infinitely many reachable graphs by some finite means, it will usually be unavoidable to abstract data values having an infinite domain. Such abstraction techniques are well-studied in the theory of abstract interpretation [CC77, Cou96, NNH99].

Chapter 3

A Short Summary of Methods and Techniques in Static Analysis and Verification

In this chapter we will summarize established analysis and verification techniques with an emphasis on methods that are relevant in our setting. In subsequent chapters we will then present adaptations of these techniques to the verification of dynamically evolving graphs.

Let us first clarify how the terms "verification" and "(static) analysis" are used. The exact meaning of this terms varies, but there seems to be a tendency to use them in the following way: The term "verification" is used for techniques that are complete in the sense that they can either provide a positive answer ("Yes, the property holds!") or a negative answer ("No, the property does not hold."). If we work with Turing-complete models, non-trivial verification problems are always undecidable, which leads to problems if we want to mechanize and automate proofs. In this case some form of interaction with the user is required. The term "(static) analysis" on the other hand refers to techniques which compute some form of approximation or abstraction of a system, containing all its runs, possibly together with additional, so called spurious, runs that have no counterpart in reality. In this case the analysis tool may give answers of the form "I don't know." if asked whether a certain property holds for a given system.

We will in the following first give a general summary of these techniques and then treat in more detail the analysis and verification of dynamic graph-like structures.

3.1 Verification Techniques

In the context of this thesis we are mainly interested in the following two verification problems: the *equivalence problem* (see Chapter 4) and the *model checking problem* (see Chapter 6).

The equivalence problem can be stated as follows: We are working in some formalism that allows us to specify systems and that furthermore has some notion of behavioural equivalence, i.e., it is defined when two systems should be considered equivalent from the point of view of behaviour. This is a widespread notion in the theory of reactive systems, and especially in process algebra [BPS01]. In this framework actions of processes, such as for instance communication actions, are visible to the environment and two processes are considered to be equivalent whenever they can not be distinguished by an external observer. Depending on the power that is given to the observer, this leads to different notions of behavioural equivalence: strong and weak bisimilarity, trace and failures preorders, or testing equivalences.

We are then provided with a system description *Sys* and a (usually much simpler) specification *Spec* of the system. The question to answer is then whether the system and its specification are behaviourally equivalent (sometimes written $Sys \sim Spec$). This technique is applicable in cases where an internally complex system has a fairly simple observable behaviour. This is true whenever the implementation is substantially more complex either for reasons of efficiency or of distribution. Especially the latter case applies to reactive systems: The behaviour of the system observed by the environment may be easily describable, but its internal structure can be much more complex, due to the fact that the functionality of the system has to be distributed and hence performed by several processes simultaneously.

It is usually required that behavioural equivalences are also congruences, i.e., closed under the operators of the calculus. This makes sure that replacing a subsystem by an equivalent one does not change the overall system behaviour.

Sometimes it is not feasible to give a full specification of the system; this is true for instance in the case of mutual exclusion protocols, where the specification of the system has more or less the same size than the system itself. In these cases and in cases where we are only interested in specific systems properties we consider the model checking problem [CGP00], which is to determine whether a system *Sys* satisfies a property φ, usually written $Sys \models \varphi$. In the case of reactive systems, this property is usually a formula in a temporal logic, specifying the future evolution of a system. Typically such properties are reachability ("There is a reachable state which satisfies ..."), safety properties and liveness properties [AS85, Pra94].

Temporal logics can be grouped into linear time logics, such as LTL, specifying properties for all possible paths in the transition graph of a system, and branching time logics, such as CTL (= computation tree logic), specifying properties for each system state and the potential branching behaviour of the system starting from that system state. Even more general temporal logics uniting the linear time as well as the branching time view are CTL* and the modal μ-calculus [CGP00].

The model checking of finite-state systems is a well-developed research area, which has produced powerful tools such as SPIN [Hol97] and SMV [McM98]. Model checking has been especially successful in hardware verification. Software, however, usually requires the verification of infinite-state transition systems, making necessary either the integration of approximation techniques (see also Section 3.2) or the generalization of model checking techniques to infinite-state systems [Esp97].

3.2 Static Analysis Techniques

Static analysis has its origins in *data flow analysis* for compiler optimizations, but it soon became clear that software verification is another important application area, as witnessed by early seminal papers on abstract interpretation [Cou96, CC79, CC77]. *Abstract interpretation* (see Chapter 6) is a technique in which a program is executed on abstract data. For instance, instead of computing with integers, one performs operations on abstract values such as odd and even. More generally, abstract interpretation can be seen as a way to over-approximate a system by a simpler system being more amenable to analysis. Abstract interpretation can be conveniently combined with model checking [CGL99] where first an over-approximation of the system is computed which is subsequently verified by a model checker. An important point to notice here is that in the temporal logics existential quantification over the runs of a system ("There exist a run such that ...") usually has to be disallowed since runs satisfying such a formula might be spurious, introduced by the approximation.[1] Whenever a formula is found to be true in model checking, the restrictions on the set of formulas ensure that it is also true for the original system. If, however, the formula does not hold on the approximation, we can not directly draw any conclusions. If a counterexample is provided, we could check whether it is spurious and, if so, refine the abstraction in such a way that the counterexample disappears, a method which is called counterexample-guided abstraction refinement [CGJ$^+$00].

A quite different analysis technique can be obtained by using *type systems* (see Chapter 5). Type systems in our setting will go beyond assigning types such as `integer` or `boolean` to program variables. We adopt a more recent view of types, where types give an abstract representation of system invariants or future behaviour.

Apart from well-known type systems for imperative programming languages and slightly more complex type systems for functional languages such as ML, quite complex type systems have been developed for process calculi, checking properties such as input/output-behaviour [PS96, KPT99], absence of deadlocks and livelocks [Kob98, Kob02], or security properties [Aba99, BDNN98, HVY00, HR00].

The type systems we are interested in here are Curry-style type systems where the program is given without type annotations and there are typing rules describing how to derive valid types. Usually such a type system is also equipped with a type inference algorithm computing the most general or principal type of a program. The fact that a program can be typed and its type can give us valid information about its behaviour. This may include the absence of runtime

[1]This restriction can be lifted by exploiting both under- and over-approximations or using so-called mixed abstractions [DGG97, Kel95].

errors, but could potentially give us much more specific information, such as for instance its input/output behaviour. Compared to other analysis methods mentioned earlier in this chapter, type systems are a modular analysis method, since the type of a system can be derived inductively by considering the types of its subsystems.

3.3 Verification and Analysis of Dynamically Evolving Structures

Concerning the analysis of dynamically evolving graph-like structures we can distinguish two main lines of research: First, methods that are derived from classical program analysis frameworks such as data flow analysis and that are concerned with the analysis of pointer structures in imperative programs. Some, even imprecise, analyses of this form are always required in the analysis of C or Java programs. Fairly simple analysis methods based on data flow or control flow analysis are the *alias analysis*, which identifies variables pointing to the same location, and the *points-to analysis*, providing estimates of pointer values. There are two different ideas on which pointer analysis may be based: either one assumes a fixed bound k on the length for access paths (so-called k-limiting) and merges all memory locations that can only be reached in more than k steps, or one merges all elements of the same type or class (such as lists, trees, arrays, etc.). Furthermore, in order to obtain good analysis results it is important to perform a context-sensitive analysis, where the analysis result for a procedure depends on the calling context. An overview over pointer analyses can be found in [Wil97].

More general methods, also based on data flow analysis, for finding over-approximations of pointer structures are known as *shape analysis* techniques. One successful idea is to represent these over-approximations as models of a 3-valued logic [SRW02], where the values of the logic state whether a pointer is present (value 1), not present (value 0) or possibly present (value 1/2). Modifications on pointer structures have the following effect on the approximation: First the part of the approximation where the modification should take place is brought into focus by removing the indefinite value 1/2, then the modification is applied and finally the resulting structure is coerced, i.e., brought into normal form. The technique is not fully automatic, in order to obtain good results, predicates and predicate transformers have to be defined manually. On the other hand, the method can derive fairly accurate information regarding reachability and acyclicity in data structures. Shape analysis has also been applied to other areas such as process calculi [NNS00].

A second, much more recent, line of research is more general and is concerned with the analysis of graph transformation systems. There has been comparatively little work on verification and analysis methods so far. This is probably caused by the fact that the analysis of graph transformation systems is necessarily quite complex and research in the area of graph grammars and graph transformation systems so far had a strong focus on semantic issues such as rewriting formalisms, expressiveness, concurrency and applications such as software engineering.

While some research groups [Var02, DFRS03] pursue the idea of *translating graph transformation systems into the input language of a model checker*, others attempt to develop *new specialized methods* for graph rewriting. Work from our side goes in this latter direction, as well as [Ren04a, Ren03, Ren04b]. Although it is tempting to use existing optimized model checking tools, there is good reason for developing new techniques, even for finite state spaces. Existing tools usually do not directly support the creation (and deletion) of an arbitrary number of objects while still maintaining a finite state space, making entirely non-trivial their use for checking finite-state GTSs. Similar problems arise for process calculi agents with name creation, which has also led to specialized techniques in this area such as HD-automata [Pis99]. A nice overview comparing these two fundamentally different approaches can be found in [RV04].

It is important to mention that apart from our approach and from some results in [Ren04a, Ren04b], which describe static analysis techniques for infinite-state systems, all methods mentioned earlier are verification techniques for finite-state systems.

Early work and ideas on the verification of graph transformation systems and specific temporal logics can be found in [Koc00, GHK98]. In these papers temporal logics is enriched by graphical constraints, suitable for specifying the behaviour of graph transformation systems. In [Hec98] a compositional method for verifying graph transformation systems is presented. These more foundational approaches do not lend themselves to easy mechanization.

Chapter 4

Behavioural Equivalences for Graphs

In this chapter we will address the following questions: What is a suitable notion of behavioural equivalence for graphs and graph transformation systems, i.e., when can two graphs be considered equivalent? Is this notion of equivalence also a congruence, i.e., is it preserved whenever a graph is embedded into a larger context? And finally, what are suitable proof techniques that help us to make equivalence proofs easier?

4.1 Basic Definitions

The behavioural equivalence we will focus on in this chapter is bisimilarity, a very natural notion that lends itself to mechanization and efficient proof techniques to a greater extent than other equivalences, such as trace or failures equivalence. Bisimilarity can be defined in a coinductive way in a very general setting, entirely independent of graph transformation systems.

Definition 4.1.1 (Bisimulation and Bisimilarity) *Let $\rightarrow \subseteq Q \times \Lambda \times Q$ be a transition relation with labels from a set Λ over a set of states Q. We write $q_1 \stackrel{\ell}{\rightarrow} q_2$ whenever (q_1, ℓ, q_2) is contained in $\rightarrow$.*

A symmetric relation $\mathcal{R} \subseteq Q \times Q$ is called bisimulation *whenever the following condition holds:*

> *Whenever $(q_1, q_2) \in \mathcal{R}$ and $q_1 \stackrel{\ell}{\rightarrow} q_1'$, then there exists a state q_2' such that $q_2 \stackrel{\ell}{\rightarrow} q_2'$ and $(q_1', q_2') \in \mathcal{R}$.*

Two states q_1, q_2 are called bisimilar *if there exists a bisimulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. This is written $q_1 \sim q_2$ and the relation $\sim$ is called* bisimilarity.

Alternatively one can also define the notion of bisimilarity as a game between two players A and B. Player A attempts to show that two states q_1 and q_2 are not bisimilar, whereas player B wants to show that they are bisimilar. Player A chooses either q_1 or q_2 and makes a transition $q_i \stackrel{\ell}{\rightarrow} q_i'$ which B should mimic doing $q_{3-i} \xrightarrow{\ell} q'_{3-i}$. This gives us a new pair (q'_1, q'_2) and the previous step is repeated. Player B wins if he can keep the game going for an infinitely long time and loses when it can not mimic a move of A. The states q_1, q_2 are bisimilar if and only if B has a winning strategy.

Bisimulations have been studied in concurrency theory for some time [Par81]. They have been used for proving the correctness of protocols [LM86, RE99] and there are several tools available that support and mechanize bisimilarity proofs, such as the Edinburgh Concurrency Workbench[1], HAL[2] (a tool which is based on HD-automata [Pis99]), tools using up-to techniques for bisimulation verification [Hir99] and of course—more generally—also theorem provers such as PVS, Isabelle or Coq.

For graph transformation systems, bisimulation theory has only been developed recently [BMS00, BCM02, KM01, EK04a, SS04].

4.2 Bisimilarity is a Congruence—A History of Research

The states of a system are usually not atomic objects, but terms or—in our setting—graphs. It is thus desirable to make sure that bisimilarity is a congruence, i.e., it is preserved by the available operators, the most important of which is usually parallel composition. Congruence is a very desirable property since it allows to replace a subsystem with an equivalent one without changing the behaviour of the overall system and furthermore helps to make bisimilarity proofs modular.

Bisimilarity naively defined on top of a given labelled transition system is usually not a congruence. A simple example is the CCS reduction rule $a.P \mid \bar{a}.Q \xrightarrow{\tau} P \mid Q$ describing two parallel processes $a.P$ and $\bar{a}.Q$ synchronizing on an action a. If we consider only this rule and denote the resulting bisimilarity by $\sim$, then clearly $a.P \sim b.P$, since both are inactive processes, but $a.P \mid \bar{a}.Q \not\sim b.P \mid \bar{a}.Q$, since one process can reduce, while the other can not.

There are several methods to ensure that bisimilarity is indeed a congruence.

Manual proofs: The most straightforward method is certainly to show manually that bisimilarity is a congruence, as it was done for process calculi CCS [Mil80] and the π-calculus [Mil99, SW01]. This approach can be very tedious, specifically it can be very hard to find the appropriate labels.

Rule formats: A different approach uses rule formats such as the de Simone format [dS85] and the `tyft/tyxt`-format [GV92]. If the rules describing a labelled transition system agree with these rule formats, then it follows automatically that bisimilarity is indeed a congruence. Usually these rule formats disallow the use of parallel composition on the left-hand side of a rule and demand that interaction is described locally from the

[1] `http://www.dcs.ed.ac.uk/home/cwb/`
[2] `http://fmt.isti.cnr.it/hal/HAL/`

point of view of a single component, while moving all the information on interaction with the external environment into the labels.

Deriving labels: A more recent idea that originated in [Sew02] and [LM00] is to start from an unlabelled transition systems describing internal reaction of the system. From this unlabelled transition system a labelled one is automatically generated in such a way that bisimilarity is a congruence. The basic idea is to use as labels the minimal contexts that a process needs to react.

Interestingly research on this topic, which is still ongoing, shifted its focus from process calculi to rewriting of graph-like structures such as bigraphs [Jen, JM03, SS04], since the derivation of labels works more smoothly in these settings.

This thesis contains contributions to the second approach (see Section 4.2 and Chapter 9) and the third approach (see Section 4.3 and Chapter 8), in both cases for graph transformation systems. In the following we will mainly give some intuition about the third approach and describe its relationship to the second approach.

4.3 Deriving Labels in the DPO Approach to Graph Rewriting

In order to define a notion of equivalence on graphs that takes into account (future) interaction with its environment or context, we have to formally define the following concepts:

- *What does it mean to compose a graph with a context?* Gluing of graphs is naturally defined by pushouts (see Definition 2.4.1), but a notion that is not so common in graph rewriting is to inductively compose graphs out of smaller subgraphs (see for instance [GH97, Kön02]). This notion requires that graphs are equipped with interfaces.

- *What are the transition labels describing interaction with the environment?* From other work on this topic it is clear that labels should be contexts which allow a graph to rewrite. Since contexts are basically graphs, labels in this setting will also be graphs. Furthermore in order not to get an overwhelmingly large (or even infinite) number of labels, we require that the contexts should be minimal in the sense that they only provide the necessary items needed to complete a partial left-hand side.

This minimality requirement for labels can be illustrated for graphs in the following way (see Figure 4.1). At first we see the schematic representation of a graph G with interface, i.e., with specific nodes and edges that can be composed with a context. In this graph G we can find a partial match of a left-hand side L, overlapping with G. If we only take the new, previously non-existing parts of L and the interface of G (in order to be able to compose), we obtain a label

F as the minimal context, also called borrowed context, which G needs in order to complete its partial match. The resulting labelled transition is depicted in the lower half of Figure 4.1.

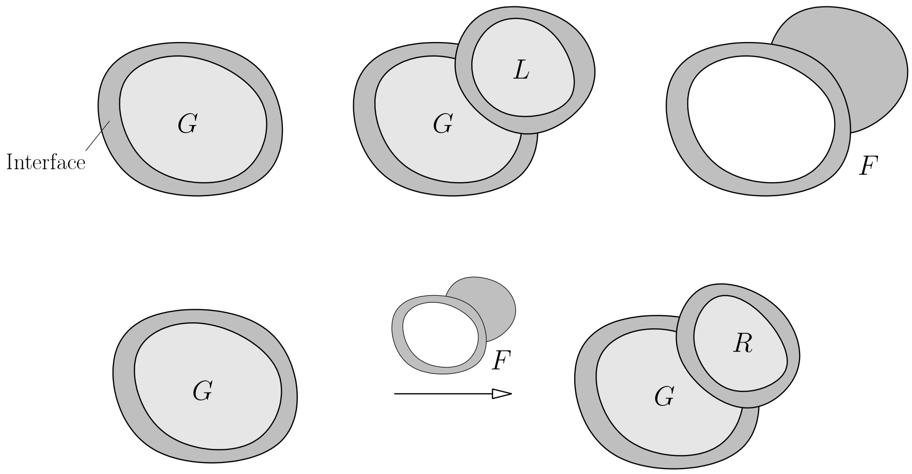
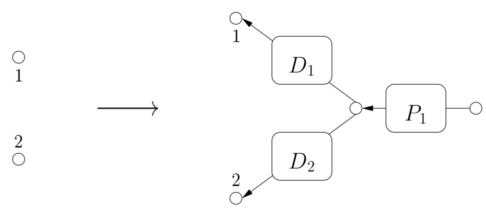


Figure 4.1: Deriving labels for graph transitions (schematic representation).

Formally, a graph with interface is an injective graph morphism $J \to G$, where J is the interface. Although this is not indicated in Figure 4.1, contexts have an inner and an outer interface in order to ensure that composing a graph with interface with a contexts results again in a graph with interface. So, a graph context is a pair of two injective graph morphisms $J \to F \leftarrow K$, where J is the inner and K is the outer interface. Composing a graph with interface $J \to G$ with a context $J \to F \leftarrow K$ gives a graph with interface $K \to \overline{G}$, which is obtained by the following pushout construction.

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

Using again pushouts and also pullbacks, a related categorical concept, we can describe how to obtain the minimal context F. A more detailed description can be found in Chapter 8. Here we will illustrate the derivation of a context using an example. We consider the rules of system Sys introduced in Section 2.3 (see Figure 2.5) and the following graph with interface $J \to G$.



This graph contains a total match of rule [CreateNewProcess] and a partial match of rule [AcquireResource$_{1,1}$]. Figure 4.2 shows how the partial left-hand side of rule [AcquireResource$_{1,1}$] can be completed to a full left-hand side L by attaching the shaded edge labelled R_1^f. This figure also indicates the borrowed context F that consists of the new components and the interface J of G.

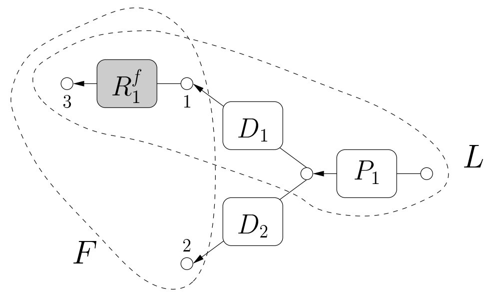


Figure 4.2: Completing a left-hand side.

Note that the graph G does *not* contain a non-trivial partial match of rule [AcquireResource$_{2,2}$] although both G and the left-hand side of this rule contain an edge labelled D_2. But there is no way to attach a context to the external nodes such that the full left-hand side is present. Naturally, there is the possibility of adding a disjoint copy of every left-hand side to G, but that is trivial partial match that need not be considered.

In the following we consider the partial match of rule [AcquireResource$_{1,1}$] depicted in Figure 4.2. So the borrowed context should contain this edge, including its inner interface, which coincides with J, and an outer interface containing nodes 1 and 2 and the new node 3 which has been freshly created. The derived graph context $J \to F \leftarrow K$ is shown in Figure 4.3. Observe that attaching $J \to F \leftarrow K$ to $J \to G$ gives us the graph depicted in Figure 4.2 with nodes 1, 2, 3 as the interface. By rewriting this graph using rule [AcquireResource$_{1,1}$] we obtain a graph H. This graph H with its interface K is also depicted in Figure 4.3. Intuitively this rewriting step says: "Whenever we attach an available resource R_1^f to a process requiring two resources, then the process can proceed by acquiring this resource." It is denoted by

$$(J \to G) \overset{J \to F \leftarrow K}{\longrightarrow} (K \to H).$$

The main result of Chapter 8 says that bisimilarity defined on top of these labelled transitions is a congruence relation. Put differently, this means that a subsystem can always be replaced by a bisimilar one, without changing the behaviour of the entire system. In Chapter 8 a simple example of this kind is presented, where we show that a simplex connection in two directions can always be replaced by a duplex connection.

Independently of our work, another way of deriving labels for graph transition systems has been presented in [SS04]. This approach describes minimality using a universal property, formally defined by so-called relative pushouts. It has turned out that both approaches generate the exact same labels. While the approach in [SS04] has the advantage that the universal property can be used in order to show the congruence result also for other equivalences such as trace and failures equivalences, our approach is much simpler and describes a

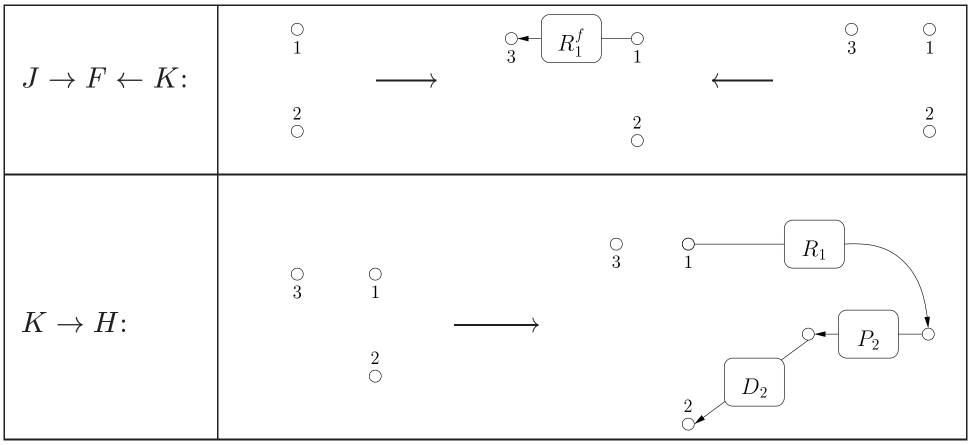


Figure 4.3: Graphs belonging to a labelled transition step.

concrete and straightforward way for the derivation of labels that is well-suited for mechanization.

We can now come back to our example and give a specification of system Sys (see Figure 4.4). The specification is more abstract than the system description itself since instead of representing processes and their resources structurally, it just stores their number. So, an edge labelled $\mathit{Spec}(p, r_1, r_2)$ stands for a subsystem consisting of p processes, r_1 of which are in possession of resource R_1, and r_2 of which are in possession of the resources R_1 and R_2.

Using the proof techniques presented in Chapter 8 we can show that the two graphs in Figure 4.5 are bisimilar. Both are equipped with a discrete interface consisting of two nodes, numbered 1 and 2.

This gives us additional certainty that the system indeed behaves as expected. Furthermore we also gain knowledge about how such a subgraph behaves when put into a context where unlimited resources are available, which is different from the situation of the start graph depicted in Figure 2.4. Since a process can multiply it can consume more than two resources, whenever they are available.

Furthermore, if we attach two edges labelled R_1^f, R_2^f to an edge labelled $\mathit{Spec}(1, 0, 0)$ and assume an empty interface (see Figure 4.6), we can easily show that the resulting graph transition system is bisimilar to a transition system consisting of one state and a loop only, i.e., it has arbitrarily long runs and never terminates. Because of the congruence property this means that also the start graph depicted in Figure 2.4 is equivalent to such a transition system and this shows that we will never reach a deadlock situation in which the entire system is blocked.

See Chapter 8 for more details and a different example.

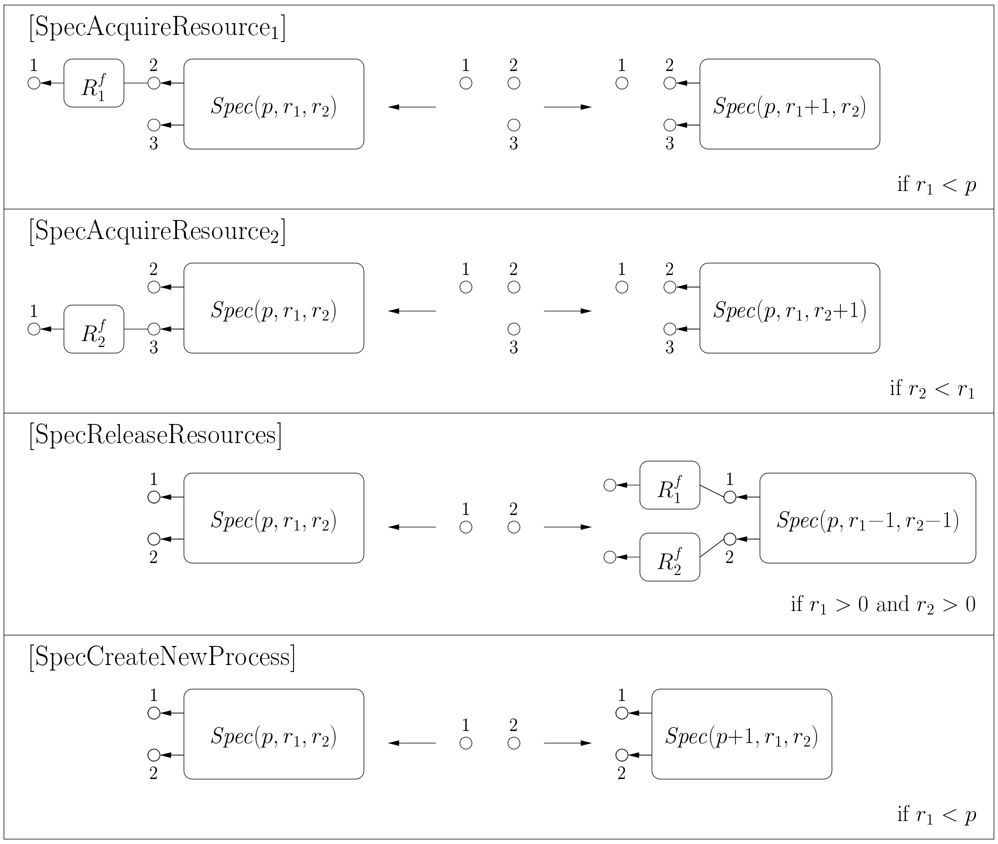


Figure 4.4: A specification of system Sys.

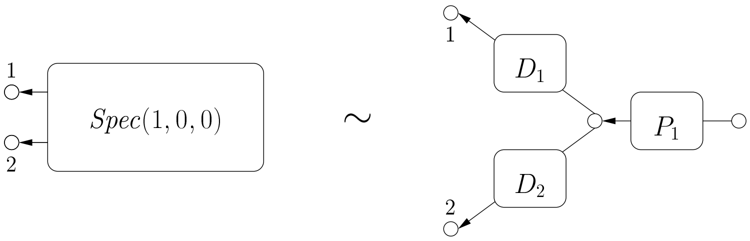


Figure 4.5: Two bisimilar graphs.

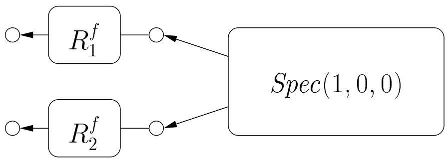


Figure 4.6: Attaching two resources to the specification.

4.4 Bisimilarity for Synchronized Graph Rewriting with Mobility

Different from research presented in the previous section, the work presented in this section is not concerned with the derivation of labels from unlabelled reduction rules, but is related to work on specific rule formats ensuring that bisimilarity is a congruence.

We start with a labelled transition system for graphs, where labels are not graphs, but actions, similar to the setting of process calculi. Furthermore, the left-hand side of a rule consists of one hyperedge only, a restriction necessary in this setting in order to obtain the congruence property. Hyperedges agree on actions on their shared nodes.

For giving the graph transformation system sufficient power in order to model meaningful systems, the following two features are introduced:

- *Synchronization:* Parts of a graph can be rewritten in parallel, provided that the hyperedges synchronize in a consistent fashion on their shared nodes. In principle, the entire graph can participate in such a rewriting step.

  The form of synchronization used where processes agree on one and the same action a is called *Hoare synchronization*. This is different from *Milner synchronization*, where an action a is matched with a coaction $\bar{a}$.

- *Mobility:* In a rewriting step not only single edges are replaced by right-hand sides, but new shared nodes can be created as well.

This framework is called *synchronized hyperedge replacement* [HM01] and its origins go back to [DM87]. It provides us with a powerful formalism that can be specified using the so-called tile logic [GM02]. Results from the theory of tile logic then imply that bisimilarity is in fact a congruence relation, whenever left-hand sides consist of single hyperedges only. Congruence in this setting is meant with respect to a set of operators on graphs having a power comparable to graph contexts.

More details are presented in Section 9.

4.5 Evaluation

The work described in Section 4.4 is strongly inspired by notions from the theory of process calculi, including the use of actions as labels. It uses a graph transformation model that is closer to process algebra than, for instance the DPO approach. This was one of the first papers to address the issue of bisimulation congruences in the setting of graph transformation systems.

The (more recent) research presented in Section 4.3 seems to be very natural in the context of graph rewriting. Furthermore it fits exactly into a line of research [LM00, SS03] which originated from the area of process calculi and which is concerned with the derivation of labelled transitions and bisimulation congruences. Hence it is a further step in recent efforts towards a closer link between process algebra and graph transformation.

The work described in Section 4.3 can be recast in the framework of adhesive categories [LS04], which—intuitively—describe structures that can be rewritten by cutting substructures away and gluing structures together. Working on this high level of generality has the advantage that we can not only present the theory for specific graph rewriting frameworks, such as directed graphs or hypergraphs, but in general for all graph-like structures. This also shows that the theory is stable and does not have to be altered when different graph models are used.

While the main focus of the work presented in this chapter is on the semantic side, other work [LM86, RE99] has shown that bisimilarity and other behavioural equivalences can be used successfully for verification. The notion of bisimilarity is very general and proof techniques are available which help to simplify and mechanize bisimulation proofs [MS92a, Hir99, Hir98]. One area where behavioural equivalences are very useful for verification is the analysis of cryptographic protocols. Properties such as secrecy and authentication that are hard to grasp otherwise can be defined and verified using behavioural equivalences [FGM00, AG97, BBN04].

Sometimes this approach can not be used since full specifications are hard to obtain for some kinds of systems. Also, the automatic derivation of bisimulation proofs is no easy task, which means that the use of behavioural equivalences for verification and analysis has to be complemented by other techniques, which are more straightforwardly mechanizable and implementable. In the next two chapters we will introduce two such techniques: type systems and abstraction.

Chapter 5

Assigning Types to Graphs

In the following we will address the following questions: What is the type of a graph? How should a type systems for graph rewriting look like? Which properties can be checked using such a type system? How can we do type inference in a modular way?

Type systems for programming languages have been used for a long time in order to ensure well-typed programs and to avoid runtime errors. For imperative programming languages, usually fairly simple types such as `int`, `real` or `bool` are used. Type systems become more complex for functional languages such as the λ-calculus [Bar90, DM82] and ML [MCP93] and for object-oriented programming with inheritance and subtyping.

Even more complex type systems have been introduced for process calculi, checking properties such as input/output behaviour [PS96, KPT99], absence of deadlocks and livelocks [Kob98, Kob02], security properties [Aba99, BDNN98, HVY00, HR00], allocation of permissions to names [RH97] and many others. There are also interesting type systems for higher-order variants of the π-calculus [YH02]. In this way type systems have evolved from a technique used to avoid runtime errors to a more general method which can be used for verification and analysis.

Seen more abstractly, a type system is an inference system, allowing to derive judgements of the form $p \triangleright t$, where p is a program and t is a type, inductively over the structure of the program. The notation $p \triangleright t$ is read as "p has type t" or "type t can be assigned to p".

Type systems have the advantage of being a modular analysis method, i.e., the type of a system can be derived from the types of its subsystems. Furthermore, type inference algorithms are usually quite efficient, compared to other verification methods. Naturally this also means that type systems approximate, sometimes quite strongly. This means that there will be many graphs satisfying a certain property that can not be typed. But there are still many interesting properties for the checking of which type systems are quite appropriate. They can especially be used for fast debugging and in order to complement other more involved techniques. There are also type-based tools such as TyPiCal[1] for analyzing processes.

[1] `http://www.kb.cs.titech.ac.jp/~kobayasi/typical/`

We will present a general framework for such type systems, also called generic type system, which attempts to play a similar role than the monotone frameworks in data flow analysis that can be instantiated in order to perform various analyses [NNH99]. Note that the main difficulty here lies in determining an elegant and not overly complex way of integrating various different type systems for different languages. Among others, type systems for the π-calculus (see Chapter 10) and the λ-calculus can be integrated into this framework.

5.1 What is the Type of a Graph?

The first question that we have to answer is the following: What is the type of a graph? In order to answer this question it is important to review an important concept from graph rewriting: so-called type graphs [CMR96]. A type graph TG represents a set of graphs, namely all graphs which can be mapped to TG via morphisms.

A concrete practical example of a type graph is a UML class diagram with added relations (inheritance, associations, dependencies, etc.). Then a state of the system consisting of objects and their relations should be mappable into the class diagram via a graph morphism, mapping each object to its corresponding class. In this sense the class diagram represents all these system states.

In typed graph rewriting a type graph TG is given a priori and all rules have to be typed over this graph, i.e., left-hand side, interface and right-hand side have to be mapped into TG such that the resulting diagram commutes. Then, if rules are applied according to their typing, we can show that every reachable graph can again be typed.

Let us emphasize that this notion of type is quite different from what will be presented in this chapter. In the existing work on type graphs a type graph is fixed a priori and all transformations have to adhere to it. This is similar to a Church-style type system, where programs with type annotations are given and it remains to check whether these annotations are valid. If one wants to use type system for analysis purposes however, one is rather interested in having a Curry-style type system where a system without type annotations is given, and the most general type of a given graph is derived using some type inference algorithm.

Still, what can be learned from earlier work on type graphs is the fact that the natural type of a graph is another graph and there exists a morphism between them. Morphisms can be seen as a subtype preorder on graphs with the following meaning: whenever there exists a morphism $\varphi\colon G_1 \to G_2$, then G_2 can be seen as an approximation of G_1 and G_1 is more "exact" than G_2. The graph G_2 can intuitively be seen as more abstract since φ fuses graph elements and furthermore adds components. Since the nodes and edges of a type graph stand for structure that might be there, but must not be there, G_2 is an approximation of G_1. Or put, differently, if G_1 represents all graphs that can be mapped to G_1 with a morphism, then all these graphs are also represented by G_2. Similar ideas will be pursued in Chapter 6.

For type systems we will generalize this idea of having a morphism between the graph G and its type in the following way:

- We will, in a preprocessing step, make local modifications to G, using a so-called linear mapping. These modifications will mainly consist in removing irrelevant parts of the graph and to single out exactly the components that contain relevant information for the property to be checked.

  In the same step we will also annotate the graph by lattice elements that give us additional information about the future behaviour of a graph. Possible annotations are for instance *input* and *output* that indicate whether a node representing a port or channel is used for input or output (or both). One could also annotate ports by information describing what kinds of messages can be sent over these ports.

- After the preprocessing of G we obtain an annotated graph T for which we require that there exists a morphism into the type graph, which is annotated as well. We do not allow arbitrary annotated graphs as types, but restrict ourselves to a subset satisfying certain consistency criteria. The graph T generated by the local preprocessing step does not necessarily satisfy these criteria. The most general such graph into which T can be mapped by a morphism can be considered as the principal type of G.

  This global step can be viewed as the application of a closure operator. Different closure operators are admissible, as long as they satisfy certain properties (see below).

These steps are necessary to obtain meaningful analysis results and in order to be able to integrate well-known existing type systems, for instance for the λ-calculus or for the π-calculus.

We will illustrate the steps above using the running example Sys introduced in Section 2.3. Our aim is to show the following simple property of the graph transformation system:

> No free resource is ever assigned to a process. Or put into slightly different words: From the target node of an edge labelled R_i^f there is no outgoing path.

This property is neither very deep nor hard to check, but it is suitable to illustrate the main ideas of a type system using the running example.

Step 1 (Determine the type of a single edge): We consider the edge labels (P_i, R_i^f, R_i, D_i) of the running example. To each edge we assign a corresponding type graph having the same interface, i.e., the same number of external (numbered) nodes (see Figure 5.1).

The components of a type graph have the following meaning:

- An unlabelled edge from a node v_1 to a node v_2 means that there might be a path from v_1 to v_2, either now or in the future when the system evolves.

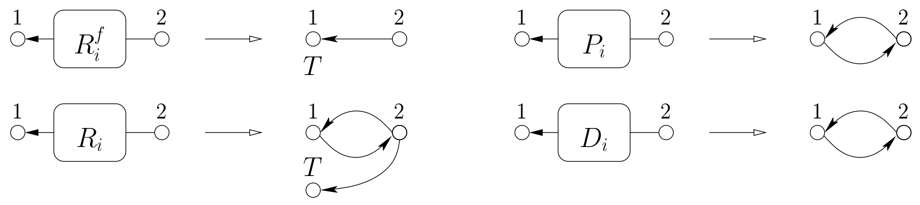


Figure 5.1: Typing edges.

- Nodes have annotations taken from a two-element lattice. In Figure 5.1 we omit the annotation whenever a node is annotated by the bottom element of the lattice and write T whenever the node is annotated by the top element. This annotation is used to mark nodes that might be the target node of an edge labelled R_i^f.

Note that there might be over-approximation in both cases. What we have to make sure is that if there is—in reality—a path between two nodes, then there exists an edge in the type graph. And if a node is the target of an edge representing a free resource, then it is marked with T. The implications in the other direction must not be satisfied. This is also the reason why T is the top element of the lattice, since the information it represents is less restricting than the information given by a missing annotation.

Into the type graphs corresponding to a single edge we have invested prior knowledge about future system behaviour. For instance, we have to know that whenever there is an edge P_1, then at some point in time the processes represented by this edge may acquire a resource which gives us a path leading from the target to the source node of the edge. In the same vein, whenever there is an edge R_i, then this resource might be released at some point in the future, which means that a node annotated with T will be created.

It is unavoidable that the local typing of a single edge depends on the rules of the type system. Also for classical type systems, such as λ-calculus type systems, a type system is given for a fixed set of rewriting rules. Then we can type every expression of the calculus respectively every possible start graph and obtain a meaningful result containing abstract information about the future behaviour of a system. It would also be an interesting direction of future work to automatically derive local types from the rewriting rules. This will have to be done in such a way that we can show type invariance, also called the subject reduction property (see also Section 5.2).

We can now replace every hyperedge of the start graph depicted in Figure 2.4 by its type and obtain the annotated graph in Figure 5.2.

Step 2 (Deriving the final type graph): If drawing an edge in the type graph means that there might be a path between these nodes, then the graph shown in Figure 5.2 is not yet a valid type graph. It is still necessary to take the transitive closure of the path-relation, leading us to the type graph depicted in Figure 5.3.

As mentioned above, this step is related to closure operators. Specifically,

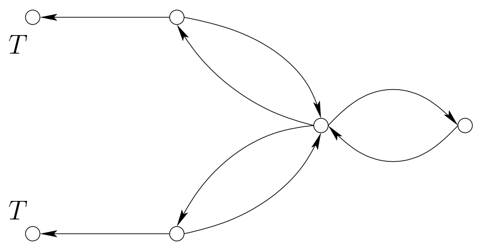


Figure 5.2: Replacing every edge by its type.

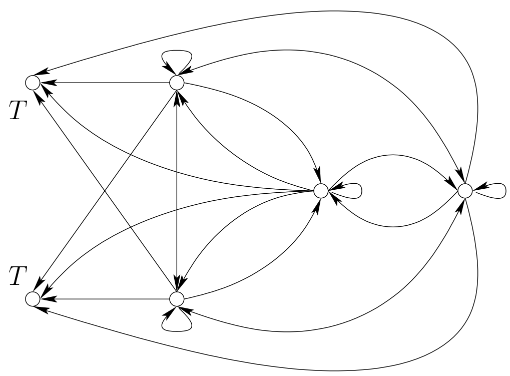


Figure 5.3: The final type graph.

it should be the case that whenever we apply this closure operator to graphs, glue these graphs together and apply the closure operator again, then this corresponds to gluing the original graphs together and applying the closure operator. Intuitively this means that gluing graphs together and taking the closure can be mixed in any possible way, provided that taking the closure is the last step. This is needed in order to ensure compositionality.

5.2 Requirements for Type Systems

Now, if we look at the type graph in Figure 5.3, there is indeed no edge going out of a T-node. From the way we have constructed the type graph we can now infer that there is indeed no outgoing path from the target of an edge representing a free resource in the start graph. This can be expressed as the first requirement we demand for a useful type system.

> **Correctness:** *Let X be a property on graphs (in our case: no free resource is ever assigned to a process) and let Y be a property on type graphs (in our case: there is no outgoing path from a node labelled T). Then it holds that whenever a graph G can be typed and the type satisfies Y, then G must satisfy X.*

Of course, this first requirement is not sufficient. In order to determine that the start graph satisfies X, we could have just inspected it. In order to show that not only the start graph but all graphs reachable from the start graph satisfy X, we need the following type invariance property.

> **Subject reduction property:** *If a graph is rewritten, it does not change its type.*

In order to actually obtain this property we have to take into account that the type of a graph might become stronger (with respect to the morphism preorder) during rewriting. So we have to define that a graph G having a type T also has all types weaker than T, i.e., all type graphs T' for which there exists a morphism $T \to T'$. Then there may be several types of a graph, but one of them is the strongest, also called minimal or principal type.

Another requirement that should hold for type systems is the following.

> **Compositionality:** *The type of a graph can be obtained by composing the types of its subgraphs, followed by an additional closure operation.*

This property is satisfied by the type system presented above, since we can always type subgraphs, glue the types together and finally compute the transitive closure as we have done before. Performing the transitive closure operation several times will not change the final result. This can be derived from the fact that the transitive closure of a relation R can be described by a universal property ("the smallest relation that contains R and for which $(x, y), (y, z) \in R$

implies $(x, z) \in R$"). In Chapter 10 we describe in more detail how defining closure operators using universal properties can lead to very general proofs of compositionality.

In general, the closure operators used in this framework are related to closure operators on lattices as they are also used in abstract interpretation [CC77]. Closure operators in our setting can be quite powerful, but in order to be admissible they have to satisfy certain properties detailed in Chapter 10. The operators derived using universal properties as explained above automatically satisfy these properties. In Chapter 10 closure operators are also called global or folding operators and are denoted by the letter f.

We have now fixed some properties that should be satisfied by every type system. This is especially important in this setting, since there is hardly any theory for type systems for graph rewriting and it is thus important to state the criteria that should be satisfied by every type system. In Chapter 10 we present a general framework for such type systems that satisfies the above properties.

A question that has not been addressed in great detail up to now, but that will be treated in Chapter 10 is the inductive construction of graphs. For "text-based" calculi types are usually defined inductively on the structure of a term. Graphs are usually not defined inductively, but are seen as collections of nodes and edges; inductive views on graphs and graph transformation are rare. Hence Chapter 10 will also introduce an appropriate notion of composing and decomposing graphs.

5.3 Proving the Subject Reduction Property

Such a general framework only makes sense if we can obtain useful theorems on this high level of abstraction. One such theorem is the definition of very general closure operators that ensure compositionality.

Another, and even more important, theorem concerns the subject reduction property. We can show that it is sufficient to show type invariance locally for every rewriting rule and that this implies the subject reduction property.

So, coming back to our example we show that the subject reduction property holds by inspecting every rule separately. For instance, we consider rule [AcquireResource$_{i,i}$] (see Figure 5.4). We type both the left-hand and the right-hand side and can see that the type of the right-hand side is stronger than that of the left-hand side, i.e., there exists a graph morphism φ from right to left. This graph morphism also has to preserve external nodes and annotations, where the annotation of a node $\varphi(v)$ (the image of a node v) must be greater than or equal to the annotation of v. All these properties are satisfied in this case.

By checking the remaining two rules as well we can conclude that this type system does satisfy the subject reduction property. Now, the type graph of the start graph depicted in Figure 5.3 does indeed satisfy property Y (no outgoing paths from a T-node), and we can hence infer that every reachable graph satisfies the property we wanted to check.

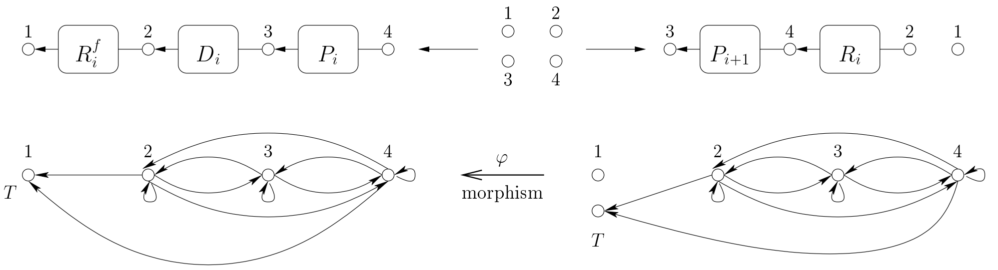


Figure 5.4: Typing rule [AcquireResource$_{i,i}$].

5.4 Applications of Type Systems

There are two quality tests such a general framework can be subjected to. First, we should show that existing type systems can be obtained as special cases by instantiating the framework. Second, we could generate new type systems checking useful properties.

We have done the former by showing how the standard π-calculus type system and a type system for concurrent object-oriented programming can be derived from our type system (see Chapter 10). It is not a part of this thesis but we have also shown how a simple type system for the λ-calculus can be obtained as a special case.

Furthermore the framework has been instantiated in order to derive information on input/output behaviour (see also Section 5.5), for checking acyclicity and for enforcing a security policy for untrustworthy applets (see [Kön00b]). Furthermore Chapter 10 contains a section on a type system for concurrent object-oriented programming.

5.5 Counting Input/Output-Capabilities—A Transfer of Ideas to the World of Mobile Processes

So far we have annotated type graphs using lattice elements, which means taking the supremum or join of the lattice whenever two nodes or edges are merged (see for instance Figure 5.4 where the two nodes on the left of the right-hand side are merged). This leads to a smooth theory, but does not allow counting, i.e., we are not able to check properties such as "there is always at least one edge labelled A" and "there are at any time at most three processes listening at a certain port". For this it is necessary to annotate type graphs with natural numbers or, more generally, monoid elements.

Unfortunately the very general theory of lattice annotations developed in Chapter 10 does not easily generalize to monoid annotations. This is still a challenging task for future work. But it turned out that such type systems can be given for the π-calculus using more specific annotations (assigning single monoid elements to channel names). This has been done by transferring the concepts already developed for graph rewriting into the world of mobile processes. Although this type system looks quite different from type systems for graph rewriting on the surface, there are strong connections, which is also witnessed by the fact that the standard type system for the π-calculus can be obtained as a special case of the graph type system (see Chapter 10).

The π-calculus type system obtained in this way nicely complements similar type systems for the π-calculus [KPT99]. More details can be found in Chapter 11.

5.6 Evaluation

Type systems are a simple, modular and efficient method for automatic analysis. As we have seen they can also be adapted to dynamic systems and used to derive invariants of structural properties of a graph or network. Type systems of this kind are new, but they integrate nicely with existing type systems, for instance for the π-calculus and for the λ-calculus.

The central idea is that one can assign types to graphs according to their structure, taking into account all possible behaviours, so that we obtain a compact abstract representation of everything that might happen in the future. Another—quite different—compact representation of the possible evolutions of a system is the unfolding of a system, which contains much more detailed information. We will show in the next chapter how unfoldings can be used in order to obtain a more expensive, and also more precise analysis technique.

The efficiency of type systems is an advantage and a disadvantage at the same time. A disadvantage, since having a technique which is runtime-efficient also means that a fair amount of approximation is involved. Furthermore type systems are meant for checking system invariants and not for model-checking formulas of a temporal logic. So it seems that type systems are mainly useful for fast debugging and not for checking overly complex properties. Certainly one analysis method such as type systems can not stand on its own, but has to be complemented with other methods, in order to be useful and applicable in practice. This will also require the closer integration of various techniques in the future.

Chapter 6

Approximating Graphs by Petri Nets—An Unfolding-Based Approach

Since graph transformation systems can model Turing-complete systems, all interesting questions about the behaviour of a graph grammar, such as termination, the occurrence of certain events or temporal properties are undecidable. Abstract interpretation is a partial solution to this problem: instead of executing the system itself we execute an abstract version of it which over-approximates the original system behaviour.

Now, there are two important questions to answer:

- *Which abstract system model should be used?* We chose to approximate graph transformation systems by Petri nets for the following reasons: graph transformation systems can be seen as a straightforward generalization of Petri nets (see also Section 2.5) and Petri nets are a simpler computational model which is not Turing-complete and which is easier to verify. Many problems that are undecidable for graph transformation systems become decidable for Petri nets [EN94, Jan90, HRY91].

  Furthermore Petri nets are inherently concurrent, meaning that no artificial ordering has to be imposed on events happening concurrently, and they also have a notion of locality, that is state changes are described by local transformations only. This has the advantage that these features, which also graph transformation systems possess, are not lost in the approximation.

  Many analysis techniques such as *invariants* and *linear algebraic techniques* are known for Petri nets. Another well-known concept are *coverability graphs* (also called Karp-Miller graphs) [Fin91], which can be seen as a safe approximation of the reachable markings of a Petri net. The analysis method presented in this chapter will both rely on and generalize this construction. Additionally, a lot of work has been done in the area of model-checking Petri nets, where part of this work is based on *unfolding techniques* [McM93, Esp94]. Unfolding techniques are a partial order reduction method allowing us to represent the events concurrently taking place in a system in a non-interleaving way, thus making the representation much more compact.[1]

- *Which properties are reflected by the abstraction?* If a property, for instance the existence of a run where the event e happens at some point, is satisfied for the abstraction, it is not necessarily the case that the property is also satisfied for the original system (see also Section 3.2). If, however, this is the case for a property, we say that the property is reflected by the abstraction.

  Hence it is our task to define appropriate logics, for expressing structural properties of a single graph as well as for expressing temporal properties, and to identify subsets of formulas that are reflected and can be used for verification.

Note however, that approximation by Petri nets is not the only choice. In the work of Rensink [Ren04a], for instance, graph rewriting is approximated by abstract graph transformation, where abstract graphs have additional connectivity and multiplicity information. The situation is somewhat related to the theory of type graphs with annotations (see Chapter 5), especially to the annotation with monoid elements.

Furthermore, in shape analysis [SRW02] sets of graphs are represented by structures in a 3-valued logic.

6.1 Over-approximation

Let us first review the main ideas behind over-approximations and abstract interpretation. Abstract interpretation was first introduced for imperative programs [CC77], which are interpreted on abstract instead of concrete data. For instance, instead of computing with integers, one uses abstract values such as "odd" and "even". The relation between abstract and concrete data values is described by a so-called Galois connection with a mapping α (the abstraction) mapping sets of concrete values to sets of abstract values, and a mapping γ (the concretion) mapping sets of abstract values to sets of concrete values. It must, for instance, hold that applying an abstraction followed by a concretion over-approximates the original value. The concept of Galois connections is in general defined for lattices.

Instead of abstracting data values, one can also abstract on another level and represent concrete system states by abstract system states. It is required that there exists a simulation relation between the two systems, i.e., whenever the concrete system can make a step, this can be mimicked by the abstract system, but not necessarily the other way round. In [LGS$^+$95] it has been

[1] If, for instance, in a system three events a, b, c are happening concurrently, an interleaving representation would result in six sequences abc, acb, bac, bca, cab, cba. A non-interleaving partial order representation of system behaviour, however, would simply consist of an unordered set $\{a, b, c\}$.

shown that the simulation property can also be expressed in terms of Galois connections and that these two notions can be considered equivalent.

One can approximate complex systems by finite transition systems [CGL99], but also by infinite-state systems that are easier to verify than the original system. For instance one can simulate an imperative program with stack and procedure calls by a pushdown system [Sch02, EHRS00] or one can abstract π-calculus processes by processes of a simpler process calculus [IK01]. Other work in this direction includes abstractions for mobile ambients using shape analysis [NN01] and for multithreaded Java programs [DJFB02].

Abstraction can be conveniently combined with model checking. However in order to obtain the necessary reflection results, it is necessary to suitably restrict the temporal logics in which system properties can be formulated. For instance in [CGL99] ACTL, a subset of CTL (computation tree logic) which has only universal quantification, is used. Similarly [LGS+95] defines a subset of formulas of the μ-calculus suitable for checking properties on the abstraction. Using both under- and over-approximations or mixed abstractions larger subsets or even the full logics can be checked [DGG97, Kel95].

6.2 Approximating Graph Transformation Systems by Petri Nets

Before explaining the approximation technique itself, we will first describe its outcome, which is a Petri net with additional graph structure, a so-called Petri graph. This Petri graph can be seen as an abstraction of the graph transformation system in the sense that the transition system on graphs can be simulated by the transition system on markings of the Petri net. This will be described in the following in more detail.

A possible Petri graph abstracting the running example of Chapter 2 is the structure depicted in Figure 6.1. It consists of a hypergraph component consisting of nodes and edges (see Figure 6.2), and a Petri net component (see Figure 6.3) with places and transitions. Places and edges coincide and arcs between edges and transitions are drawn using dashed lines. The initial marking is indicated using black tokens. Also note that in this case it is just a coincidence that every edge label occurs exactly once and that there are exactly as many transitions as rules. In general, there will be more edges than edge labels and more transitions than rules in the Petri graph, especially when we compute more precise approximations (see Section 6.6).

In this situation, if we assume that the name of every edge is equal to its label, we could also describe the transitions in the following way:

transition	rule	pre-set	$\to$	post-set
t_1	[AcquireResource$_{1,1}$]	P_1, D_1, R_1^f	$\to$	P_2, R_1
t_2	[AcquireResource$_{2,2}$]	P_2, D_2, R_2^f	$\to$	P_3, R_2
t_3	[ReleaseResources]	P_3, R_1, R_2	$\to$	$P_1, D_1, D_2, R_1^f, R_2^f$
t_4	[CreateNewProcess]	P_1, D_1, D_2	$\to$	$2 \cdot P_1, 2 \cdot D_1, 2 \cdot D_2$

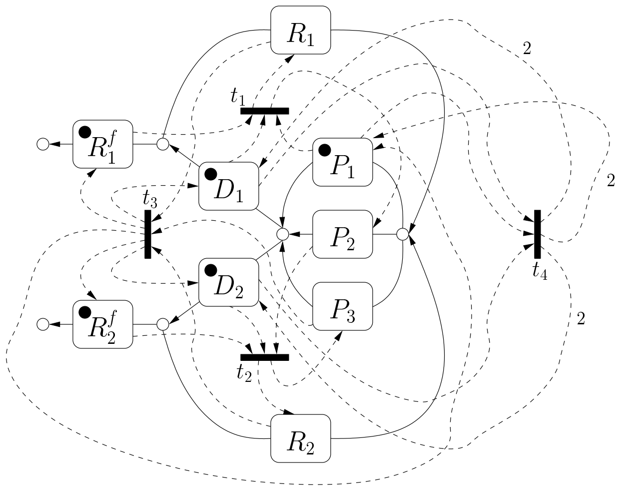


Figure 6.1: The approximated unfolding of system Sys.

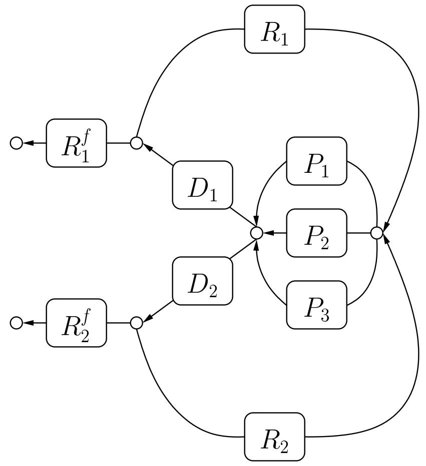


Figure 6.2: The hypergraph component of the approximated unfolding of system Sys.

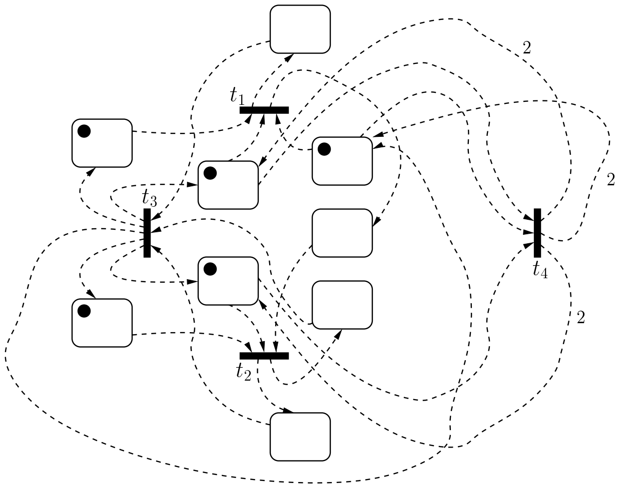


Figure 6.3: The Petri net component of the approximated unfolding of system Sys.

The Petri graph is a valid abstraction of the system, in the following sense:

> There exists a relation S—a so-called *simulation*—between the reachable graphs in the graph transformation system and the reachable markings in the Petri net such that:
>
> - It holds that $(G_0, m_0) \in S$, where G_0 is the start graph and m_0 is the initial marking.
> - Whenever $(G', m') \in S$ and G' can be rewritten to G'' using rule r, then there exists a marking m'' such that m'' can be obtained from m' by firing a transition corresponding to rule r. (In other words: the relation S is a simulation).
> - For every $(G', m') \in S$ there is a morphism from G' into the hypergraph underlying the Petri graph, such that the image of the edges of G' corresponds to m'.

While the first two properties are standard for simulations, the third property requires closer scrutiny. In Figure 6.4 we can see an example of a pair (G', m') where G' is a reachable graph—obtained by applying rule [CreateNewProcess] to the initial graph—and m' is a reachable marking—obtained by firing the transition corresponding to rule [CreateNewProcess]. The graph G' can be mapped into the graph component of the hypergraph such that the tokens represent the cardinality of the preimage of each edge. For instance there are two edges labelled P_1 which are mapped to the edge P_1 on the right-hand side.

So, the information that is reflected is the number of occurrences of edges as well as inequality of nodes (if two nodes are distinct in the Petri graph, they are also distinct in the graph represented by a marking). What might get lost however is information concerning the equality of nodes. For instance the two source nodes as well as the two target nodes of the edges labelled P_1 are fused by the morphism. This loss of information has to be taken into account whenever we check the validity of logical formulas on the approximation (see Section 6.5).

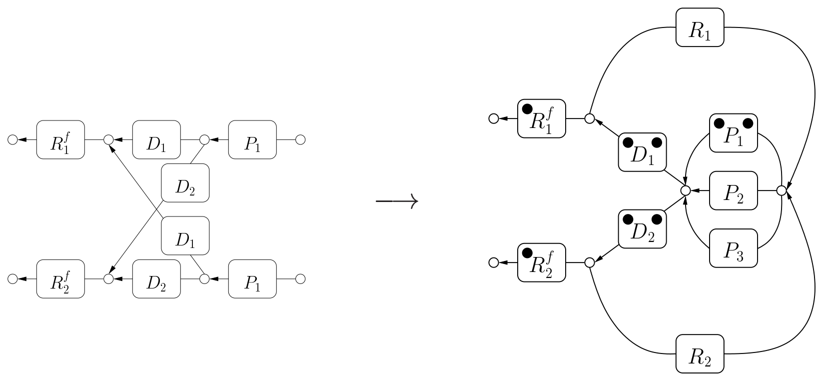


Figure 6.4: A graph approximated by a marking.

Figure 6.5 shows one step of the graph transformation system—transforming the start graph using rule [CreateNewProcess]—and the corresponding firing of a transition in the Petri graph. The left-hand side and the right-hand side of the rule are marked with bold lines. The image of the left-hand side is marked by three black token in the left-hand Petri graph. The two remaining tokens are shaded grey. This application of rule [CreateNewProcess] corresponds to the firing of the depicted transition t_4, all other transitions have been removed for clarity. The transition removes these three tokens and creates six, two for each edge (note the arcs labelled 2). This gives us the situation depicted in Figure 6.4.

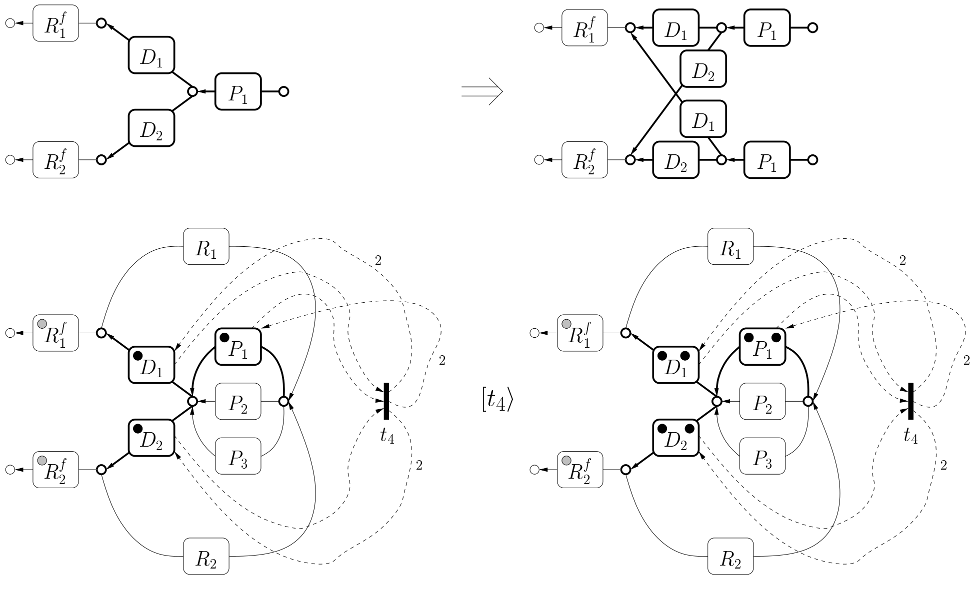


Figure 6.5: The Petri graph simulating the graph transformation system.

Every graph transformation step can be simulated in such a way, but not necessarily every firing of transitions in the Petri graph can be simulated by graph rewriting. This is a phenomenon caused by over-approximation and the introduction of spurious runs.

6.3 Unfolding Techniques

In order to obtain such Petri graphs we use unfoldings, which are structures that represent all possible runs of the system. The unfolding of a graph grammar can be seen as an infinite Petri net. (Alternatively it can also be seen as another graph grammar [Bal00, Rib96].) So, in order to obtain a finite structure, we approximate these unfoldings, a novel approach that has not been used before.

Unfoldings represent all possible runs of a system in a compact way in a single structure. This compactness is ensured in the following way:

- Whenever two runs have the same prefix, this prefix is only represented once.
- Whenever several events may happen concurrently, we do not represent all possible interleavings of these events, but remember only the set of events and the causal ordering between them. For this reason, unfoldings are a special case of partial order reduction techniques.
- If, as in the case of graph grammars or Petri nets, the state of a system is a non-atomic object such as a graph or a set of tokens, we represent state changes only in a local way. That is if a state s_1 is rewritten to a state s_2, we do not represent s_2, but only the update that is needed to transform s_1 into s_2.

System models that can be unfolded are for instance Petri nets [Win87, McM93], graph grammars [BCM99, Rib96] and products of transition systems [ER99]. We will in the following give a general description of the unfolding procedure, independent of the system model. We only assume that the system is given by a set of rewriting rules. We construct a series of Petri nets $U_0, U_1, U_2, \dots$ with initial markings that will converge to the full unfolding. These Petri nets may possibly contain additional structure such as the hypergraph structure in the Petri graphs introduced above.

$i = 0$: U_0 is the initial system state. All items of the initial state are initially marked. By items we mean the atomic subcomponents of a system state, such as places in the case of Petri nets and nodes and edges in the case of graphs.

$i \to i+1$: We take U_i and search for a match of a left-hand side L of a rewriting rule. The items of the left-hand side have to be coverable in U_i, i.e., there must be a marking reachable from the initial marking that contains at least all these items.

Then we attach the right-hand side R to U_i according to the embedding rules (if there are any) and add a transition t that records that L can be removed and R can be created.

Naturally the construction is quite informal and many details still have to be filled out. Using this construction we obtain a potentially infinite Petri net. Such Petri nets are called occurrence nets, since each creation of an item has a unique cause and furthermore the net is acyclic. We have chosen here to represent unfoldings by occurrence nets, but note that it is sometimes more convenient to use a different representation, for instance occurrence grammars.

As an example, we start unfolding the graph transformation system Sys from Section 2.3. We begin with the start graph, where the initial marking is indicated by black tokens (see Figure 6.6).

We first find a match of rule [CreateNewProcess] and unfold it which results in transition t_1. There is a match of rule [AcquireResource$_{1,1}$] as well, but the order in which these matches are unfolded is irrelevant. We unfold this rule by attaching the right-hand side according to the interface and add a transition recording this transformation step. In order to distinguish connections between edges and nodes from connections between edges and transitions, we draw the latter with dashed lines. Remember that edges basically play the role of places in a Petri net.

Now we have three matches of rule [CreateNewProcess] (one of them already unfolded) and two matches of rule [AcquireResource$_{1,1}$], all of which are coverable. We choose to unfold one of the matches of rule [AcquireResource$_{1,1}$] and obtain transition t_2. (Ignore the grey edges for the moment.) For the moment we stop unfolding at this point.

The structures that we obtain during unfolding are again Petri graphs as introduced earlier.

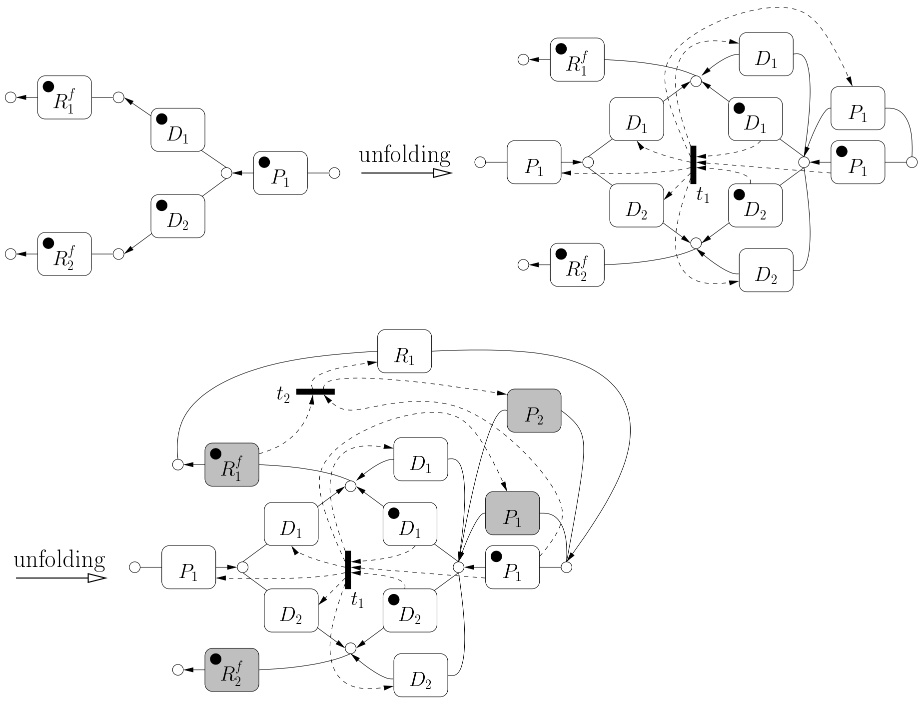


Figure 6.6: Unfolding a graph transformation system.

There are some further points to be discussed:

Concurrency, causality, conflict: Above, we said that in order to check whether a left-hand side should be unfolded, we should check whether a marking is coverable. Coverability is verified by using coverability graphs [KM69, Rei85], a technique which—despite its straightforward algorithm—is known not to be primitive recursive. But in the case of occurrence nets, checking coverability is greatly simplified, by computing the following relations:

- *Causality:* Causality is the smallest transitive relation for which it holds that all items in the post-set of a transition are causally dependent on all items in the pre-set of a transition. It can be easily computed while the net is constructed.
- *Conflict:* Two items are in conflict whenever the two transitions that create them consume the same item in their pre-set. Furthermore, whenever two items are in conflict, all their causal successors are in conflict as well. That means that no two such items can occur together in the same branch of a computation, not even at different times. Again, the conflict relation can be computed during the unfolding.
- *Concurrency:* Two items are concurrent whenever they can occur in the same computation at the same time. In other words, a set of items is concurrent when it can be covered. The concurrency relation can be obtained as the complement of the union of causality and conflict.

  So checking whether a set of items is coverable amounts to checking that neither pair in the set is contained either in the causality or in the conflict relation.

Figure 6.6 exhibits examples for these relations. For instance, the grey edges labelled P_2 and R_2^f are concurrent. There is a reachable marking obtained by firing transition t_2 once that covers both of them. The two grey edges labelled R_1^f and P_2, on the other hand, are causally related. The edge P_2 can only be covered when the token contained in R_1^f is consumed. An example for a pair of edges being in conflict are the grey edges labelled P_1 and P_2. The edge P_1 is in the post-set of t_1, while the edge P_2 is in the post-set of t_2, and both transition consume a token from the edge labelled P_1 which is initially marked.

Contextual arcs and inhibitor arcs: Especially when we are working with graph grammars not all edges of the left-hand side will be consumed. The interface graph I of a rule $L \leftarrow I \rightarrow R$ is not removed, but its presence is required in order to apply the rule. In this case the concept of "contextual arcs" or "read arcs" [Bus98] is helpful. These arcs connect transitions with places in a Petri net and a transition may only fire if the places with which it is connected via contextual arcs are covered. The unfolding technique can be extended in order to deal with this feature [VSY98].

If the interface consists of nodes only, i.e. if it is discrete, and nodes are never deleted (but become disconnected instead), then there is no need to use contextual arcs. It is enough to track causal dependencies on edges, causal dependencies on nodes can then be derived from this information. This approach is taken in Chapter 12 and we have also used it in the example unfolding above, whereas in Chapter 15 we also allow non-discrete contexts.

If the graph transformation system is extended with negative application conditions, i.e., rules may not be applicable if specific subgraphs are present, then we need the concept of "inhibitor arcs" [Bus98] in order to model this situation. This case is quite problematic, since the concurrency relation is not computable in the efficient way described above. Furthermore there are problems if we consider approximated unfoldings as in Section 6.4. These problems are not too surprising since Petri nets with inhibitor arcs are already Turing-complete. For this reason we will not consider negative application conditions in the rest of this thesis.

The unfolding is usually infinite, also for finite-state systems having infinite runs. However, if a system is finite-state there is a solution to this problem: We know that already a finite prefix of the unfolding represents all reachable system states. This prefix can be efficiently constructed by looking for so-called *cut-off transitions*. A transition t is a cut-off whenever the system state it represents, i.e., the state obtained after firing t and all its causes, can also be obtained by another transition t' having a smaller causal history. Hence, in this case, one does not continue unfolding after cut-off transitions, as suggested by the name cut-off. The prefix obtained in this way is called finite complete prefix, it has been first presented in [McM93], substantial optimizations have been introduced in [ERV02]. Our contribution to this area is to show that finite complete prefixes can also be constructed for graph grammars (see Chapter 15). We also show how to check a logic on graphs on this finite unfolding.

6.4 Approximated Unfolding

If a system is infinite-state, it is not possible to stop at a certain point in the unfolding and claim to have seen all reachable states. Just stopping at a certain point will only give us a subset of all reachable graphs. Thus, we will extend the unfolding procedure and perform—in addition to unfolding steps—also folding steps that merge parts of the unfolding. Now the step that transforms U_i into U_{i+1} reads as follows:

$i \to i+1$: We take U_i and search for the match of a left-hand side L that has not already been unfolded. Then we

- search for another match of the same-left hand side L' such that all items of the first match are causally dependent on items of the second match and merge both left-hand sides *(folding step)*

- *or*—if no folding is possible—we unfold the match *(unfolding step)*.

When this procedure terminates we call the result the approximated unfolding or covering of the graph transformation system.

The folding condition is there to ensure that the unfolding terminates. Intuitively, finding an occurrence of a left-hand side L that causally depends on an earlier occurrence means that we have located a potential cycle that is short-circuited by merging both left-hand sides. Showing that this condition implies termination in all possible cases is non-trivial. The proof can be found in Chapter 12 (see Proposition 12.4.7).

Figure 6.7 shows three steps of the algorithm described above. First, a match for rule [CreateNewProcess] is found and unfolded as in Figure 6.6. Then we can find three matches of the left-hand side of rule [CreateNewProcess], of which two are causally dependent on the third one, i.e., the match which is initially marked. So we can merge all three left-hand sides in two folding steps and obtain the structure shown below.

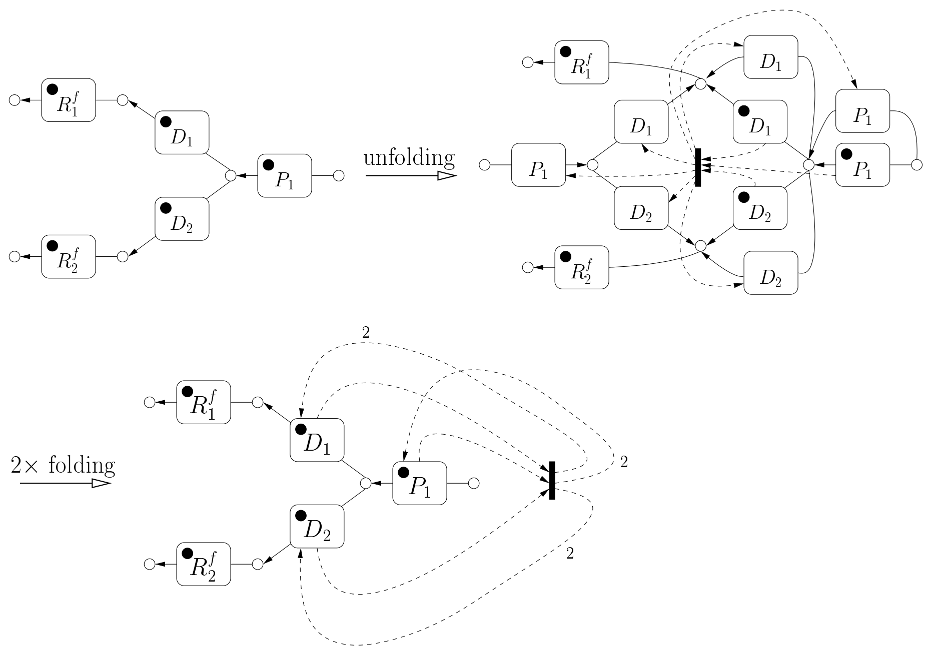


Figure 6.7: An unfolding step and two folding steps.

If we continue doing unfolding and folding steps as described in the algorithm, we finally end up with the Petri graph described earlier and depicted in Figure 6.1. Although the algorithm for computing the approximated unfolding is non-deterministic, it can be shown that it is confluent and that hence the resulting Petri graph is uniquely determined. That is, even if we had applied the unfolding and folding steps in a different order, we would have obtained the same approximation. This shows that the over-approximation is not arbitrary and does not change if a different sequence of steps is chosen.

6.5 Logics for Analyzing Graph Transformation Systems

The Petri graph is an over-approximation of the original graph transformation system, but we have yet to show how we can extract the information contained in it and use it for model-checking.

As discussed above, it is necessary to restrict the logics accordingly so that its formulas are reflected by the abstraction. For the temporal operators of the logics, the situation is equivalent to the one for other abstractions of transition system, so that we can use suitable subsets of CTL, such as ACTL [CGL99], or a subset of the modal μ-calculus [LGS+95], where existential quantification ("There exists a successor state such that ...") is disallowed. More details about the temporal part of the logics are given in Chapter 13.

We will therefore concentrate on the logics describing properties of system states which, in this case, are graphs. In Chapter 14 we describe a second-order monadic logic on graphs that allows quantification over edges as well as over sets of edges. This logic is quite expressive, unlike a first-order logic it allows us to state properties such as "There exists no circle of edges labelled A." Note that this property is reflected by the abstraction, i.e., whenever it holds for the graph obtained from a marking by duplicating all edges according to the number of tokens contained in the edge, then it also holds for all graphs represented by this marking in the sense explained in the definition of the simulation relation above. This is true since this property requires us to check the *inequality* of nodes, but not their *equality*. A property such as "There exists a circle of edges labelled A." is *not* reflected by the abstraction.

Summarizing we can say that in the logic all forms of quantification (existential/universal as well as first-order/second-order), disjunctions as well as conjunctions can be allowed. Furthermore we can inspect edge labels, equality and inequality of edges. For nodes equality checking is disallowed, i.e., we can not use formulas such as

$$\exists x \exists y (lab(x) = A \wedge lab(y) = B \wedge source(x) = source(y))$$

("There exist two edges labelled A and B having the same source node.").

This is a restriction that is caused quite naturally by the form of approximation we are using. However, we are currently also exploring possibilities to weaken this restriction.

In Chapter 14 a type system is presented that identifies formulas which are reflected. In order to actually check these formulas on the Petri net, they have to be transformed into formulas on Petri net markings. We are using a simple propositional logics on Petri net markings, where the only atomic predicate is of the form $\#p \geq c$, meaning that there are at least c tokens in place p. Surprisingly, even monadic second-order logic formulas can be encoded into such a simple logic—losing no information at this stage—provided that we have already computed the approximating Petri net. This is shown in more detail in Chapter 14.

For many practical examples checking whether all reachable graphs satisfy a formula amounts to showing that certain markings are not coverable, which can be done using coverability graphs [Rei85] or a backward reachability algorithm for Petri nets [AJKP98]. These possibilities were also explored in [Tur04]. Also, more complicated properties, even in combination with temporal operators are decidable for Petri nets in many cases [HRY91, HR89, Jan90]. However, one has to be careful, since state-based temporal logics for unbounded Petri nets are not decidable in general, whereas action-based temporal logics usually are decidable [Esp97]. Still, in practice it has turned out that many interesting properties can be proved using our prototype tool AUGUR (see Section 6.7).

Apart from using a monadic second-order logic it is possible to use more user-friendly specification languages such as regular expressions that specify forbidden paths that are disallowed in all reachable graphs [Rel04]. A different possibility would be to allow the user to specify forbidden subgraphs using a graph editor.

Regardless of the specification language, if we want to show for system Sys that no vicious cycle corresponding to a deadlock as shown in Figure 14.3 occurs, we can do this by requiring that no two edges labelled P_2 will ever be present in a graph. If no such edges are present, there can be no processes waiting for each other to release their resources. This can be shown by checking that the formula $\#p_2 \leq 1$ holds for all reachable markings, where p_2 is the place corresponding to the edge labelled P_2. We have now derived this formula in an ad-hoc manner, but we could also have obtained it by describing vicious circles either in the second-order monadic logic or as regular expressions.

6.6 Obtaining Better Approximations

Overly coarse approximations are an inherent problem in abstract interpretation. Whenever a formula does not hold for the approximation, it might either be the case that it is also invalid for the original system, or that we have approximated too strongly, thereby losing information. In Chapter 13 we describe a partial solution to this problem. It is possible to compute better and better over-approximations using approximated unfoldings if we unfold without folding steps up to a certain causal depth and allow folding only for items of a larger depth. It can be shown that the sequence of these increasingly more exact over-approximations (or coverings) converges to the full unfolding in the limit. Using the words of category theory, the limit of the chain of over-approximations is the exact unfolding.

In the same vein, we can also under-approximate a graph transformation system by truncating the unfolding at a certain point. It can be shown that the colimit of the chain of under-approximations is the full unfolding as well.

With these results we can justify our method of approximation: We might approximate too strongly, but by computing better and better approximations we can converge to the original system behaviour in the limit. Naturally these better approximations increase in size. In order to minimize this blowup, we are currently adapting the technique of counterexample-guided abstraction refinement. In this technique, the refinement of an over-approximation which is too coarse is guided by spurious counterexamples. One attempts to refine only the parts of the approximation that are necessary in order to make the spurious counterexample disappear.

6.7 Tool Support

There exists a prototype tool called AUGUR[2] implementing a large part of the functionality described above. This tool was implemented and is currently being implemented in cooperation with Vitali Kozioura, Tobias Heindel, Ingo Walther, Nicolas Relange, Sinan Turan and Lars Heinemann at the University of Stuttgart.

The overall design principle is to produce an open and modular system which can be easily extended and combined with other tools. Hence the focus is not on the graphical user interface but on producing components that can be combined with each other and with external tools.

With AUGUR one can:

- Compute approximated unfoldings of graph grammars specified in GTXL (Graph Transformation Exchange Language), a XML standard for graph transformation systems.[3] The resulting Petri graph is then written to a file in the GXL (Graph Exchange Language) format. The first prototype version which has since been extended is described in [Wal03].
- It is also possible to compute subsequently better approximations as described in Section 6.6 by disallowing folding steps until after depth k.
- The input graph transformation and the output can be visualized using the GraphViz package[4] with its tools `dot` and `neato`.
- For analysis of the resulting Petri graph, the Petri net component can be converted to the input format of the Petri net tool LoLA (Low Level Analyzer) [Sch00]. One can also use the tools for coverability checking described in [Tur04]. Furthermore the encoding of regular expressions into formulas on markings as described above has already been implemented (see [Rel04]).

For instance, the approximated unfolding of the running example system Sys can be computed using AUGUR. The hypergraph component and Petri net component are shown in Figure 6.8 respectively Figure 6.9 (compare with Figures 6.2 and 6.3). The numbers associated with edges and transitions are internal identities used by the tool.

6.8 Evaluation

Our approximation technique is—at the time of writing—the only fully developed method for analyzing general infinite-state graph transformation systems.

[2] `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`

[3] See `http://www.gupro.de/GXL/`, `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`

[4] See `http://www.research.att.com/sw/tools/graphviz/`

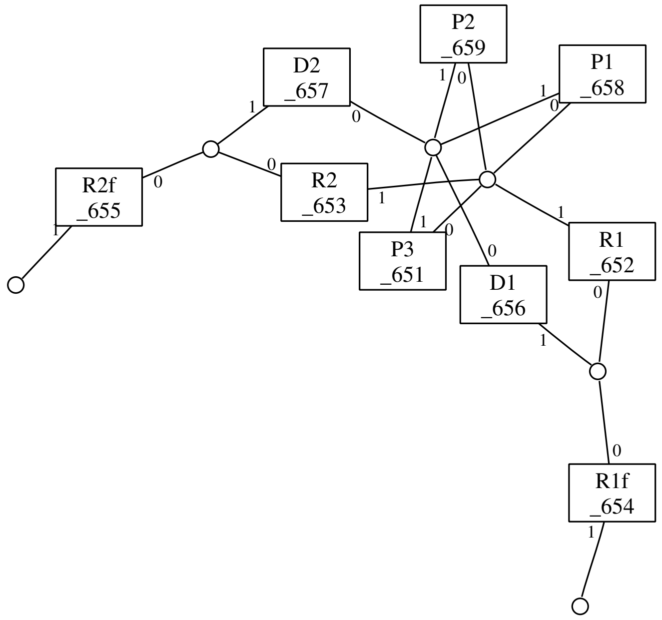


Figure 6.8: The hypergraph component of the running example (output of the tool AUGUR)

Although at the moment only a prototype version of the analysis tool AUGUR exists, whose efficiency could still be greatly improved, we were able to analyze non-trivial examples such as a mutual exclusion protocol (8 rules), a network of public and private servers (12 rules) and insertion of elements into red-black trees including rebalancing (26 rules).

It seems that this is a powerful technique that can be easily mechanized, as has been shown with our implementation. It can also be easily integrated with existing tools for graphs and Petri nets. The accuracy of the approximation is parameterized, i.e., we can obtain better abstractions if necessary.

However, in order to avoid blowups in the size of the approximation it will be necessary in the future to develop methods for refining the approximation in a more specific and goal-oriented manner. Furthermore we will work on the extension of specification languages and their integration.

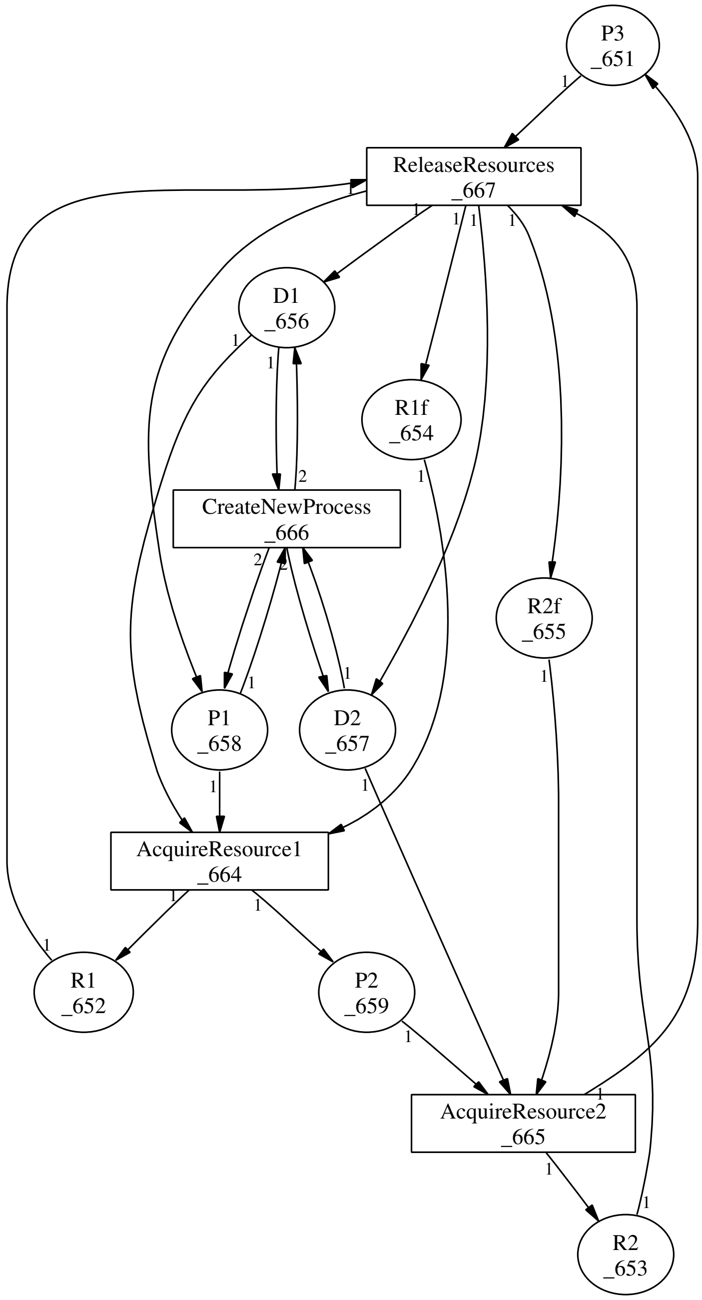


Figure 6.9: The Petri net component of the running example (output of the tool AUGUR)

Chapter 7

Conclusion

We have shown—in a largely informal way—how to transfer various verification and analysis techniques into the world of graph rewriting. The theory behind these techniques can be found in more detail in the contributions forming the second part of this thesis.

While at the moment these different methods are mostly unconnected, it will be an interesting topic of future research to integrate these techniques in order to combine their strengths and alleviate their weaknesses. It is important that this integration is done thoroughly and not in an ad-hoc way.

In the context of this work a good starting point for such an integration is the combination of type systems and approximated unfoldings. Some properties which are very hard to check on the approximating Petri nets, such as for instance the acyclicity of all reachable graphs, are easy to analyze using a type system. So it seems that the information that can be obtained from typing a graph grammar should be incorporated into the analysis of the Petri net. In order to combine these two types of analysis, it will be necessary to modify the encoding of logical formulas on graphs into formulas on Petri net markings in such a way that typing information is taken into account.

A different possibility of integration is to do approximation and static analysis for graph rewriting with external environment, connected to the bisimulation congruences presented in Chapter 4.

Furthermore it has turned out that the techniques described in this thesis and especially unfoldings of graph transformation systems have applications in areas different from verification. As we have already mentioned in the introduction diagnosis and testing can also be extremely useful for understanding existing software and hardware systems. We are currently exploring how unfoldings of graph transformation systems can be put to good use in the areas of distributed diagnosis and of test case generation for code generators [BKS04]. In the former line of research graph grammars are used in order to model mobility, whereas in the latter case graph rewriting is needed since the input to the code generator is a graphical Simulink model which is transformed using graph rewriting rules.

These examples suggest that there might be further interesting application areas for the techniques presented here.

Part II

Contributions

The following list contains all contributions to this thesis. The papers can be grouped and assigned to the chapters of the first part of this thesis as follows:

Behavioural Equivalences for Graphs (Chapter 4):

[1] Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Proc. of FOSSACS '04*, pages 151–166. Springer, 2004. LNCS 2987.

[2] Barbara König and Ugo Montanari. Observational equivalence for synchronized graph rewriting with mobility. In *Proc. of TACS '01*, pages 145–164. Springer-Verlag, 2001. LNCS 2215.

Assigning Types to Graphs (Chapter 5):

[1] Barbara König. A general framework for types in graph rewriting. In *Proc. of FST TCS '00*, pages 373–384. Springer-Verlag, 2000. LNCS 1974.

[2] Barbara König. Analysing input/output-capabilities of mobile processes with a generic type system. *Journal of Logic and Algebraic Programming*. to appear.

Approximating Graphs by Petri Nets—An Unfolding-Based Approach (Chapter 6):

[1] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[2] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.

[3] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03 (International Static Analysis Symposium)*, pages 255–272. Springer-Verlag, 2003. LNCS 2694.

[4] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.

Chapter 8

Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting

(Joint work with Hartmut Ehrig)

Abstract

Motivated by recent work on the derivation of labelled transitions and bisimulation congruences from unlabelled reaction rules, we show how to solve this problem in the DPO (double-pushout) approach to graph rewriting. Unlike in previous approaches, we consider graphs as objects, instead of arrows, of the category under consideration. This allows us to present a very simple way of deriving labelled transitions (called rewriting steps with borrowed context) which smoothly integrates with the DPO approach, has a very constructive nature and requires only a minimum of category theory. The core part of this paper is the proof sketch that the bisimilarity based on rewriting with borrowed contexts is a congruence relation.

8.1 Introduction

In the last few years the problem of deriving labelled transitions and bisimulation congruences from unlabelled reaction or rewriting rules has received great attention. This line of research was motivated by the theory of bisimulation congruences for process calculi, such as the π-calculus [SW01]. A bisimilarity defined on unlabelled reduction rules is usually not a congruence, that is, it is not closed under the operators of the process calculus. Congruence is a very desirable property since it allows us to replace a subsystem with an equivalent one without changing the behaviour of the overall system and futhermore helps to make bisimilarity proofs modular.

Previous solutions have been to either require that two processes are related if and only if they are bisimilar under all possible contexts (see [MS92b]) or to derive a labelled transition system manually. Since the first solution needs quantification over all possible contexts, proofs of bisimilarity can be very complicated. In the second solution, proofs tend to be much easier, but it is necessary to show that the labelled variant of the transition system is equivalent to the unlabelled variant.

So the idea which was formulated in the papers of Leifer/Milner [Lei01, LM00], Sewell [Sew02] and Sassone/Sobocinski [SS03] is to automatically derive a labelled transition system 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. Consider, for example, the CCS process $a.P$. It reduces when put into the contexts $\_ \mid \bar{a}.Q$ and $\_ \mid \bar{a}.Q \mid b.R$, but one is interested only in the first context, since it is in some sense smaller than the second one. This yields the labelled transition

$$a.P \xrightarrow{\_\mid \bar{a}.Q} P \mid Q,$$

saying that $a.P$ put into this contexts reacts and reduces to $P \mid Q$. Using all possible contexts as labels would also result in a bisimulation congruence, but we do not gain anything compared to quantification over all contexts.

In [Lei01, LM00] the notion of "minimal context" is formalized as the categorical concept of relative pushout respectively idem pushout. This notion has also been applied to bigraphs [JM03]. However, the theory is complicated by the fact that one can not work with isomorphism classes of graphs, since in this case the category under consideration would not possess all necessary relative pushouts. Thus one is forced to give unique names to all edges and nodes in a graph and to either work in a precategory or to construct a suitable category starting from such a precategory. Another approach, given by Sassone and Sobocinski [SS03], is to work with cells inside a 2-category.

It is our aim to achieve similar results in the context of graph rewriting [Roz97], a framework which allows to model dynamic and concurrent systems consisting of interconnected components in a natural and intuitive way. Many process calculi such as the π-calculus [GM02, MP95, Kön00c] and the ambient calculus [GM01] can be translated into graph rewriting. We are specifically interested in the double-pushout (DPO) approach [CMR$^+$97], one of the standard approaches to graph rewriting. So far, there is not yet a uniform theory of bisimulation for graph transformation systems. Using the concepts explained earlier would be possible in theory, but contradicts the philosophy behind graph rewriting where graphs are considered only up to isomorphism. Furthermore, deriving labels via relative pushouts is entirely non-trivial and can be rather complicated.

The approach which is presented in this paper is motivated by the work of Leifer/Milner and other contributions to this area, but does not directly rely on their theory. Instead we present a very simple way of deriving minimal contexts—we call them borrowed contexts—which smoothly integrates with the DPO approach and which has a very constructive nature. The only categorical concepts that are needed are standard pushouts and pullbacks. The main difference to previous approaches is that in our case graphs are objects and not arrows of the category under consideration. Our arrows instead are graph morphisms which provide the necessary tracking information for nodes and edges which, in the case of graphs as arrows, can only be provided by adding support to a category. This work is based on ideas presented in [Ehr02], a paper which points out similarities and differences between Milner's bigraphs [JM03, Mil01] and the DPO approach to graph rewriting.

Our main result states that bisimilarity defined on graph rewriting with borrowed contexts is indeed a congruence relation (see Theorem 8.4.3).

The paper is structured as follows: In Section 8.2 we will give a short introduction to the DPO approach, followed by the definition of rewriting with borrowed contexts (Section 8.3). Section 8.4 provides the proof ideas that the resulting bisimilarity is a congruence. After having introduced a proof technique we continue with an example showing borrowed contexts at work in Section 8.5.

This paper requires only basic knowledge of category theory [Mac71]. In fact, we only need pushouts and pullbacks, including some general as well as specific preservation, composition and decomposition properties. The general properties hold in any category and the specific ones at least in the category of sets and, as needed in the paper, in the category of graphs. The specific properties are presented in our technical report [EK04b], which also contains the full proof of our main result.

8.2 The DPO Approach to Graph Rewriting

We will first define a family of categories of graphs and graph morphisms, being as general as possible by defining graph structures [EHK+97], which include different forms of graphs such as directed graphs and hypergraphs.

Definition 8.2.1 (Graph Structures) *A* graph structure signature $GS = (S, OP,$
$\Sigma)$ *consists of a set of sorts* S*, a family* $(OP_{s,s'})_{s,s' \in S}$ *of* unary *operator symbols and a family* $(\Sigma_s)_{s\in S}$ *of labelling alphabets.*

A graph structure A *over* GS *is a sort-indexed family* $(A_s)_{s\in S}$ *of carrier sets together with a sort-indexed family of labelling functions* $(l_s^A)_{s\in S}$ *such that* $l_s^A \colon A_s \to \Sigma_s$ *and an* OP*-indexed family of mappings* $(op^A)_{op\in OP}$ *such that* $op^A \colon A_s \to A_{s'}$ *if* $op \in OP_{s,s'}$*.*

A graph structure morphism $\varphi\colon A \to B$ *is a sort-indexed family of mappings* $\varphi = (\varphi_s \colon A_s \to B_s)_{s\in S}$ *such that* $l_s^A(x) = l_s^B(\varphi(x))$ *and* $op^B(\varphi(x)) = \varphi(op^A(x))$ *for all* $x \in A_s$*. A graph structure morphism* φ *is called* injective *if all its mappings are injective. It is an* isomorphism *if all mappings are bijective. An isomorphism of the form* $\varphi\colon A \to A$ *is called* automorphism*.*

The simplest graph structure signature has two sorts: *node* and *edge* and two operator symbols $s, t \in OP_{edge,node}$ standing for "source" and "target". Graph structures over this signature are ordinary labelled directed graphs and graph structure morphisms are standard graph morphisms. The sets Σ_{node} and Σ_{edge} contain node respectively edge labels. In the following we will say "graph" instead of "graph structure" and "graph morphism" or just "morphism" instead of "graph structure morphism".

A category of graphs and graph morphisms has all pushouts and pullbacks, which can be constructed componentwise in the category **Set**. Furthermore, constructing the pushout or pullback of two injective morphisms always gives us two injective morphisms. Working exclusively in the category of injective morphisms is not possible since this category does not have all pushouts and pullbacks, which is due to missing non-injective mediating morphisms. So far we can obtain our main result (Theorem 8.4.3) only if we work with injective morphisms, which is, however, a natural requirement.

Definition 8.2.2 (Graph Transformation System) *A* rule *or* production *is a pair $(\varphi_L\colon I \to L, \varphi_R\colon I \to R)$ of injective graph morphisms. It can be applied to a graph G, resulting in a graph H, if there is an injective match morphism $\varphi\colon L \to G$ and we can find a graph C and morphisms such that the two squares in the following diagram are both pushouts.*

$$\begin{tikzcd} L \arrow[d, "\varphi"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[d] \\ G & C \arrow[l] \arrow[r] & H \end{tikzcd}$$

A graph transformation system *is a set $\mathcal{P}$ of productions.*

The diagram above consisting of two pushouts has led to the name double-pushout or DPO approach. The intuition behind this approach is to find a left-hand side L in a graph G, remove L apart from the interface I and to attach R to the interface in the remaining graph C, resulting in H.

Note: Instead of writing $(\varphi_L\colon I \to L, \varphi_R\colon I \to R)$ we will in the following abbreviate a rule by $(L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R)$, or even $(L \leftarrow I \to R)$ if there is no danger of misunderstanding. This short form will be used for other morphisms as well.

We use a running example throughout the paper which is deliberately kept very simple. Figure 8.1 shows three spans $L \leftarrow I \to R$ which form the rule set $\mathcal{P}$ of our example graph transformation system. The graphs are directed graphs with edge labels where nodes are unlabelled (or are labelled with a dummy label). We give rules for a simplex connection S and a duplex connection D over both of which messages M—represented by a loop—are sent. A duplex connection can be used both ways, whereas a simplex connection has a fixed direction. The connections themselves are preserved and are therefore in the interfaces of the rules. An alternative choice, which is also covered by the concept of graph structures, would have been to model this situation by hypergraphs with unary edges (for messages) and binary edges (for connections).

In order to state congruence results, we first need a notion of contexts and contexualization.

Definition 8.2.3 (Graphs with interfaces and graph contexts) *A graph G with* interface *J is an injective morphism $J \to G$. Furthermore a* context *or* cospan *consists of two injective morphisms $J \to E \leftarrow \overline{J}$.*

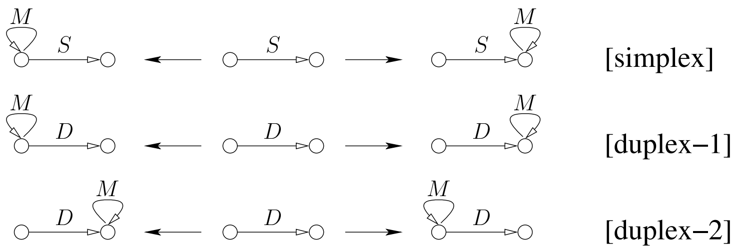


Figure 8.1: Rules of a graph transformation system.

The composition of a graph with interface $J \to G$ and 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 $J \to E$.

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

Note that composition is defined only up to isomorphism, since the pushout object is unique only up to isomorphism.

This notion of interfaces, contexts and composition is within the spirit of the DPO approach where the pushouts for G and H in Definition 8.2.2 can be interpreted as composition of L with C respectively R with C along interface I. In the context of this paper however it is important to consider also the graph G with interface J leading to $\overline{G}$ with interface $\overline{J}$, which requires a context E with two interfaces J and $\overline{J}$. Discrete interfaces, which are a special case, have already been used, see for instance [GH97].

8.3 Rewriting with Borrowed Contexts

We are now ready for the central definition of this paper: graph rewriting with borrowed contexts on graphs with interfaces. The underlying idea is to allow not only total, but also partial matches of a left-hand side. The missing part of the left-hand side is then displayed as the label of the resulting transition.

Definition 8.3.1 (Rewriting with borrowed contexts) *Let $\mathcal{P}$ be a set of graph productions of the form $(L \leftarrow I \to R)$ and let $J \to G$ be a graph with interface. We say that $J \to G$ reduces to $K \to H$ with transition label $(J \to F \leftarrow K)$ whenever there is a production $(L \leftarrow I \to R) \in \mathcal{P}$ and there are graphs D, G^+, C and additional morphisms such that the following diagram commutes and the squares are either pushouts (PO) or pullbacks (PB) with* *injective morphisms.*

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

Symbolically this is denoted by the transition $(J \to G) \overset{J \to F \leftarrow K}{\longrightarrow} (K \to H)$, *which is also called* rewriting step with borrowed context.

The squares in the diagram above have the following meaning: the upper left-hand square merges the left-hand side L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$ of the left-hand side in G. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, which produces the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F which is missing in order to obtain a total match of L, along with a morphism $J \to F$ indicating how F should be attached 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.

The two pushout complements that are constructed in Definition 8.3.1 may not exist. The middle square in the upper row can only be completed if the dangling edge condition is satisfied, i.e., if the left-hand side L is connected to the rest of the graph G^+ exclusively via its interface I and no edges would be left "dangling" by removing it. The left square in the lower row can only be completed if there is a way to extend the partial match to a left-hand side L by attaching some context $J \to F$ to $J \to G$. In other words, the dangling edge condition is required also for the morphism $G \to G^+$ with respect to the interface morphism $J \to G$.

In this case the borrowed context F is minimal in the following sense: Given the partial match $G \leftarrow D \to L$, the pushout G^+ is the minimal graph containing both G and L attached according to the partial match. The borrowed context F is a pushout complement of the injective morphisms $J \to G \to G^+$, leading to the injective morphisms $J \to F \to G^+$. This implies that F is the unique graph (up to isomorphism) that is needed to extend G to the minimal graph G^+.

From the properties of the category of graph structures we can infer that all morphisms in the diagram above are injective. It is thus possible to draw a schematic representation of the four left-hand side squares of Definition 8.3.1 (see Figure 8.2). This figure also illustrates that the new interface K is the "union" of the interfaces I and J, minus the graph components that are internal in either G or L.

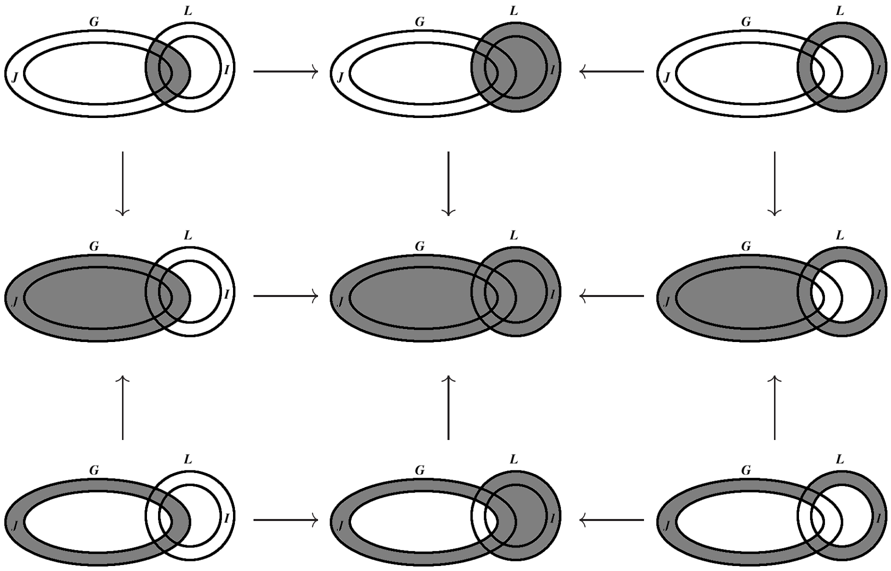


Figure 8.2: Graphical represenation of rewriting with borrowed contexts.

In order to illustrate Definition 8.3.1, we regard rule [simplex] of Figure 8.1 and an example graph G consisting of two S-edges for which we find a partial match of the left-hand side. This results in the derivation shown in Figure 8.3. Note that the image of a node under a morphism is implicitly given by its position, i.e., the left-hand node is always mapped to a left-hand node, analogously for the right-hand node.

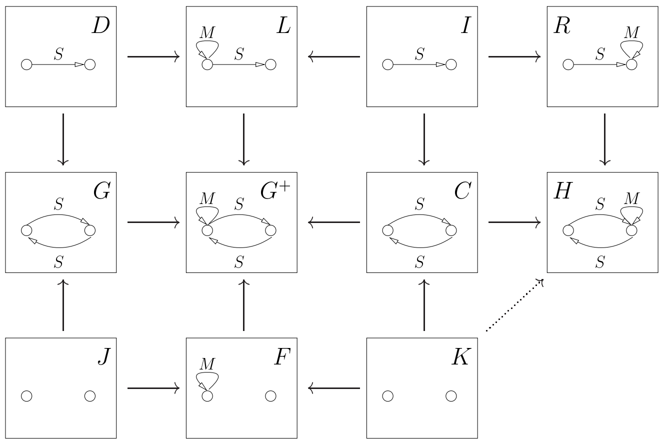


Figure 8.3: Rewriting with borrowed contexts in the example graph transformation system.

8.4 Bisimilarity is a Congruence

We now arrive at the main theorem of this paper: We will show that the bisimilarity defined on labelled graph transition systems is a congruence. Before that we need two more definitions.

Definition 8.4.1 (Bisimulation and Bisimilarity) *Let $\mathcal{P}$ be a set of productions. Let $\mathcal{R}$ be a symmetric relation containing pairs of graphs with interfaces of the form $(J \to G, J \to G')$, also written $(J \to G)\,\mathcal{R}\,(J \to G')$.*

The relation $\mathcal{R}$ is a bisimulation *if whenever we have $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition $(J \to G) \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H)$ (in words: $J \to G$ reduces to $K \to H$ with transition label $J \to F \leftarrow K$) can be derived from $\mathcal{P}$, then there exists a morphism $K \to H'$ and a transition $(J \to G') \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H')$ with the same transition label $J \to F \leftarrow K$ 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 morphisms. The relation $\sim$ is called bisimilarity*.*

In order to state Theorem 8.4.3, we have to be able to close a bisimulation or simply a relation under all possible contexts.

Definition 8.4.2 (Closure under Contextualization) *Let $\mathcal{R}$ be a relation on graphs with interfaces as in Definition 8.4.1. By $\hat{\mathcal{R}}$ we denote the closure of $\mathcal{R}$ under contextualization, 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 morphisms $(\overline{J} \to \overline{G}, \overline{J} \to \overline{G}')$ which results from the composition of $J \to G$ and $J \to E \leftarrow \overline{J}$ respectively $J \to G'$ and $J \to E \leftarrow \overline{J}$.*

A relation $\mathcal{R}$ is a congruence, i.e., closed under contexts whenever $\hat{\mathcal{R}} = \mathcal{R}$. Since obviously $\mathcal{R}$ is contained in $\hat{\mathcal{R}}$, it suffices to show $\hat{\mathcal{R}} \subseteq \mathcal{R}$. We only give a proof sketch, the full proof can be found in [EK04b].

Theorem 8.4.3 (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 (Sketch):

Remark: In this proof we are using properties of the category of graph structures, such as pushout complement splitting and special decomposition properties, that do not necessarily hold in other categories (cf. the remarks at the end of the introduction).

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

So let $\mathcal{R}$ be a bisimulation and let $(\overline{J} \to \overline{G})\,\hat{\mathcal{R}}\,(\overline{J} \to \overline{G}')$. We assume that

$$(\overline{J} \to \overline{G}) \stackrel{\overline{J} \to \overline{F} \leftarrow \overline{K}}{\longrightarrow} (\overline{K} \to \overline{H}).$$

Our goal is to show that there exists a transition

$$(\overline{J} \to \overline{G}') \overset{\overline{J} \to \overline{F} \leftarrow \overline{K}}{\longrightarrow} (\overline{K} \to \overline{H}')$$

with $(\overline{K} \to \overline{H}) \,\hat{\mathcal{R}}\, (\overline{K} \to \overline{H}')$, which implies that $\hat{\mathcal{R}}$ is a bisimulation. In Step 1 we will construct a transition $(J \to G) \overset{J \to F \leftarrow K}{\longrightarrow} (K \to H)$ which implies a transition $(J \to G') \overset{J \to F \leftarrow K}{\longrightarrow} (K \to H')$ with $(K \to H) \,\mathcal{R}\, (K \to H')$, since $\mathcal{R}$ is a bisimulation. In Step 2 we will extend the second transition to obtain the transition stated in our goal above.

Step 1: Our first assumption $(\overline{J} \to \overline{G}) \,\hat{\mathcal{R}}\, (\overline{J} \to \overline{G}')$ means that there is some pair $(J \to G) \,\mathcal{R}\, (J \to G')$ and a context $J \to E \leftarrow \overline{J}$ such that $\overline{J} \to \overline{G}$ and $\overline{J} \to \overline{G}'$ can be obtained by composing $J \to G$ and $J \to G'$ with this context. The second assumption is the transition $(\overline{J} \to \overline{G}) \overset{\overline{J} \to \overline{F} \leftarrow \overline{K}}{\longrightarrow} (\overline{K} \to \overline{H})$ which leads to the situation depicted in Diagram (8.1), where the decomposition of $\overline{J} \to \overline{G}$ is shown explicitly and all morphisms are injective and all (basic) squares are pushouts, apart from the square $\overline{K}$, $\overline{C}$, $\overline{F}$, $\overline{G}^+$, which is a pullback.

$$\begin{tikzcd}
 & \overline{D} \arrow[r] \arrow[d] & L \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & \overline{G} \arrow[r] & \overline{G}^+ & \overline{C} \arrow[l] \arrow[r] & \overline{H} \\
J \arrow[u] \arrow[r] & E \arrow[u] & & & \\
 & \overline{J} \arrow[u] \arrow[r] & \overline{F} \arrow[uu] & \overline{K} \arrow[l] \arrow[uu] \arrow[uur, dotted] &
\end{tikzcd} \tag{8.1}$$

$$\begin{tikzcd}
 & \overline{D} \arrow[r] \arrow[d] & L \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[r] & \overline{G} \arrow[r] & \overline{G}^+ & \overline{C} \arrow[l] \arrow[r] & \overline{H} \\
J \arrow[u] \arrow[r] & E \arrow[u] \arrow[r] & E_2 \arrow[u] & E_1 \arrow[l] \arrow[u] \arrow[ur, dotted] & \\
 & \overline{J} \arrow[u] \arrow[r] & \overline{F} \arrow[u] & \overline{K} \arrow[l] \arrow[u] \arrow[uur, dotted] &
\end{tikzcd} \tag{8.2}$$

We can now split the lower pushout and the lower pullback along E (see Diagram (8.2)).

As a next step we construct D as the pullback of $G \to \overline{G}$ and $\overline{D} \to \overline{G}$, followed by the construction of $\tilde{G}$ as the pushout of the resulting morphisms. In Diagram (8.2) we can now split the upper row of pushouts and the pushout to the very left, obtaining the graphs F_1, G^+, C and H. We then construct F as the pullback of $G^+ \to \overline{G}^+$ and $E_2 \to \overline{G}^+$ and K as pullback of the morphisms $C \to \overline{C}$ and $E_1 \to \overline{C}$. This results in Diagram (8.3), with two commuting cubes in the middle of the diagram.

All morphisms are injective, all squares commute and we can infer that the squares D, G, L, G^+ and I, L, C, G^+ and J, G, F, G^+ and I, R, C, H are pushouts and the square K, C, F, G^+ is a pullback, as in Definition 8.3.1. Hence, from Diagram (8.3) we can derive the following transition:

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

using the notation of Definition 8.3.1. Since $\mathcal{R}$ is a bisimulation, this implies

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

with $(K \to H) \,\mathcal{R}\, (K \to H')$. Furthermore we can infer from Diagram (8.3) that $\overline{K} \to \overline{H}$ can be obtained by composing $K \to H$ with the context $K \to E_1 \leftarrow \overline{K}$, since the square $K, H, E_1, \overline{H}$ is a pushout.

$$
\begin{tikzcd}
D \arrow[rr] \arrow[dd] & & \overline{D} \arrow[rr] \arrow[dl] \arrow[dd] & & L \arrow[dl] \arrow[dd] & & I \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R \arrow[dl] \arrow[dd] \\
& \tilde{G} \arrow[rr] \arrow[dr] & & G^+ \arrow[dr] & & C \arrow[ll] \arrow[rr] \arrow[dr] & & H \arrow[dr] & \\
G \arrow[rr] \arrow[ur] & & \overline{G} \arrow[rr] & & \overline{G}^+ & & \overline{C} \arrow[ll] \arrow[rr] & & \overline{H} \\
& F_1 \arrow[rr] \arrow[uu] \arrow[dr] & & F \arrow[uu] \arrow[dr] & & K \arrow[ll] \arrow[uu] \arrow[dr] \arrow[uurr, dotted] & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[uu] \arrow[rr] & & E_2 \arrow[uu] & & E_1 \arrow[ll] \arrow[uu] \arrow[uurr, dotted] & & \\
& & \overline{J} \arrow[rr] \arrow[u] & & \overline{F} \arrow[u] & & \overline{K} \arrow[ll] \arrow[u] \arrow[uuurr, dotted] & &
\end{tikzcd}
\tag{8.3}
$$

Step 2: We will now extend the transition from $J \to G'$ to $K \to H'$ with $(K \to H)\,\mathcal{R}\,(K \to H')$ obtained above to construct a transition from $(\overline{J} \to \overline{G}')$ to $(\overline{K} \to \overline{H}')$ with $(\overline{K} \to \overline{H})\,\hat{\mathcal{R}}\,(\overline{K} \to \overline{H}')$. We will construct $\overline{K} \to \overline{H}'$ in such a way that it is the composition of $K \to H'$ with the context $K \to E_1 \leftarrow \overline{K}$. Recall also that $\overline{J} \to \overline{G}'$ is the composition of $J \to G'$ and the context $J \to E \leftarrow \overline{J}$.

We now cut away the upper layer of Diagram (8.3) and we obtain Diagram (8.4) where all squares are pushouts, apart from the square $\overline{K}, \overline{F}, E_1, E_2$, which is a pullback.

$$
\begin{tikzcd}
& F_1 \arrow[rr] \arrow[dr] & & F \arrow[dr] & & K \arrow[ll] \arrow[dr] & \\
J \arrow[ur] \arrow[rr] & & E \arrow[rr] & & E_2 & & E_1 \arrow[ll] \\
& & \overline{J} \arrow[rr] \arrow[u] & & \overline{F} \arrow[u] & & \overline{K} \arrow[ll] \arrow[u]
\end{tikzcd}
\tag{8.4}
$$

From the derivation step of $J \to G'$ given earlier one can derive Diagram (8.5) for some rule $L' \leftarrow I' \to R'$ where the lower right-hand square is a pullback and all other squares are pushouts. The morphism $J \to F$ is split by F_1 and therefore we can split the two left-hand side pushouts as shown in Diagram (8.6).

$$
\begin{tikzcd}
D' \arrow[r] \arrow[d] & L' \arrow[d] & I' \arrow[l] \arrow[r] \arrow[d] & R' \arrow[d] \\
G' \arrow[r] & G'^+ & C' \arrow[l] \arrow[r] & H' \\
J \arrow[r] \arrow[u] & F \arrow[u] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}
\tag{8.5}
$$

$$
\begin{tikzcd}
D' \arrow[r] \arrow[d] & \overline{D}' \arrow[r] \arrow[d] & L' \arrow[d] & I' \arrow[l] \arrow[r] \arrow[d] & R' \arrow[d] \\
G' \arrow[r] & \tilde{G}' \arrow[r] & G'^+ & C' \arrow[l] \arrow[r] & H' \\
J \arrow[r] \arrow[u] & F_1 \arrow[r] \arrow[u] & F \arrow[u] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}
\tag{8.6}
$$

We compose Diagrams (8.4) and (8.6) and construct the graph $\overline{G}'$ as the pushout of $F_1 \to \tilde{G}'$ and $F_1 \to E$, the graph $\overline{G}'^{+}$ as the pushout of $F \to G'^{+}$ and $F \to E_2$ and the graph $\overline{C}'$ as pushout of $K \to C'$ and $K \to E_1$. This results in Diagram (8.7), which is identical to Diagram (8.3) in structure.

$$\begin{tikzcd}
D' \arrow[rr] \arrow[dd] & & \overline{D}' \arrow[rr] \arrow[dl] \arrow[dd] & & L' \arrow[dl] \arrow[dd] & & I' \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R' \arrow[dl] \arrow[dd] \\
& \tilde{G}' \arrow[rr] \arrow[dr] & & G'^{+} \arrow[dr] & & C' \arrow[ll] \arrow[rr] \arrow[dr] & & H' \arrow[dr] & \\
G' \arrow[ur] \arrow[rr] & & \overline{G}' \arrow[rr] & & \overline{G}'^{+} & & \overline{C}' \arrow[ll] \arrow[rr] & & \overline{H}' \\
& F_1 \arrow[uu] \arrow[rr] \arrow[dr] & & F \arrow[uu] \arrow[dr] & & K \arrow[uu] \arrow[ll] \arrow[dr] \arrow[uurr, dotted] & & & \\
J \arrow[uu] \arrow[ur] \arrow[rr] & & E \arrow[uu] \arrow[rr] & & E_2 \arrow[uu] & & E_1 \arrow[uu] \arrow[ll] \arrow[uurr, dotted] & & \\
& & \overline{J} \arrow[u] \arrow[rr] & & \overline{F} \arrow[u] & & \overline{K} \arrow[u] \arrow[ll] \arrow[uuurr, dotted] & &
\end{tikzcd} \tag{8.7}$$

In Diagram (8.7), the three right-hand squares in the upper row are all pushouts, the square $\overline{J}, \overline{F}, \overline{G}', \overline{G}'^{+}$ is a pushout and the square $\overline{K}, \overline{F}, \overline{C}', \overline{G}'^{+}$ is a pullback.

Hence, by Definition 8.3.1 we infer that

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

and since the square $K, H', E_1, \overline{H}'$ is also a pushout we can infer that $\overline{K} \to \overline{H}'$ can be obtained by composing $K \to H'$ and the context $K \to E_1 \leftarrow \overline{K}$. From earlier considerations we know that $\overline{K} \to \overline{H}$ is the composition of $K \to H$ with $K \to E_1 \leftarrow \overline{K}$ and hence $(\overline{K} \to \overline{H})\,\hat{\mathcal{R}}\,(\overline{K} \to \overline{H}')$. This means that we have achieved our goal stated at the beginning of the proof sketch, which implies that $\hat{\mathcal{R}}$ is a bisimulation and $\sim$ is a congruence.

□

8.5 Borrowed Contexts at Work: An Example

In order to further pursue the example we will first introduce a proof technique simplifying bisimilarity proofs. This technique is a straightforward instance of an up-to technique [San95]. The underlying idea behind the technique is the observation that the relation $\mathcal{R}$ should be as small as possible, in order to obtain a compact proof. This goal can be reached by slightly extending the notion of bisimulation: We now demand that if a transition is matched by another, the pair of resulting graphs can be found in $\mathcal{R}$ after removal of identical contexts. Hence, this extended notion of bisimulation is called "bisimulation up to context". We first need an auxiliary definition.

Definition 8.5.1 (Progression) *Let $\mathcal{R}, \mathcal{S}$ be relations containing pairs of graphs 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 whenever $(J \to G)\, \mathcal{R}\, (J \to G')$ and $(J \to G) \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H)$, there exists a morphism $K \to H'$ such that $(J \to G') \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H')$ and $(K \to H)\, \mathcal{S}\, (K \to H')$.*

For example, $\mathcal{R}$ is a bisimulation if and only if $\mathcal{R} \rightarrowtail \mathcal{R}$.

Definition 8.5.2 (Bisimulation up to Context) *Let $\mathcal{R}$ be a symmetric relation containing pairs of graphs with interfaces of the form $(J \to G, J \to G')$.*

If $\mathcal{R} \rightarrowtail \hat{\mathcal{R}}$, then $\mathcal{R}$ is called bisimulation up to context.

We will show in Proposition 8.5.3 that every bisimulation up to context is contained in the bisimilarity $\sim$. The attractiveness of bisimulations up to context stems from the fact that such a relation can be much smaller than the least bisimulation that contains it and thus proofs can be compressed. This technique might even allow us to work with a finite relation instead of an infinite one.

Proposition 8.5.3 (Bisimulation up to Context implies Bisimilarity) *Let $\mathcal{R}$ be a bisimulation up to context. Then it holds that $\mathcal{R} \subseteq \sim$.*

Proof (Sketch): By carefully examining the proof of Theorem 8.4.3 again we can see that some simple modifications give us the following (stronger) theorem:

If $\mathcal{R} \rightarrowtail \mathcal{S}$, then also $\hat{\mathcal{R}} \rightarrowtail \hat{\mathcal{S}}$.

Since $\mathcal{R}$ is a bisimulation up to context we have $\mathcal{R} \rightarrowtail \hat{\mathcal{R}}$. The stronger version of Theorem 8.4.3 now implies $\hat{\mathcal{R}} \rightarrowtail \widehat{(\hat{\mathcal{R}})}$. Since the composition of contexts is associative we have $\widehat{(\hat{\mathcal{R}})} = \hat{\mathcal{R}}$, which implies $\hat{\mathcal{R}} \rightarrowtail \hat{\mathcal{R}}$ and hence that $\hat{\mathcal{R}}$ is a bisimulation, i.e., $\hat{\mathcal{R}} \subseteq \sim$. This implies $\mathcal{R} \subseteq \hat{\mathcal{R}} \subseteq \sim$. $\square$

Since contextualization is defined only up to isomorphism, we can assume that $\hat{\mathcal{R}}$ is closed under isomorphism in the following sense: For every span $G \leftarrow J \to G'$, all isomorphic spans $\tilde{G} \leftarrow \tilde{J} \to \tilde{G}'$ are also contained in $\hat{\mathcal{R}}$.

Similarly, we can restrict ourselves to abstract transitions when checking for bisimilarity: Assume that $(J \to G)\, \mathcal{R}\, (J \to G')$ and there are two transitions

$$(J \to G) \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H) \quad \text{and} \quad (J \to G) \stackrel{J \to \tilde{F} \leftarrow \tilde{K}}{\longrightarrow} (\tilde{K} \to \tilde{H})$$

with isomorphisms from $\tilde{F}, \tilde{K}, \tilde{H}$ to F, K, H respectively such that the entire diagram commutes (see Diagram (8.8)). It is sufficient to show the existence of a transition $(J \to G') \stackrel{J \to F \leftarrow K}{\longrightarrow} (K \to H')$ such that $(K \to H)\, \hat{\mathcal{R}}\, (K \to H')$. From this transition and Diagram (8.8) we can derive Diagram (8.9), where the arrows pointing upwards are isomorphisms and the diagram commutes. In such a situation we can infer the existence of a transition $(J \to G') \stackrel{J \to \tilde{F} \leftarrow \tilde{K}}{\longrightarrow} (\tilde{K} \to \tilde{H}')$ such that $H \leftarrow K \to H'$ and $\tilde{H} \leftarrow \tilde{K} \to \tilde{H}'$ are isomorphic spans, from which it follows that $(\tilde{K} \to \tilde{H})\, \hat{\mathcal{R}}\, (\tilde{K} \to \tilde{H}')$.

$$
\begin{tikzcd}
 & & F & K \arrow[l] \arrow[r] & H \\
G & J \arrow[l] \arrow[ur] \arrow[dr] & & & \\
 & & \hat{F} \arrow[uu, "\wr"] & \hat{K} \arrow[l] \arrow[r] \arrow[uu, "\wr"] & \hat{H} \arrow[uu, "\wr"]
\end{tikzcd}
\tag{8.8}
$$

$$
\begin{tikzcd}
 & & F & K \arrow[l] \arrow[r] & H' \\
G' & J \arrow[l] \arrow[ur] \arrow[dr] & & & \\
 & & \hat{F} \arrow[uu, "\wr"] & \hat{K} \arrow[l] \arrow[r] \arrow[uu, "\wr"] & \hat{H}' \arrow[uu, "\wr"]
\end{tikzcd}
\tag{8.9}
$$

We now show how to exploit this proof technique and prove that two graphs are bisimilar. We assume that the set $\mathcal{P}$ of rules depicted in Figure 8.1 is given and we consider the two graphs with interfaces of Figure 8.4.

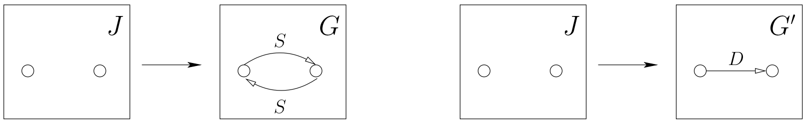


Figure 8.4: Two graphs with interfaces which are bisimilar.

We consider the symmetric relation

$$\mathcal{R} = \{(J \to G, J \to G'), (J \to G', J \to G)\}$$

and we will show that it is a bisimulation up to context. For each of the three rules there are several partial matches for both G and G'. Most of these matches are not very interesting, since the graph to be rewritten and the left-hand side overlap only in their interfaces, but the corresponding transitions have to be checked nevertheless. (We discuss possible simplifications in the conclusion.)

In order to give a general idea, we consider only two transitions of $J \to G$ in detail, where both are instances of rule [simplex]. These two transitions will be written

$$(J \to G) \stackrel{J \to F_i \leftarrow K_i}{\longrightarrow} (K_i \to H_i),$$

where $i = 1, 2$. The graphs F_i, K_i, H_i and the corresponding morphisms are depicted in Figure 8.5. Note that we have already shown how to derive the first transition of $J \to G$ in Figure 8.3 (where $F_1 = F$, $K_1 = K$, $H_1 = H$).

In order to show that $\mathcal{R}$ is a bisimulation up to context, we have to find matching transitions

$$(J \to G') \stackrel{J \to F_i \leftarrow K_i}{\longrightarrow} (K_i \to H_i')$$

for $i = 1, 2$, such that $(K_i \to H_i) \,\hat{\mathcal{R}}\, (K_i \to H_i')$. Such transitions can be derived and the graphs H_i' with their corresponding morphisms are also depicted in Figure 8.5. Note that the first transition is an instance of rule [duplex-1], while the second transition is an instance of rule [duplex-2].

Furthermore, it holds that $(K_i \to H_i) \,\hat{\mathcal{R}}\, (K_i \to H_i')$ for $i = 1, 2$, since these graphs can be obtained by composing $J \to G$ respectively $J \to G'$ with a context consisting of two nodes and a looping M-edge. After checking also the remaining transitions we can conclude $(J \to G) \sim (J \to G')$ from Proposition 8.5.3.

This means that in every context we can replace a duplex connection by two simplex connections and vice versa. Even this small example shows us that

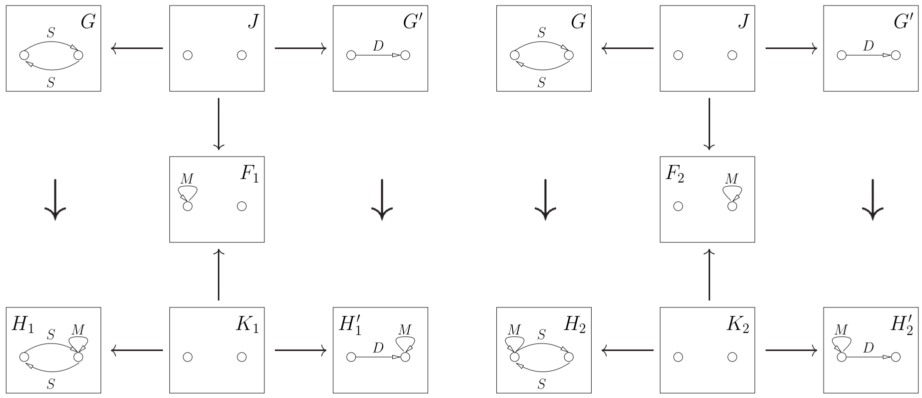


Figure 8.5: Matching transitions of bisimilar graphs.

in order to obtain a bisimilarity result, proof techniques are needed in order to keep $\mathcal{R}$ finite. Otherwise we would have to deal with a relation containing infinitely many elements.

8.6 Conclusion

We have presented a way to derive labelled transitions and bisimulation congruences for graph transformation systems. It is our hope that this work will be helpful for the transfer of concepts from the world of process algebras to the world of graph rewriting and vice versa. We believe that having graphs as objects (and graph morphisms as arrows) instead of having graphs as arrows is useful for tracking graph components and thus enables us to easily state which components are associated with each other in different graphs. Hence we need not consider explicit names for graph components.

We have made some investigations concerning the adaptation of the concept of relative pushouts for cospans of graphs. However, there are fundamental problems, mainly caused by graphs having non-trivial automorphisms (see, e.g., the counterexample in [Lei01] on pages 80/81, which can be directly transferred into our framework). We believe, however, that our construction is very close in spirit to the notion of relative pushouts introduced by Leifer and Milner and that it should be possible to show the equivalence of these two notions in a suitable graph category with support.

Our results do not only hold in graph structure categories, but also in other categories which satisfy certain properties typical for the categories of sets and high-level replacement systems [EGPP99]. In this context it is also interesting to point out that most of the categorical properties we need hold already in adhesive categories [LS04].

In the future we also plan to address the following two questions: How should weak bisimilarity be defined and is it a congruence? Do our results still hold if we allow for non-injective morphisms? Furthermore we plan to introduce more proof techniques in order to simplify bisimulation proofs. One such technique is clearly suggested by the example in Section 8.5. Whenever a graph and a left-hand side overlap only in their interfaces, another graph with the same interface will certainly be able to match the corresponding rewriting step with borrowed context, since this step only changes the interface itself. Hence it should be possible to remove some superfluous transitions without changing the underlying bisimilarity.

Another interesting question would be to find out which bisimulation congruences are produced by the various encodings of π-calculus into graph rewriting and to see in what way they are related to existing congruences for this calculus. It also remains to determine in what way our bisimilarity is related to dynamic bisimulation as presented in [BMS00, MS92c].

Acknowledgements: We would like to thank the anonymous referees for their helpful comments. We are especially grateful to Robin Milner for suggesting this research topic and for sharing his ideas with us.

Chapter 9

Observational Equivalence for Synchronized Graph Rewriting with Mobility

(Joint work with Ugo Montanari)

Abstract

We introduce a notion of bisimulation for graph rewriting systems, allowing us to prove observational equivalence for dynamically evolving graphs and networks.
We use the framework of synchronized graph rewriting with mobility which we describe in two different, but operationally equivalent ways: on graphs defined as syntactic judgements and by using tile logic. One of the main results of the paper says that bisimilarity for synchronized graph rewriting is a congruence whenever the rewriting rules satisfy the basic source property. Furthermore we introduce an up-to technique simplifying bisimilarity proofs and use it in an example to show the equivalence of a communication network and its specification.

9.1 Introduction

Graph rewriting can be seen as a general framework in which to specify and reason about concurrent and distributed systems [EKMR99]. The topology and connection structure of these systems can often be naturally represented in terms of nodes and connecting edges, and their dynamic evolution can be expressed by graph rewriting rules. We are specifically interested in hypergraphs where an arbitrarily long sequence of nodes—instead of a pair of nodes—is assigned to every edge.

However, the theory of graph rewriting [Roz97] lacks a concept of observational equivalence, relating graphs which behave the same in all possible context, which is quite surprising, since observational equivalences, such as bisimilarity or trace equivalence, are a standard tool in the theory of process calculi.

We are therefore looking for a semantics for (hyper-)graph rewriting systems that abstracts from the topology of a graph, and regards graphs as processes which are determined by their interaction with the environment, rather than by their internal structure. It is important for the observational equivalence to be a congruence, since this will enable compositional proofs of equivalence and assure substitutivity, i.e. that equivalent subcomponents of a system are exchangeable.

The applications we have in mind are the verification of evolving networks, consisting, e.g., of processes, messages and other components. A possible scenario would be a user who has limited access to a dynamically changing network. We want to show that the network is transparent with respect to the location of resources, failure of components, etc. by showing that it is equivalent to a simpler specification. Such an equivalence of networks is treated in the example in Section 9.7.

One possible (and well-studied) candidate for an observational equivalence is bisimilarity. So the central aim of this paper is to introduce bisimilarity for graph rewriting—we will explain below why it is convenient to base this equivalence on the model of synchronized graph rewriting, as opposed to other models—and to introduce an up-to proof technique, simplifying actual proofs of bisimilarity.

When defining bisimulation and bisimilarity for graph rewriting systems, two possibilities come to mind: the first would be to use unlabelled context-sensitive rewrite rules as, for example, in the double-pushout approach [Ehr79]. The definition of an observational congruence in this context, however, ordinarily requires universal quantification over all possible contexts of an expression, which is difficult to handle in practice. This makes us choose the second possibility, which is closer to process algebras: we use synchronized graph rewriting, which allows only context-free rewrite rules whose expressive power is increased considerably by adding synchronization and mobility (i.e. communication of nodes), thus including a large class of rewriting systems. In this case we can define a simple syntactic condition (the basic source property) on the rewrite rules ensuring that bisimilarity is a congruence (compare with the de Simone format [dS85] and the `tyft/tyxt`-format [GV92]).

As synchronization mechanism we choose Hoare synchronization which means that all edges that synchronize via a specific node produce the same synchronization action. This is different from Milner synchronization (as in CCS [Mil80]) where two synchronizing processes produce two different signals: an action a and a coaction $\bar{a}$.

We prefer Hoare synchronization since it makes it easier to handle the kind of multiple synchronization we have in mind: several edges connected to each other on a node must agree on an action a, which means that there is no clear distinction between action and coaction. This, in turn, causes other nodes connected to the same edges to perform a different action, and in this way synchronization is propagated by edges and spreads throughout an entire connected component. It is conceivable to implement different synchronization mechanisms as processes working as "connectors", thus modeling in this way a variety of coordination mechanisms.

Edges synchronizing with respect to an action a may, at the same time, agree to create new nodes which are then shared among the right-hand sides of the respective rewrite rules. (This form of mobility was first presented in [HIM00] and is also extensively treated in [HM01].) From the outside it is not possible to determine whether newly created nodes are different or equal, it is only possible to observe the actions performed.

Apart from the obvious representation of graphs in terms of nodes and edges, there are several other approaches representing graphs by terms, which allow for a more compositional presentation of graphs and enable us, for example, to do induction on the graph structure. We will introduce two of these term representations: first graphs as syntactic judgements, where nodes are represented by names and we have operators such as parallel composition and name hiding at our disposal. This representation allows for a straightforward definition of graph rewriting with synchronization and mobility.

The second representation defines graphs in terms of arrows of a P-monoidal category [BGM98]. This allows for an easy presentation of graph rewriting in tile logic, a rewriting framework which deals with the rewriting of open terms that can still be contextualized and instantiated and allows for different ways of composing partial rewrites. To show the compositionality of our semantics, we use a property of tile logic, i.e. the fact that if a tile system satisfies a so-called decomposition property, then bisimilarity defined on top of this tile system is a congruence (see also [BFEMOM00a]).

Apart from the fact that we use tile logic as a tool to obtain the congruence result, we also show how mobility, and specifically the form of mobility used in synchronized graph rewriting, can be handled in the context of tile logic.

9.2 Synchronized Graph Rewriting with Mobility

We start by introducing a representation of (hyper-)graphs as syntactic judgements, where nodes in general correspond to names, external nodes to free names and (hyper-)edges to terms of the form $s(x_1, \ldots, x_n)$ where the x_i are arbitrary names.

Definition 9.2.1 (Graphs as Syntactic Judgements) *Let N be a fixed infinite set of names. A* syntactic judgement *is of the form $\Gamma \vdash G$ where $\Gamma \subseteq N$ is a set of names (the interface of the graph) and G is generated by the grammar*

$$\begin{array}{rcl} G & ::= & \textit{nil (empty graph)} \mid G|G \textit{ (parallel composition)} \mid \\ & & (\nu x)G \textit{ (node hiding)} \mid s(x_1, \ldots, x_n) \textit{ (edge)} \end{array}$$

where $x \in N$ and $s(x_1, \ldots, x_n)$ with arbitrary $x_i \in N$ is called an edge of arity n labelled s. (Every label is associated with a fixed arity.)

Let $fn(G)$ denote the set of all free names of G, i.e. all names not bound by a ν-operator. We demand that $fn(G) \subseteq \Gamma$.

We assume that whenever we write Γ, x, then x is not an element of Γ.

We need to define a structural congruence on syntactic judgements in order to identify those terms that represent isomorphic graphs (up to isolated nodes) (see [Kön99a, Kön00c]).

Definition 9.2.2 (Structural Congruence) *Structural congruence* $\equiv$ *on syntactic judgements obeys the rules below and is closed under parallel composition* $|$ *and the hiding operator* ν. *(We abbreviate equations of the form* $\Gamma \vdash G \equiv \Gamma \vdash G'$ *by* $G \equiv G'$.*)*

$$\Gamma \vdash G \equiv \rho(\Gamma) \vdash \rho(G) \;\; \textit{where } \rho \textit{ is an injective substitution}$$

$$(G_1|G_2)|G_3 \equiv G_1|(G_2|G_3) \qquad G_1|G_2 \equiv G_2|G_1 \qquad G|nil \equiv G$$

$$(\nu x)(\nu y)G \equiv (\nu y)(\nu x)G \qquad (\nu x)nil \equiv nil \qquad (\nu x)G \equiv (\nu y)G\{y/x\} \;\; \textit{if } y \notin \mathit{fn}(G)$$

$$(\nu x)(G|G') \equiv (\nu x)G|G' \;\; \textit{if } x \notin \mathit{fn}(G')$$

We sometimes abbreviate $(\nu x_1)\dots(\nu x_n)G$ by $(\nu\{x_1,\dots,x_n\})G$.

Example 9.2.3 We regard the syntactic judgement $y \vdash (\nu x)(\nu z)(P(x) \mid S(x,y,z) \mid P(z))$ which consists of two processes P which are connected to each other and the only external node y via a switch S. A graphical representation of this syntactic judgement can be found in Figure 9.2 (graph in the lower left corner).

In order to define rewriting on syntactic judgements we introduce the notion of rewriting rule. We use a set *Act* of arbitrary actions, which can be thought of as the set of signals which are allowed in a network.

Definition 9.2.4 (Rewriting Rules) *Let Act be a set of actions, containing also the idle action* ε. *Each action* $a \in Act$ *is associated with an arity* $ar(a) \in \mathbb{N}$, *the arity of* ε *is* 0. *(The arity indicates the number of nodes created by an action.)*

A rewriting rule is of the form

$$x_1,\dots,x_n \vdash s(x_1,\dots,x_n) \xrightarrow{\Lambda} x_1,\dots,x_n,\Gamma_\Lambda \vdash G$$

where all x_i *are distinct,* $\Lambda \subseteq \{x_1,\dots,x_n\} \times Act\backslash\{\varepsilon\} \times N^*$ *such that* Λ *is a total function in its first argument, i.e. if* $(x_i,a_i,\tilde{y}_i) \in \Lambda$ *we write* $\Lambda(x_i) = (a_i,\tilde{y}_i)$, *respectively* $act_\Lambda(x_i) = a_i$ *and* $n_\Lambda(x_i) = \tilde{y}_i$, *and we demand that* $ar(a_i) = |\tilde{y}_i|$.

Furthermore[1] $\Gamma_\Lambda = \bigcup_{x_i \in \Lambda} Set(n_\Lambda(x_i))$ *and we demand that* $\{x_1,\dots,x_n\} \cap \Gamma_\Lambda = \emptyset$.

A rewriting rule of the form given above indicates that an edge $s(x_1,\dots,x_n)$ is rewritten, synchronizing on each node x_i with an action a_i, and during this synchronization a string $\tilde{y}_i$ of new nodes is created. The set Γ_Λ contains all new nodes in the interface which are created by the rewriting step.

The following example will be used as a running example throughout the paper.

[1]For any string $\tilde{s}$, we denote the set of its elements by $Set(\tilde{s})$.

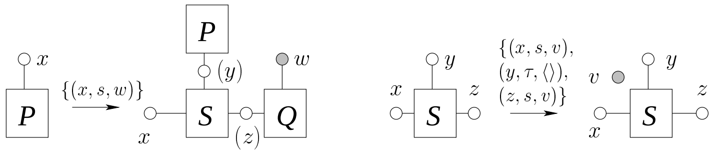


Figure 9.1: Graphical representation of example rules

Example 9.2.5 We describe a network of processes P of arity 1 and processes Q of arity 2 connected to each other via switches S of arity 3.

We use three kinds of actions, apart from the idle action ε: τ and a (both of arity 0) and s (of arity 1) which is the action used for establishing a shared name. A process of our example network can perform the following rewriting steps:[2] P can either send a signal a, or it can extend the network by transforming itself into a switch with two processes connected to it, or it can perform an s action and fork a process Q whose second node is connected to a newly created, privately shared channel. The action τ is different from the idle action and is used in this example to represent internal activity.

$$\begin{array}{lcl} x \vdash P(x) & \stackrel{\{(x,a,\langle\rangle)\}}{\longrightarrow} & x \vdash P(x) \\ y \vdash P(y) & \stackrel{\{(y,\tau,\langle\rangle)\}}{\longrightarrow} & y \vdash (\nu x)(\nu z)(P(x) \mid S(x,y,z) \mid P(z)) \\ x \vdash P(x) & \stackrel{\{(x,s,w)\}}{\longrightarrow} & x, w \vdash (\nu y)(\nu z)(S(x,y,z) \mid P(y) \mid Q(z,w)). \end{array}$$

The process Q, on the other hand, can perform any combination of a and τ actions.

$$x, y \vdash Q(x,y) \stackrel{\{(x,a_1,\langle\rangle),(y,a_2,\langle\rangle)\}}{\longrightarrow} x, y \vdash Q(x,y) \quad \text{where } a_1, a_2 \in \{a, \tau\}.$$

Switches have the task to route the signals and actions originating at the processes and in the case of an s action a new node v is created. In both rules we require that $\{x,y,z\} = \{x_1,x_2,x_3\}$:

$$\begin{array}{lcl} x, y, z \vdash S(x,y,z) & \stackrel{\{(x_1,a,\langle\rangle),(x_2,a,\langle\rangle),(x_3,\tau,\langle\rangle)\}}{\longrightarrow} & x, y, z \vdash S(x,y,z) \\ x, y, z \vdash S(x,y,z) & \stackrel{\{(x_1,s,v),(x_2,s,v),(x_3,\tau,\langle\rangle)\}}{\longrightarrow} & x, y, z, v \vdash S(x,y,z) \end{array}$$

A graphical representation of the third rule for P and the second rule for S (with $x_1 = x, x_2 = z, x_3 = y$) is depicted in Figure 9.1, where the bound names are indicated by their enclosure in round brackets.

In order to be able to define inference rules which describe how to derive more complex transitions from the basic rules, we first introduce the following notion of a most general unifier which transforms a relation Λ, which does not necessarily satisfy the conditions of definition 9.2.4, into a function.

[2]The empty sequence is denoted by $\langle\rangle$.

Definition 9.2.6 (Most General Unifier) *Let $\sigma : N \to N$ be a name substitution. If $\Lambda = \{(x_i, a_i, \tilde{y}_i) \mid i \in \{1, \dots, n\}\}$, then $\sigma(\Lambda) = \{(\sigma(x_i), a_i, \sigma^*(\tilde{y}_i)) \mid i \in \{1, \dots, n\}\}$ where σ^* is the extension of σ to strings.*

For any $\Lambda = \{(x_i, a_i, \tilde{y}_i) \mid i \in \{1, \dots, n\}\} \subseteq N \times Act \times N^*$ we call a substitution $\rho : N \to N$ a unifier *of Λ whenever $\rho(x_i) = x_i$ for $i \in \{1, \dots, n\}$ and $x_i = x_j$ implies $a_i = a_j$ and $\rho^*(\tilde{y}_i) = \rho^*(\tilde{y}_j)$.*

The mapping ρ is called a most general unifier *whenever it is a unifier with a minimal degree of non-injectivity. Unifiers do not necessarily exist.*

Example 9.2.7 The substitution $\rho = \{u/v, u/r, u/s, u/w, u/t\}$ is a unifier for $\Lambda = \{(x, a, uuvw), (x, a, rsst)\}$ since $\rho(\Lambda) = \{(x, a, uuuu)\}$. A most general unifier is, for example, $\rho' = \{u/v, u/r, u/s, w/t\}$ where $\rho'(\Lambda) = \{(x, a, uuuw)\}$.

The set $\Lambda = \{(x, a, u), (x, b, v)\}$, where $a \neq b$, does not have a unifier.

Most general unifiers are needed in order to make sure that whenever two nodes are merged, the strings of nodes created by synchronizing on them, are also merged. Regard, for example, the rewriting rules

$$x \vdash s(x) \xrightarrow{\Lambda_1 = \{(x,a,y)\}} x, y \vdash s'(x, y) \text{ and } x \vdash t(x) \xrightarrow{\Lambda_2 = \{(x,a,z)\}} x, z \vdash t'(x, z).$$

Then—since the edges s and t should agree on a common new name—we expect that

$$x \vdash s(x) \mid t(x) \xrightarrow{\Lambda = \{(x,a,y)\}} x, y \vdash s'(x, y) \mid t'(x, y)$$

where Λ can be obtained by applying the most general unifier to $\Lambda_1 \cup \Lambda_2$.

We introduce the following inference rules for transitions, which are similar to the rules given in [HIM00, HM01].

Definition 9.2.8 (Inference Rules for Transitions) *All possible transitions $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G'$ between graphs are generated by a set R of rewriting rules and the inference rules given below and are closed under injective renaming of all names occurring in a transition.*

$$\textit{(ren)} \quad \frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'}{\rho(\Gamma) \vdash \rho(G) \xrightarrow{\rho'(\rho(\Lambda))} \rho(\Gamma), \Gamma_{\rho'(\rho(\Lambda))} \vdash \rho'(\rho(G'))}$$

where $\rho : \Gamma \to \Gamma$ and ρ' is the most general unifier for $\rho(\Lambda)$.

$$\textit{(par)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma, \Gamma_{\Lambda_1} \vdash G_1' \quad \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Gamma_{\Lambda_2} \vdash G_2'}{\Gamma \vdash G_1 | G_2 \xrightarrow{\rho(\Lambda_1 \cup \Lambda_2)} \Gamma, \Gamma_{\rho(\Lambda_1 \cup \Lambda_2)} \vdash \rho(G_1' | G_2')}$$

if $\Gamma_{\Lambda_1} \cap \Gamma_{\Lambda_2} = \emptyset$ and ρ is the most general unifier for $\Lambda_1 \cup \Lambda_2$.

$$\textit{(hide)} \quad \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\tilde{y})\}} \Gamma, x, \Gamma_\Lambda, Y \vdash G'}{\Gamma \vdash (\nu x)G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash (\nu x)(\nu Y)G'} \quad \textit{where } Y = \mathit{Set}(\tilde{y}) \backslash \Gamma_\Lambda.$$

$$\textit{(idle)} \quad \Gamma \vdash G \xrightarrow{\Lambda} \Gamma \vdash G \quad \textit{where}^3\ \Lambda(x) = (\varepsilon, \langle\rangle) \textit{ for } x \in \Gamma.$$

[3] The empty sequence is denoted by $\langle\rangle$.

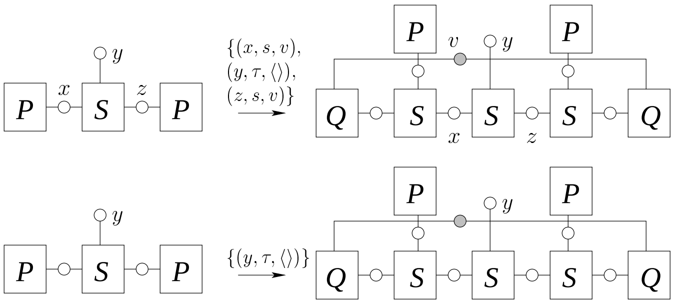


Figure 9.2: Processes establishing a privately shared channel

$$\textit{(new)}\quad \frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Gamma_\Lambda \vdash G'}{\Gamma, x \vdash G \xrightarrow{\Lambda \uplus \{(x,a,\tilde{y})\}} \Gamma, x, \Gamma_\Lambda, \tilde{y} \vdash G'}$$

We also write $R \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$ *whenever this transition can be derived from a set* R *of rewriting rules.*

In every transition Λ assigns to each free name the action it performs and the string of new nodes it creates. Rule *(ren)* deals with non-injective renaming of the nodes of a graph, which is necessary in order to handle edges of the form $s(\dots,x,\dots,x,\dots)$, i.e. edges which are connected several times to the same node. Parallel composition of syntactic judgements is treated in rule *(par)* which makes sure that whenever a synchronization on a node creates a string $\tilde{y}_1$ in the rewriting of $\Gamma \vdash G_1$ and the synchronization on the same node creates a string $\tilde{y}_2$ in the rewriting of $\Gamma \vdash G_2$, then both strings are identified by ρ. In rule *(hide)*, which deals with hiding of names, we do not only have to hide the name itself, but all the names which have been created exclusively by interaction on this name, i.e. all names in the set Y. Furthermore every syntactic judgement can always make an explicit idle step by performing action ε on all its external nodes (rule *(idle)*) and we can add an additional name to the interface which performs arbitrary actions (rule *(new)*). This is due to Hoare synchronization which requires that any number of edges, and therefore also zero edges, can synchronize on a given node.

Example 9.2.9 One of the most interesting rewriting steps which can be derived from the rules of example 9.2.5 is the forking of two processes Q at the same time and the establishment of a privately shared channel between them. We intend to reduce the syntactic judgement $y \vdash (\nu x)(\nu z)(P(x) \mid S(x,y,z) \mid P(z))$ from example 9.2.3. The task of the switch S is to route the signal s on which both processes synchronize, and also to propagate the newly created name.

We first derive a transition for $x,y,z \vdash P(x) \mid S(x,y,z) \mid P(z)$ which is depicted in the upper half of Figure 9.2 and which can be obtained by composing the rewriting rules given in Figure 9.1 where the concept of the most general unifier forces $v = w$. Then in the next step we hide both names x and z which causes all names produced by interaction on x or z to be hidden as well, which means that v is removed from the interface (see the lower half of Figure 9.2).

We can also observe that when a process P creates a new node which is communicated to the environment, a form of name extrusion as in the π-calculus [Mil99] is performed. In the extrusion rule of the labelled transition semantics of the π-calculus, a private, but extruded, name may also appear free in the right-hand side of the rule.

9.3 Representation of Graphs in a P-monoidal Category

In order to be able to describe graph rewriting in tile logic, we will now describe a second possibility of graph representation, which abstracts from names, i.e. nodes are not addressed via their name, but via their position in the interface. In this way we identify all graphs which can be seen as isomorphic, i.e. which are equal up to the laws of structural congruence given in Definition 9.2.2.

We will introduce new operators, such as the duplicator ∇ and the coduplicator Δ (splitting respectively merging nodes), the permutation ρ, the discharger ! and the codischarger ? (hiding respectively creating nodes), which will be defined below (see also Figure 9.3). For the representation of rewriting steps as tiles, it is convenient to be able to describe these unary operators as graphs as well. In order to achieve this, we introduce an interface consisting of two sequences of nodes: root and variable nodes. Additionally we have two binary operators: composition ; and a monoidal operation $\otimes$.

We will now describe graphs as arrows of a P-monoidal (or Part-monoidal) category [BGM98], which can be obtained from dgs-monoidal categories [GH97] by adding an axiom.

P-monoidal categories are an extension of gs-monoidal categories. These describe term graphs, i.e. terms minus copying and garbage collection. Intuitively P-monoidal categories do not only contain term graphs, but also term graphs turned “upside down” and all possible combinations of these graphs.

We first give a formal definition in terms of category theory and then informally describe the meaning of the constants and operations in our setting.

Definition 9.3.1 (P-monoidal category) *A* gs-monoidal category $\mathbf{G}$ *is a six-tuple* $(\mathbf{C}, \otimes, e, \rho, \nabla, !)$ *where* $(\mathbf{C}, \otimes, e, \rho)$ *is a symmetric strict monoidal category and* $! : Id \Rightarrow e : \mathbf{C} \to \mathbf{C}$, $\nabla : Id \Rightarrow \otimes \circ D : \mathbf{C} \to \mathbf{C}$ *are two transformations (D is the diagonal functor), such that* $!_e = \nabla_e = id_e$ *and the following* coherence axioms

$$\nabla_a; id_a \otimes \nabla_a = \nabla_a; \nabla_a \otimes id_a \qquad \nabla_a; id_a \otimes !_a = id_a \qquad \nabla_a; \rho_{a,a} = \nabla_a$$

and the monoidality axioms

$$\nabla_{a \otimes b}; id_a \otimes \rho_{b,a} \otimes id_b = \nabla_a \otimes \nabla_b \qquad !_a \otimes !_b = !_{a \otimes b}$$

are satisfied.

$id_{\underline{1}}$	$\rho_{\underline{1},\underline{1}}$	$\nabla_{\underline{1}}$	$\Delta_{\underline{1}}$	$!_{\underline{1}}$	$?_{\underline{1}}$	edge $s : \underline{n} \to \underline{0}$
1 ○ [1]	1 2 ○ ○ [2] [1]	1 ○ [1] [2]	1 2 ○ [1]	1 ○	○ [1]	1 ○ ⋯ ○ n s

Figure 9.3: P-monoidal operators

A P-monoidal category $\mathbf{D}$ *is an eight-tuple* $(\mathbf{C}, \otimes, e, \rho, \nabla, !, \Delta, ?)$ *such that both the six-tuples* $(\mathbf{C}, \otimes, e, \rho, \nabla, !)$ *and* $(\mathbf{C}^{op}, \otimes, e, \rho, \Delta, ?)$ *are gs-monoidal categories (where* $\mathbf{C}^{op}$ *is the dual category of* $\mathbf{C}$*) and satisfy*

$$\Delta_a; \nabla_a = id_a \otimes \nabla_a; \Delta_a \otimes id_a \qquad \nabla_a; \Delta_a = id_a \qquad ?_a; !_a = id_e$$

In order to model graphs we use a P-monoidal category where the objects are of the form $\underline{n}$*,* $n \in \mathbb{N}$*,* $e = \underline{0}$ *and* $\underline{n} \otimes \underline{m}$ *is defined as* $\underline{n+m}$*. If* Σ *is a set of symbols each associated with a sort* $\underline{n} \to \underline{0}$*, then* $\mathbf{PMon}(\Sigma)$ *is the P-monoidal category freely generated from the symbols in* Σ *which are interpreted as arrows.*

In order to save brackets we adopt the convention that the monoidal operation $\otimes$ takes precedence over ; (the composition operator of the category). Note that by omitting the last axiom $?_a; !_a = id_e$ we obtain exactly the definition of a dgs-monoidal category.

We depict an arrow $t : \underline{n} \to \underline{m}$ of $\mathbf{PMon}(\Sigma)$ by drawing a hypergraph with two sequences of external nodes: n root nodes and m variable nodes (see Figure 9.3). Root nodes are indicated by labels $1, 2, \ldots$, variable nodes by labels $[1], [2], \ldots$ The composition operator ; merges the variable nodes of its first argument with the root nodes of its second argument. The tensor product $\otimes$ takes the disjoint union of two graphs and concatenates the sequences of root respectively variable nodes of its two arguments. Note that the axiom $?_a; !_a = id_e$ has the intuitive meaning that isolated nodes are garbage-collected.

Similar to the case of syntactic judgements it can be shown that two terms of $\mathbf{PMon}(\Sigma)$ are equal if and only if the underlying hypergraphs are isomorphic (up to isolated nodes) [BGM98].

There is a one-to-one correspondence between P-monoidal terms $w : \underline{m} \to \underline{n} \in \mathbf{PMon}(\emptyset)$ (corresponding to the set of all discrete graphs) and equivalence relations on the union of $\{r\} \times \{1, \ldots, m\}$ and $\{v\} \times \{1, \ldots, n\}$. We say that $(r, i) \equiv_w (r, j)$ whenever the i-th and the j-th root node of w are equal, additionally $(r, i) \equiv_w (v, j)$ whenever the i-th root node and the j-th variable node are equal and $(v, i) \equiv_w (v, j)$ whenever the i-th and the j-th variable node are equal. An equivalence relation on a set can also be seen as a partition of this set, which is the origin of the name P(art)-monoidal category.

Syntactic judgements can be encoded into P-monoidal terms. We introduce a mapping α assigning to each name its position in the sequence of external nodes. One name may appear in several positions.

Definition 9.3.2 (Encoding of Syntactic Judgements) *Let* $\Gamma \vdash G$ *be a syntactic judgement and let* $\alpha : \{1, \ldots, n\} \to \Gamma$ *be a surjective (but not nec-*

essarily injective) function, indicating which positions a name should occupy in the interface. We will also call α *an* n-ary interface mapping.

Then $[\![\Gamma \vdash G]\!]_\alpha : \underline{n} \to \underline{0}$ *is an arrow of* $\mathbf{PMon}(\Sigma)$ *where* Σ *contains* $s : \underline{m} \to \underline{0}$ *for every edge of the form* $s(x_1, \ldots, x_m)$*. The encoding is defined as follows:*

$$\begin{array}{rcl}
[\![\Gamma \vdash G_1|G_2]\!]_\alpha &=& \nabla_{\underline{n}}; [\![\Gamma \vdash G_1]\!]_\alpha \otimes [\![\Gamma \vdash G_2]\!]_\alpha \\
[\![\Gamma \vdash (\nu x)G]\!]_\alpha &=& id_{\underline{n}} \otimes ?_{\underline{1}}; [\![\Gamma, x \vdash G]\!]_{\alpha \cup \{n+1 \mapsto x\}} \text{ if } x \notin \Gamma \\
[\![\Gamma \vdash nil]\!]_\alpha &=& !_{\underline{n}} \\
[\![\Gamma \vdash s(x_1, \ldots, x_m)]\!]_\alpha &=& w; s
\end{array}$$

where $w : \underline{n} \to \underline{m} \in \mathbf{PMon}(\emptyset)$ *(the "wiring") such that* $\equiv_w$ *is the smallest equivalence containing* $\{((r, i), (v, j)) \mid \alpha(i) = x_j\}$*.*

Note that if α is injective, all P-monoidal terms of the form $[\![\Gamma \vdash G]\!]_\alpha$ lie in a subcategory of $\mathbf{PMon}(\Sigma)$ which is generated by all symbols and constants apart from $\Delta_{\underline{n}}$, which means in practice that all root nodes in the interface of a graph are distinct.

Example 9.3.3 By encoding the syntactic judgement

$$\Gamma \vdash G = y \vdash (\nu x)(\nu z)(P(x) \mid S(x, y, z) \mid P(z))$$

of Example 9.2.3 with the mapping $\alpha : \{1\} \to \{y\}$, $\alpha(1) = y$ we obtain the following P-monoidal term

$$id_{\underline{1}} \otimes ?_{\underline{1}}; id_{\underline{2}} \otimes ?_{\underline{1}}; \nabla_{\underline{3}}; (!_{\underline{1}} \otimes id_{\underline{1}} \otimes !_{\underline{1}}; P) \otimes (\nabla_{\underline{3}}; (\rho_{\underline{1},\underline{1}} \otimes id_{\underline{1}}; S) \otimes (!_{\underline{2}} \otimes id_{\underline{1}}; P)).$$

Proposition 9.3.4 *It holds that* $\Gamma \vdash G \equiv \Gamma' \vdash G'$ *if and only if there exist injective* α, α' *such that* $[\![\Gamma \vdash G]\!]_\alpha = [\![\Gamma' \vdash G']\!]_{\alpha'}$*.*

9.4 A Tile Logic Representation for Synchronized Graph Rewriting with Mobility

We now describe graph rewriting in the framework of tile logic, in which we consider rewrites of the form $s \xrightarrow[b]{a} t$ where s and t are configurations (i.e. hypergraphs) of a system and both may have root and variable nodes, their interface to the environment. The observation a describes the actions of s with respect to its root nodes, while b describes the interaction with respect to its variable nodes. The rules of tile logic describe how to derive partial rewrites and how to extend them whenever configurations are contextualized or instantiated, or—in this case—whenever two graphs are combined.

We first define the notion of a tile.

Definition 9.4.1 *Let* $\mathcal{H}$ *and* $\mathcal{V}$ *be two categories which coincide in their set of objects, which is* $\{\underline{n} \mid n \in \mathbb{N}\}$*. We call* $\mathcal{H}$ *the* horizontal category *and* $\mathcal{V}$ *the* vertical category*. The arrows of* $\mathcal{H}$ *are also called* configurations *and the arrows of* $\mathcal{V}$ *are called* observations*.*

Tile	Parallel composition	Horizontal composition	Vertical composition

$$
\begin{tikzcd}
\underline{n} \arrow[r, "s"'] \arrow[d, "a"'] & \underline{m} \arrow[d, "b"] \\
\underline{n'} \arrow[r, "t"'] & \underline{m'}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \underline{k} \arrow[rr, "s'"'] \arrow[dd, "a'"'] & & \underline{l} \arrow[dd, "b'"'] \\
\underline{n} \arrow[rr, "s"'] \arrow[dd, "a"'] & & \underline{m} \arrow[dd, "b"] & \\
 & \underline{k'} \arrow[rr, "t'"'] & & \underline{l'} \\
\underline{n'} \arrow[rr, "t"'] & & \underline{m'} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\underline{n} \arrow[r, "s"'] \arrow[d, "a"'] & \underline{m} \arrow[r, "s'"'] \arrow[d, "c"'] & \underline{k} \arrow[d, "b"'] \\
\underline{n'} \arrow[r, "t"'] & \underline{m'} \arrow[r, "t'"'] & \underline{k'}
\end{tikzcd}
\qquad
\begin{tikzcd}
\underline{n} \arrow[r, "s"'] \arrow[d, "a"'] & \underline{m} \arrow[d, "b"'] \\
\underline{n'} \arrow[r, "u"'] \arrow[d, "a'"'] & \underline{m'} \arrow[d, "b'"'] \\
\underline{n''} \arrow[r, "t"'] & \underline{m''}
\end{tikzcd}
$$

Figure 9.4: Composing tiles

A tile (compare [GM99]) is of the form $s \xrightarrow[b]{a} t$ *where* $s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}$ *are elements of* $\mathcal{H}$, *and* $a : \underline{n} \to \underline{n'}, b : \underline{m} \to \underline{m'}$ *are elements of* $\mathcal{V}$.

Tiles can be depicted as squares (see the leftmost square in Figure 9.4).

We can now define the more specific tiles of tile graph rewriting and the way in which they can be composed.

Definition 9.4.2 (Tile graph rewriting) *Let* $\Sigma = \{s : \underline{n} \to \underline{0} \mid s \text{ is an edge of arity } n\}$ *and let* $\mathbf{PMon}(\Sigma)$ *be the horizontal category* $\mathcal{H}$ *whereas the vertical category* $\mathcal{V}$ *is the free monoidal category*[4] *generated by the arrows* $a : \underline{1} \to \underline{1+n}$ *for every* $a \in Act\backslash\{\varepsilon\}$ *with* $ar(a) = n$. *The idle action* ε *corresponds to the identity arrow* $id_{\underline{1}}$.

Tiles can be constructed in the following way: a tile is either taken from a fixed set $\mathcal{R}$ *of* generator tiles, *or it is a* reflexive tile *(h-refl) or (v-refl), or it is one of the auxiliary tiles (dupl), (codupl), (disch), (codisch) or (perm), or it is obtained by parallel composition (p-comp), horizontal composition (h-comp) or vertical composition (v-comp) (see also Figure 9.4).*

We write $\mathcal{R} \Vdash s \xrightarrow[b]{a} t$ *whenever this tile can be derived from the generator tiles in* $\mathcal{R}$.

$$
\textit{(h-refl)}\ \frac{s : \underline{n} \to \underline{m} \in \mathcal{H}}{s \xrightarrow[id_{\underline{m}}]{id_{\underline{n}}} s}
\qquad
\textit{(v-refl)}\ \frac{a : \underline{n} \to \underline{m} \in \mathcal{V}}{id_{\underline{n}} \xrightarrow[a]{a} id_{\underline{m}}}
$$

$$
\textit{(dupl)}\ \frac{a : \underline{n} \to \underline{m} \in \mathcal{V}}{\nabla_{\underline{n}} \xrightarrow[a \otimes a]{a} \nabla_{\underline{m}}}
\qquad
\textit{(codupl)}\ \frac{a : \underline{n} \to \underline{m} \in \mathcal{V}}{\Delta_{\underline{n}} \xrightarrow[a]{a \otimes a} \Delta_{\underline{m}}}
$$

$$
\textit{(disch)}\ \frac{a : \underline{n} \to \underline{m} \in \mathcal{V}}{!_{\underline{n}} \xrightarrow[id_{\underline{0}}]{a} !_{\underline{m}}}
\qquad
\textit{(codisch)}\ \frac{a : \underline{n} \to \underline{m} \in \mathcal{V}}{?_{\underline{n}} \xrightarrow[a]{id_{\underline{0}}} ?_{\underline{m}}}
$$

[4]Given a set A of arrows, the free monoidal category generated by A consists of all arrows which can be obtained from composing the arrows of A with the composition operator ; and the monoidal operator $\otimes$, observing the category axioms (; is associative and $\varepsilon = id_{\underline{1}}$ is its unit), the monoidality axioms ($\otimes$ is associative and $id_{\underline{0}}$ is its unit) and $a_1; a_1' \otimes a_2; a_2' = (a_1 \otimes a_2); (a_1' \otimes a_2')$.

$$\textit{(perm)}\ \frac{a:\underline{n}\to\underline{m},\ b:\underline{n'}\to\underline{m'}\in\mathcal{V}}{\rho_{\underline{n},\underline{n'}}\ \xrightarrow[b\otimes a]{a\otimes b}\ \rho_{\underline{m},\underline{m'}}}\qquad \textit{(p-comp)}\ \frac{s\ \xrightarrow[b]{a}\ t,\ s'\ \xrightarrow[b']{a'}\ t'}{s\otimes s'\ \xrightarrow[b\otimes b']{a\otimes a'}\ t\otimes t'}$$

$$\textit{(h-comp)}\ \frac{s\ \xrightarrow[c]{a}\ t,\ s'\ \xrightarrow[b]{c}\ t'}{s;s'\ \xrightarrow[b]{a}\ t;t'}\qquad \textit{(v-comp)}\ \frac{s\ \xrightarrow[b]{a}\ u,\ u\ \xrightarrow[b']{a'}\ t}{s\ \xrightarrow[b;b']{a;a'}\ t}$$

We first show that if the generator tiles exhibit a certain well-formedness property, then we can construct every tile in the following way: first, we can use all rules apart from *(v-comp)* in order to construct several tiles which, finally, can be combined with rule *(v-comp)*. This says, basically, that it is sufficient to examine tiles which describe one single rewriting step.

Proposition 9.4.3 *We assume that the set $\mathcal{R}$ of generator tiles satisfies the following properties: let $s\ \xrightarrow[b]{a}\ t$ be a generator tile, then it holds that $s\in\Sigma$ and furthermore there are actions $a_1,\dots,a_n\in Act\backslash\{\varepsilon\}$, such that $a=a_1\otimes\ldots\otimes a_n$ and $b=id_{\underline{0}}$.*

Now let $\mathcal{R}\Vdash s\ \xrightarrow[b]{a}\ t$. Then it holds that there are configurations $s=s_0,s_1,\dots,s_m=t$ and observations $a'_1,\dots,a'_m,b'_1,\dots,b'_m$ such that $\mathcal{R}\Vdash s_{i-1}\ \xrightarrow[b'_i]{a'_i}\ s_i$ for $i\in\{1,\dots,n\}$ and the respective tiles can be derived without rule (v-comp). Furthermore $a=a'_1;\dots;a'_m$ and $b=b'_1;\dots;b'_m$.

We can now formulate one of the two main results of this paper: the operational correspondence between rewriting of syntactic judgements and of P-monoidal terms.

We first introduce the following notation: let $x_1\dots x_n$ be a string of names. By $\alpha=\langle x_1\dots x_n\rangle$ we denote the interface mapping $\alpha:\{1,\dots,n\}\to\{x_1,\dots,x_n\}$ where $\alpha(i)=x_i$.

Proposition 9.4.4 (Operational Correspondence) *Let R be a set of rewriting rules on syntactic judgements. We define a set $\mathcal{R}$ of generator tiles as follows:*

$$\begin{aligned}\mathcal{R} &= \{s\ \xrightarrow[id_{\underline{0}}]{a_1\otimes\ldots\otimes a_m}\ [\![\Gamma'\vdash G']\!]_{\langle x_1\tilde{y}_1\ldots x_m\tilde{y}_m\rangle}\mid\\ &\qquad (x_1,\dots,x_m\vdash s(x_1,\dots,x_m)\xrightarrow{\Lambda}\Gamma'\vdash G')\in R,\ a_i=act_\Lambda(x_i),\\ &\qquad \tilde{y}_i=n_\Lambda(x_i)\}.\end{aligned}$$

- *It holds that $R\Vdash(\Gamma\vdash G\xrightarrow{\Lambda}\Gamma'\vdash G')$ implies $\mathcal{R}\Vdash([\![\Gamma\vdash G]\!]_\alpha\ \xrightarrow[id_{\underline{0}}]{a_1\otimes\ldots\otimes a_m}\ [\![\Gamma'\vdash G']\!]_{\langle\alpha(1)\tilde{y}_1\ldots\alpha(m)\tilde{y}_m\rangle})$ where $a_i=act_\Lambda(\alpha(i))$, $\tilde{y}_i=n_\Lambda(\alpha(i))$.*

- *And it holds that if $\mathcal{R} \Vdash ([\![\Gamma \vdash G]\!]_\alpha \xrightarrow[id_{\underline{0}}]{a_1\otimes\ldots\otimes a_m} t)$ for some P-monoidal term t, then $R \Vdash (\Gamma \vdash G \xrightarrow{\Lambda} \Gamma' \vdash G')$ where $a_i = act_\Lambda(\alpha(i))$, $\tilde{y}_i = n_\Lambda(\alpha(i))$ and $[\![\Gamma' \vdash G']\!]_{\langle \alpha(1)\tilde{y}_1\ldots\alpha(m)\tilde{y}_m\rangle} = t$.*

Proof (Sketch): The first half of the proposition can be shown by induction on the inference rules applied. The second half of the proposition is shown by induction on the syntactic structure of $\Gamma \vdash G$ and with the decomposition property (Proposition 9.5.3 which will be proved in Section 9.5 without referring back to this proposition). □

Example 9.4.5 Encoding the rewrite rules on syntactic judgements from Example 9.2.5 into generator tiles gives us the tiles depicted in Figure 9.5.

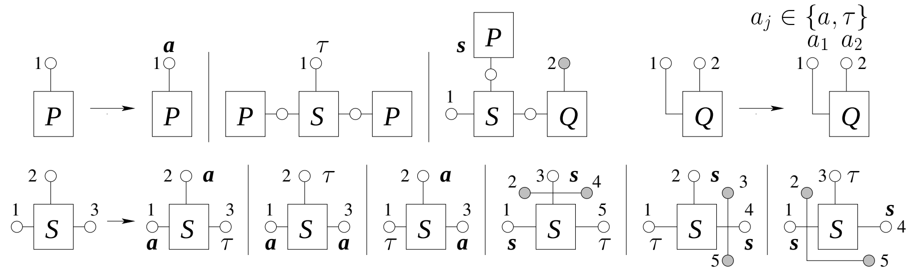


Figure 9.5: Generator tiles

Here we represent a tile of the form $s \xrightarrow[id_{\underline{0}}]{a_1\otimes\ldots\otimes a_n} t$ by drawing s and t as graphs and by labelling the free nodes of t by the actions $a_1, \ldots, a_n$. Specifically if $N_i = \sum_{j=1}^{i}(ar(a_j)+1)$, then the $N_{i-1}+1$-st free node of t is labelled a_i, while the next $ar(a_i)-1$ nodes stay unlabelled (and are shaded grey in the graphical representation), indicating that these nodes are generated by the action a_i. Two nodes that are connected by a line represent one and the same node.

9.5 Bisimilarity is a Congruence

Based on tiles we can now define the notions of bisimulation and bisimilarity and thus define a notion of an observable, compositional equivalence on graphs.

Definition 9.5.1 (Bisimulation on tiles) *Given a labelled transition system, a bisimulation is a symmetric, reflexive relation $\sim$ on the states of the transition system, such that if $s \sim t$ and $s \xrightarrow{a} s'$, then there exists a transition $t \xrightarrow{a} t'$ such that $s' \sim t'$. We say that two states s and t are bisimilar ($s \simeq t$) whenever there is a bisimulation $\sim$ such that $s \sim t$.*

In tile graph rewriting the tile $s \xrightarrow[b]{a} t$ is considered to be a transition with label (a, b). We say that two configuration s, t are bisimilar wrt. a set $\mathcal{R}$ of generator tiles (in symbols $s \simeq_\mathcal{R} t$) whenever s and t are bisimilar in the transition system generated by $\mathcal{R}$.

It is already known that bisimilarity is a congruence whenever the underlying tile system satisfies the following decomposition property.

Definition 9.5.2 (Decomposition Property) *A tile system satisfies the decomposition property if for all tiles* $s \xrightarrow[b]{a} t$ *entailed by the tile system, it holds that (1) if* $s = s_1; s_2$ *then there exist* $c \in \mathcal{V}, t_1, t_2 \in \mathcal{H}$ *such that* $s_1 \xrightarrow[c]{a} t_1$, $t_1 \xrightarrow[b]{c} t_2$ *and* $t = t_1; t_2$ *(2) if* $s = s_1 \otimes s_2$ *then there exist* $a_1, a_2, b_1, b_2 \in \mathcal{V}, t_1, t_2 \in \mathcal{H}$ *such that* $s_1 \xrightarrow[b_1]{a_1} t_1$, $s_2 \xrightarrow[b_2]{a_2} t_2$, $a = a_1 \otimes a_2$, $b = b_1 \otimes b_2$ *and* $t = t_1 \otimes t_2$.

Proposition 9.5.3 (cf. [GM99]) *If a tile system satisfies the decomposition property, then bisimilarity defined on its transition system is a congruence.*

Similar to the case of de Simone [dS85] or `tyft/tyxt`-formats [GV92], there is a sufficient syntactical property ensuring that bisimilarity is indeed a congruence, which is stated in the second main result of this paper.

Proposition 9.5.4 *If, in tile graph rewriting, all generator tiles satisfy the basic source property, i.e. if for every generator tile* $s \xrightarrow[b]{a} t$ *it holds that* $s \in \Sigma$, *then the decomposition property holds. Thus bisimilarity is a congruence.*

Proof (Sketch): By induction on the derivation of a tile, following the lines of a similar proof in [BFEMOM00b]. □

Corollary 9.5.5 *All the tile graph rewriting systems derived from rewriting rules on syntactic judgements satisfy the basic source property. Thus the decomposition property holds and bisimilarity is a congruence.*

Having established that bisimilarity is indeed a congruence in the case of tile graph rewriting we now transfer this result back to rewriting on syntactic judgements with the help of the operational correspondence (Proposition 9.4.4). First we have to define bisimulation on syntactic judgements. We use the following intuition: an observer from the outside has access to the external nodes of a graph, however he or she is not able to determine their names and he or she should also not be able to find out whether or not two nodes are equal. So, given two syntactic judgements, we add an interface mapping α which assigns numbers to names and in this way hides the internal details from an external observer.

For the next definition remember that the mapping $i \mapsto x_i$ is denoted by $\langle x_1 \dots x_n \rangle$.

Definition 9.5.6 (Bisimulation on syntactic judgements) *Let* $\Gamma \vdash G$ *be a syntactic judgements. An* n*-ary interface for a syntactic judgement is a surjective mapping* $\alpha : \{1, \dots, n\} \to \Gamma$, *as defined in Definition 9.3.2.*

A symmetric, reflexive relation $\sim$ *on pairs consisting of syntactic judgements and their corresponding interfaces is called a bisimulation (wrt. a set* R *of rewriting rules) if whenever* $(\Gamma_1 \vdash G_1, \alpha_1) \sim (\Gamma_2 \vdash G_2, \alpha_2)$, *then*

- *there is an* $n \in \mathbb{N}$ *such that* α_1 *and* α_2 *are both* n*-ary interfaces*

- *whenever* $\Gamma_1 \vdash G_1 \xrightarrow{\Lambda_1} \Gamma_1' \vdash G_1'$ *with* $\Lambda_1(\alpha_1(i)) = (a_i, \tilde{y}_i)$*, then it holds that* $\Gamma_2 \vdash G_2 \xrightarrow{\Lambda_2} \Gamma_2' \vdash G_2'$ *with* $\Lambda_2(\alpha_2(i)) = (a_i, \tilde{z}_i)$ *and*

$$(\Gamma_1' \vdash G_1', \langle \alpha_1(1)\tilde{y}_1 \ldots \alpha_1(n)\tilde{y}_n \rangle) \;\sim\; (\Gamma_2' \vdash G_2', \langle \alpha_2(1)\tilde{z}_1 \ldots \alpha_2(n)\tilde{z}_n \rangle).$$

We say that two pairs $(\Gamma_1 \vdash G_1, \alpha_1)$ *and* $(\Gamma_2 \vdash G_2, \alpha_2)$ *are bisimilar (wrt. a set* R *of rewriting rules) whenever there is a bisimulation* $\sim$ *(wrt. a set* R *of rewriting rules) such that* $(\Gamma_1 \vdash G_1, \alpha_1) \;\sim\; (\Gamma_2 \vdash G_2, \alpha_2)$*. Bisimilarity on syntactic judgements is denoted by the symbol* $\simeq_R$*.*

In order to show that bisimilarity on syntactic judgements is a congruence with respect to parallel composition and hiding, we need the following result on full abstraction.

Proposition 9.5.7 (Full abstraction) *The encoding* $[\![\_]\!]_\alpha$ *is fully abstract in the following sense: for any set* R *of rewriting rules it holds that*

$$(\Gamma_1 \vdash G_1, \alpha_1) \simeq_R (\Gamma_2 \vdash G_2, \alpha_2) \iff [\![\Gamma_1 \vdash G_1]\!]_{\alpha_1} \simeq_{\mathcal{R}} [\![\Gamma_2 \vdash G_2]\!]_{\alpha_2}$$

where $\mathcal{R}$ *is defined as in Proposition 9.4.4.*

Proof (Sketch): Straightforward by regarding the respective definitions of bisimilarity, from Proposition 9.4.3 and the operational correspondence result in Proposition 9.4.4. □

Now it is straightforward to show that bisimilarity on syntactic judgements is a congruence as well.

Proposition 9.5.8 *Let* R *be a set of rewriting rules and let* $\mathcal{R}$ *be the corresponding set of generator tiles defined as in Proposition 9.4.4.*

We assume that $(\Gamma_1, X_1 \vdash G_1, \alpha_1) \simeq_R (\Gamma_2, X_2 \vdash G_2, \alpha_2)$ *such that* $\alpha_i : \{1, \ldots, n+m\} \to \Gamma_i \cup X_i$ *and* $\alpha_i^{-1}(X_i) = \{n+1, \ldots, n+m\}$ *for* $i \in \{1, 2\}$*. Then it holds that* $(\Gamma_1 \vdash (\nu X_1)G_1, \alpha_1|_{\{1,\ldots,n\}}) \simeq_R (\Gamma_2 \vdash (\nu X_2)G_2, \alpha_2|_{\{1,\ldots,n\}})$*.*

And if $(\Gamma_1 \vdash G_1, \alpha_1) \simeq_R (\Gamma_2 \vdash G_2, \alpha_2)$ *and* $(\Gamma_1 \vdash G_1', \alpha_1) \simeq_R (\Gamma_2 \vdash G_2', \alpha_2)$*, then it follows that* $(\Gamma_1 \vdash G_1 \mid G_1', \alpha_1) \simeq_R (\Gamma_2 \vdash G_2 \mid G_2', \alpha_2)$*.*

Proof (Sketch): Straightforward by using the full abstraction result from Proposition 9.5.7, by using that fact that bisimilarity on P-monoidal terms is a congruence (see Proposition 9.5.4) and by regarding the definition of the encoding $[\![\_]\!]_\alpha$ in Definition 9.3.2. □

9.6 Bisimulation up-to Congruence

In order to show that two graphs are bisimilar in practice, we need a proof technique for bisimulation, a so-called bisimulation up-to congruence (for up-to techniques see also [MS92a]).

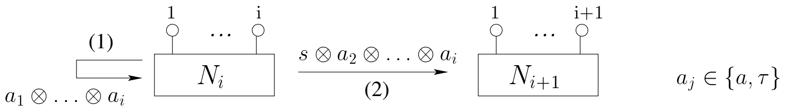


Figure 9.6: Specification of the communication network

Definition 9.6.1 *For a given relation B on P-monoidal terms, we denote by $\equiv_B$ the smallest congruence (with respect to the operators ; and $\otimes$) that contains B.*

A symmetric relation B is called a bisimulation up-to congruence whenever $(s,t) \in B$ and $s \xrightarrow[b]{a} s'$ imply $t \xrightarrow[b]{a} t'$ and $s' \equiv_B t'$.

Proposition 9.6.2 *If the decomposition property holds for the respective tile logic and B is a bisimulation up-to congruence, then $\equiv_B$ is a bisimulation.*

It is typically easier to show that B is a bisimulation up-to congruence than to show that $\equiv_B$ is a bisimulation, mainly because B can be much smaller than $\equiv_B$ and so there are fewer cases to consider. It may even be the case that B is finite and $\equiv_B$ is infinite in size.

9.7 Example: Communication Network

We return to our running example and intend to investigate which steps a single process can perform, or rather which are the actions that are observable from the outside. To this aim we give a specification N_1 which models in a single edge the entire communication topology P may generate. Note that a process P may start with a single free node, but can create new free nodes by performing s actions. The specification has to take this into account and N_1 may therefore reduce to N_2, N_3 etc., where $N_i : \underline{i} \to \underline{0}$.

The generator tiles for the specification are depicted in Figure 9.6, where this time we put the observations on the arrows rather than on the free nodes of the right-hand side.

The edge N_i can either perform an arbitrary combination of a and τ actions and stay N_i or it can perform an s action on its first node and a's and τ's on the remaining nodes and become N_{i+1}.

In order to show that P and N_1 are indeed bisimilar we proceed as follows: We consider the tile system generated by both the tiles belonging to processes and switches and the tiles of the specification, and denote the combined set of generator tiles by $\mathcal{R}$. Since the set of edges involved in the first set of generator tiles is disjoint from that in the second set, the rules can not interfere with each other and P respectively N_1 can not perform more reductions with the additional tiles.

Proposition 9.7.1 *The symmetric closure of the relation B depicted in Figure 9.7 is a bisimulation up-to congruence. Since the basic source property and therefore the decomposition property hold, it follows that $\equiv_B$ is a bisimulation.*

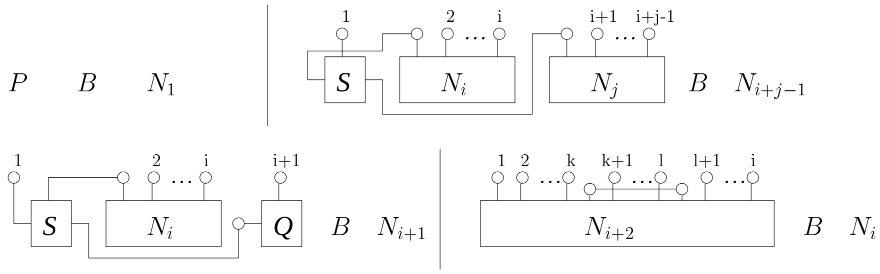
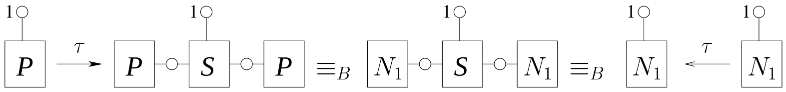


Figure 9.7: Example of a bisimulation up-to congruence

And since $(P, N_1) \in B$, *we conclude that* $P \simeq_{\mathcal{R}} N_1$.

Proof (Sketch): From Proposition 9.4.3 it follows that it is sufficient to regard only tiles which can be composed without using rule *(v-comp)*.

We exemplarily treat the following case: let $(P, N_1) \in B$ and we assume that P performs a τ action and replaces itself with a switch and two processes, i.e. the second rewrite rule for P is applied. In this case $N_1 \xrightarrow[id_{\underline{0}}]{\tau} N_1$ and we have to show that the two resulting graphs are in the $\equiv_B$-relation:



□

The scenario we have presented resembles the view of a user which starts with one single port of access to a network which may be huge. The user can request further connections into the network (with an s action) and he or she can send signals a which are received in the network. However, in whatever way the user interacts with the entire network, its topology will always be hidden, its expansion unobservable and it thus constitutes a black box. Internal communications, of which several may take place in parallel, are indistinguishable from τ-steps and thus completely hidden from the environment.

If, however, we start with a disconnected graph with i external nodes and compare it to an edge N_i, the two expressions are *not* bisimilar. Consider for example the two graphs $P \otimes P$ and N_2, both of arity 2. We observe that $P \otimes P \xrightarrow[id_{\underline{0}}]{a \otimes id_{\underline{1}}} P \otimes P$, whereas N_2 can not match this transition. If we assume that the first node of N_2 produces an action a, we either get a or τ as the action of the second node, but we never get $id_{\underline{1}}$. In general we can state that whenever we have a graph t consisting of processes and switches, we can determine its connected external nodes in the following way: a set of external nodes is connected if and only if there is a transition such that exactly the nodes of the set perform an action different from $id_{\underline{1}} = \varepsilon$, and that furthermore there is no proper subset with the same property.

Another scenario would be to start with a graph with several external nodes of which two or more are connected via switches. In this case an s action originating on one of the external nodes may be routed to a different external node, giving us two new nodes in the interface, which, however, must be equal.

9.8 Conclusion

We have presented synchronized graph rewriting with mobility for two forms of graph representation (syntactic judgements and arrows in a P-monoidal category) and we have shown that bisimilarity for synchronized graph rewriting is a congruence with respect to graph composition. A tile logic semantics for synchronized graph rewriting without mobility has already been defined in [MR99], whereas synchronized graph rewriting with mobility has so far only been considered for syntactic judgements [HIM00, HM01], but not in the context of tile logics. Moreover no equivalence of graphs based on observations has been introduced there.

In [HM01] not only Hoare synchronization, but also Milner synchronization is treated and an encoding of the π-calculus into synchronized graph rewriting is given, using Milner synchronization.

An earlier form of synchronized graph rewriting has been treated in [DM87]. Furthermore, the mobility treated in this paper is reminiscent of the rendezvous mechanism presented in [DDK93].

In general we know of little work concerning the definition of observational equivalences for graph rewriting. As already mentioned in the introduction there are basically two ways to go when one wants to introduce bisimilarity on graphs. The first alternative would be to base the theory on unlabelled production as in the double-pushout approach [Ehr79]. Without labels on the transitions it is necessary to define canonical forms of graphs, in order to be able to observe something. Work by Fernández and Mackie on interaction nets [FM98] and by Yoshida on process graphs [Yos94] goes in that direction.

In π-calculus, for example, the canonical forms mentioned above are called "barbs" and a process has a barb for channel c whenever it is able to perform an input or output on c. The resulting bisimulation is therefore called barbed bisimulation [MS92b], which ordinarily does not induce a congruence, and the definition of the smallest congruence containing barbed bisimilarity requires universal quantification over all possible contexts. This, however, makes actual proofs of bisimilarity complicated.

In this paper, however, we chose to use synchronized graph rewriting as a framework. We model transitions whose transition labels (observations) describe exactly the interaction of a single edge with its environment. This enables us to define a simple syntactic property on the rewriting rules which ensures that bisimilarity is a congruence. Existing work is mainly related to action calculi [Mil96, LM00] which also have a graphical representation.

As we have seen in the example in Section 9.7, the bisimilarity defined in this paper is rather coarse-grained: it can determine whether a network is connected or disconnected, but we can, for example, not determine the degree of parallelism in a network. In order to be able to do this, it would be necessary to establish a concurrent semantics for synchronized graph rewriting. Another interesting extension would be to enrich the notion of observation: so far we are only able to observe the actions performed on external nodes, but we are not able to determine whether, for example, two nodes are connected by an edge. It seems therefore promising to extend the framework in such a way that we are allowed to observe occurrences of specific subgraphs.

Acknowledgements: We would like to thank Roberto Bruni and Dan Hirsch for their help.

Chapter 10

A General Framework for Types in Graph Rewriting

Abstract

A general framework for typing graph rewriting systems is presented: the idea is to statically derive a type graph from a given graph. In contrast to the original graph, the type graph is invariant under reduction, but still contains meaningful behaviour information. We present conditions, a type system for graph rewriting should satisfy, and a methodology for proving these conditions. In two case studies it is shown how to incorporate existing type systems (for the polyadic π-calculus and for a concurrent object-oriented calculus) into the general framework.

10.1 Introduction

In the past, many formalisms for the specification of concurrent and distributed systems have emerged. Some of them are aimed at providing an encompassing theory: a very general framework in which to describe and reason about interconnected processes. Examples are action calculi [Mil96], rewriting logic [Mes96] and graph rewriting [EKMR99] (for a comparison see [GM02]). They all contain a method of building terms (or graphs) from basic elements and a method of deriving reduction rules describing the dynamic behaviour of these terms in an operational way.

A general theory is useful, if concepts appearing in instances of a theory can be generalised, yielding guidelines and relieving us of the burden to prove universal concepts for every single special case. An example for such a generalisation is the work presented for action calculi in [LM00] where a method for deriving a labelled transition semantics from a set of reaction rules is presented. We concentrate on graph rewriting (more specifically hypergraph rewriting) and attempt to generalise the concept of type systems, where, in this context, a type may be a rather complex structure.

Compared to action calculi[1] and rewriting logic, graph rewriting differs in a significant way in that connections between components are described explicitly (by connecting them by edges) rather than implicitly (by referring to the same channel name). We claim that this feature—together with the fact that it is easy to add an additional layer containing annotations and constraints to a graph—can simplify the design of a type system and therefore the static analysis of a graph rewriting system.

After introducing our model of graph rewriting and a method for annotating graphs, we will present a general framework for type systems where both—the expression to be typed and the type itself—are hypergraphs and will show how to reduce the proof obligations for instantiations of the framework. We are interested in the following properties: correctness of a type system (if an expression has a certain type, then we can conclude that this expression has certain properties), the subject reduction property (types are invariant under reduction) and compositionality (the type of an expression can always be derived from the types of its subexpressions). Parts of the proofs of these properties can already be conducted for the general case.

We will then show that our framework is realistic by instantiating it to two well-known type systems: a type system avoiding run-time errors in the polyadic π-calculus [Mil93] and a type system avoiding "message not understood"-errors in a concurrent object-oriented setting. A third example enforcing a security policy for untrustworthy applets is included in the full version [Kön00b].

10.2 Hypergraph Rewriting and Hypergraph Annotation

We first define some basic notions concerning hypergraphs (see also [Hab92]) and a method for inductively constructing hypergraphs.

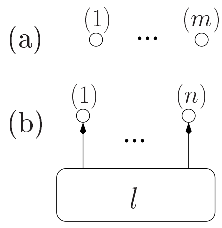
Definition 10.2.1 (Hypergraph) *Let L be a fixed set of labels. A hypergraph $H = (V_H, E_H, s_H, l_H, \chi_H)$ consists of a set of nodes V_H, a set of edges E_H, a connection mapping $s_H : E_H \to V_H^*$, an edge labelling $l_H : E_H \to L$ and a string $\chi_H \in V_H^*$ of external nodes. A hypergraph morphism $\varphi : H \to H'$ (consisting of $\varphi_V : V_H \to V_{H'}$ and $\varphi_E : E_H \to E_{H'}$) satisfies[2] $\varphi_V(s_H(e)) = s_{H'}(\varphi_E(e))$ and $l_H(e) = l_{H'}(\varphi_E(e))$. A strong morphism (denoted by the arrow $\twoheadrightarrow$) additionally preserves the external nodes, i.e. $\varphi_V(\chi_H) = \chi_{H'}$. We write $H \cong H'$ (H is isomorphic to H') if there is a bijective strong morphism from H to H'.*

The arity of a hypergraph H is defined as $ar(H) = |\chi_H|$ while the arity of an edge e of H is $ar(e) = |s_H(e)|$. External nodes are the interface of a hypergraph towards its environment and are used to attach hypergraphs.

[1] Here we mean action calculi in their standard string notation. There is also a graph notation for action calculi, see e.g. [Has97].

[2] The application of φ_V to a string of nodes is defined pointwise.

Notation: We call a hypergraph *discrete*, if its edge set is empty. By $\mathbf{m}$ we denote a discrete graph of arity $m \in \mathbb{N}$ with m nodes where every node is external (see Figure (a) to the right, external nodes are labelled (1), (2), ... in their respective order).
The hypergraph $H = [l]_n$ contains exactly one edge e with label l where $s_H(e) = \chi_H$, $ar(e) = n$ and[3] $V_H = Set(\chi_H)$ (see (b), nodes are ordered from left to right).



The next step is to define a method (first introduced in [Kön99b]) for the annotation of hypergraphs with lattice elements and to describe how these annotations change under morphisms. We use annotated hypergraphs as types where the annotations can be considered as extra typing information, therefore we use the terms *annotated hypergraph* and *type graph* as synonyms.

Definition 10.2.2 (Annotated Hypergraphs) *Let $\mathcal{A}$ be a mapping assigning a lattice $\mathcal{A}(H) = (I, \leq)$ to every hypergraph and a function $\mathcal{A}_\varphi : \mathcal{A}(H) \to \mathcal{A}(H')$ to every morphism $\varphi : H \to H'$. We assume that $\mathcal{A}$ satisfies:*

$$\mathcal{A}_\varphi \circ \mathcal{A}_\psi = \mathcal{A}_{\varphi \circ \psi} \qquad \mathcal{A}_{id_H} = id_{\mathcal{A}(H)} \qquad \mathcal{A}_\varphi(a \vee b) = \mathcal{A}_\varphi(a) \vee \mathcal{A}_\varphi(b) \qquad \mathcal{A}_\varphi(\bot) = \bot$$

where $\vee$ is the join-operation, a and b are two elements of the lattice $\mathcal{A}(H)$ and $\bot$ is its bottom element.

If $a \in \mathcal{A}(H)$, then $H[a]$ is called an annotated hypergraph. And $\varphi : H[a] \to_{\mathcal{A}} H'[a']$ is called an $\mathcal{A}$-morphism if $\varphi : H \to H'$ is a hypergraph morphism and $\mathcal{A}_\varphi(a) \leq a'$. Furthermore $H[a]$ and $H'[a']$ are called isomorphic if there is a strong bijective $\mathcal{A}$-morphism φ with $\mathcal{A}_\varphi(a) = a'$ between them.

Example: We consider the following annotation mapping $\mathcal{A}$: let $(\{\mathit{false}, \mathit{true}\}, \leq)$ be the boolean lattice where $\mathit{false} < \mathit{true}$. We define $\mathcal{A}(H)$ to be the set of all mappings from V_H into $\{\mathit{false}, \mathit{true}\}$ (which yields a lattice with pointwise order). By choosing an element of $\mathcal{A}(H)$ we fix a subset of the nodes. So let $a : V_H \to \{\mathit{false}, \mathit{true}\}$ be an element of $\mathcal{A}(H)$ and let $\varphi : H \to H'$, $v' \in V_H$. We define: $\mathcal{A}_\varphi(a) = a'$ where $a'(v') = \bigvee_{\varphi(v)=v'} a(v)$. That is, if a node v with annotation *true* is mapped to a node v' by φ, the annotation of v' will also be *true*.

From the point of view of category theory, $\mathcal{A}$ is a functor from the category of hypergraphs and hypergraph morphisms into the category of lattices and join-morphisms (i.e. functions preserving the join operation of the lattice).

We now introduce a method for attaching (annotated) hypergraphs with a construction plan consisting of discrete graph morphisms.

Definition 10.2.3 (Hypergraph Construction) *Let $H_1[a_1], \ldots, H_n[a_n]$ be annotated hypergraphs and let $\zeta_i : \mathbf{m_i} \to D$, $1 \leq i \leq n$ be hypergraph morphisms where $ar(H_i) = m_i$ and D is discrete. Furthermore let $\varphi_i : \mathbf{m_i} \twoheadrightarrow H_i$ be the unique strong morphisms.*

[3] $Set(\tilde{s})$ is the set of all elements of a string $\tilde{s}$

For this construction we assume that the node and edge sets of $H_1, \ldots, H_n$ and D are pairwise disjoint. Furthermore let $\approx$ be the smallest equivalence on their nodes satisfying $\zeta_i(v) \approx \varphi_i(v)$ if $1 \leq i \leq n$, $v \in V_{\mathbf{m_i}}$. The nodes of the constructed graph are the equivalence classes of $\approx$. We define

$$\textcircled{\scriptsize D}_{i=1}^{n}(H_i, \zeta_i) = ((V_D \cup \bigcup\nolimits_{i=1}^{n} V_{H_i})/\approx, \bigcup\nolimits_{i=1}^{n} E_{H_i}, s_H, l_H, \chi_H)$$

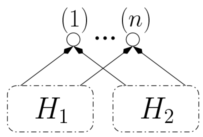
where $s_H(e) = [v_1]_\approx \ldots [v_k]_\approx$ if $e \in E_{H_i}$ and $s_{H_i}(e) = v_1 \ldots v_k$. Furthermore $l_H(e) = l_{H_i}(e)$ if $e \in E_{H_i}$. And we define $\chi_H = [v_1]_\approx \ldots [v_k]_\approx$ if $\chi_D = v_1 \ldots v_k$.

If $n = 0$, the result of the construction is D itself.

We construct embeddings $\varphi : D \twoheadrightarrow H$ and $\eta_i : H_i \to H$ by mapping every node to its equivalence class and every edge to itself. Then the construction of annotated graphs can be defined as follows:

$$\textcircled{\scriptsize D}_{i=1}^{n}(H_i[a_i], \zeta_i) = \left(\textcircled{\scriptsize D}_{i=1}^{n}(H_i, \zeta_i)\right) \left[\bigvee\nolimits_{i=1}^{n} \mathcal{A}_{\eta_i}(a_i)\right]$$

In other words: we join all graphs $D, H_1, \ldots, H_n$ and fuse exactly the nodes which are the image of one and the same node in the $\mathbf{m_i}$, χ_D becomes the new sequence of external nodes. Lattice annotations are joined if the annotated nodes are merged. In terms of category theory, $\textcircled{\scriptsize D}_{i=1}^{n}(H_i[a_i], \zeta_i)$ is the colimit of the ζ_i and the φ_i regarded as $\mathcal{A}$-morphisms (D and the $\mathbf{m_i}$ are annotated with the bottom element $\perp$). We do not mention this fact in the rest of the paper, but it is used extensively in the proofs (for the proofs and several examples see the full version [Kön00b]).



We also use another, more intuitive notation for graph construction. Let $\zeta_i : \mathbf{m_i} \to D$, $1 \leq i \leq n$. Then we depict $\textcircled{\scriptsize D}_{i=1}^{n}(H_i, \zeta_i)$ by drawing the hypergraph $(V_D, \{e_1, \ldots, e_n\}, s_H, l_H, \chi_D)$ where $s_H(e_i) = \zeta_i(\chi_{\mathbf{m_i}})$ and $l_H(e_i) = H_i$.

Example: we can draw $\textcircled{\scriptsize n}_{i=1}^{2}(H_i, \zeta_i)$ where $\zeta_1, \zeta_2 : \mathbf{n} \twoheadrightarrow \mathbf{n}$ as in the picture above (note that the edges have dashed lines). Here we fuse the external nodes of H_1 and H_2 in their respective order and denote the resulting graph by $H_1 \Box H_2$. If there is an edge with a dashed line labelled with an edge $[l]_n$ we rather draw it with a solid line and label it with l (see e.g. the second figure in section 10.4.1).

Definition 10.2.4 (Hypergraph Rewriting) *Let $\mathcal{R}$ be a set of pairs (L, R) (called rewriting rules), where the left-hand side L and the right-hand side R are both hypergraphs of the same arity. Then $\to_\mathcal{R}$ is the smallest relation generated by the pairs of $\mathcal{R}$ and closed under hypergraph construction.*

In our approach we generate the same transition system as in the double-pushout approach to graph rewriting described in [Ehr79] (for details see [Kön00d]).

We need one more concept: a linear mapping which is an inductively defined transformation, mapping hypergraphs to hypergraphs and adding annotation.

Definition 10.2.5 (Linear Mapping) *A function from hypergraphs to hypergraphs is called arity-preserving if it preserves arity and isomorphism classes of hypergraphs.*

Let t be an arity-preserving function that maps hypergraphs of the form $[l]_n$ to annotated hypergraphs. Then t can be extended to arbitrary hypergraphs by defining $t(\textcircled{D}_{i=1}^{n}([l_i]_{n_i}, \zeta_i)) = \textcircled{D}_{i=1}^{n}(t([l_i]_{n_i}), \zeta_i)$ and is then called a linear mapping.

10.3 Static Analysis and Type Systems for Graph Rewriting

Having introduced all underlying notions we now specify the requirements for type systems. We assume that there is a fixed set $\mathcal{R}$ of rewrite rules, an annotation mapping $\mathcal{A}$, a predicate X on hypergraphs (representing the property we want to check) and a relation $\triangleright$ with the following meaning: if $H \triangleright T$ where H is a hypergraph and T a type graph (annotated wrt. to $\mathcal{A}$), then H has type T. It is required that H and T have the same arity.

We demand that $\triangleright$ satisfies the following conditions: first, a type should contain information concerning the properties of a hypergraph, i.e. if a hypergraph has a type, then we can be sure that the property X holds.

$$H \triangleright T \;\Rightarrow\; X(H) \quad \textbf{(correctness)} \tag{10.1}$$

During reduction, the type stays invariant.

$$H \triangleright T \wedge H \rightarrow_{\mathcal{R}} H' \;\Rightarrow\; H' \triangleright T \quad \textbf{(subject reduction property)} \tag{10.2}$$

From (10.1) and (10.2) we can conclude that $H \triangleright T$ and $H \rightarrow^{*}_{\mathcal{R}} H'$ imply $X(H')$, that is X holds during the entire reduction.

The strong $\mathcal{A}$-morphisms introduced in Definition 10.2.2 impose a preorder on type graphs. It should always be possible to weaken the type with respect to that preorder.

$$H \triangleright T \wedge T \twoheadrightarrow_{\mathcal{A}} T' \;\Rightarrow\; H \triangleright T' \quad \textbf{(weakening)} \tag{10.3}$$

We also demand that the type system is compositional, i.e a graph has a type if and only if this type can be obtained by typing its subgraphs and combining these types. We can not sensibly demand that the type of an expression is obtained by combining the types of the subgraphs in exactly the same way the expression is constructed, so we introduce a partial arity-preserving mapping f doing some post-processing.

$$\begin{gathered}\forall\, i\colon H_i \triangleright T_i \;\Rightarrow \textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright f(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \\ \textcircled{D}_{i=1}^{n}(H_i, \zeta_i) \triangleright T \;\Rightarrow\; \exists\, T_i\colon (H_i \triangleright T_i \text{ and } f(\textcircled{D}_{i=1}^{n}(T_i, \zeta_i)) \twoheadrightarrow_{\mathcal{A}} T) \\ \textbf{(compositionality)}\end{gathered} \tag{10.4}$$

A last condition—the existence of minimal types—may not be strictly needed for type systems, but type systems satisfying this condition are much easier to handle.

$$H \text{ typable} \;\Rightarrow\; \exists T\colon \big(H \triangleright T \land (H \triangleright T' \iff T \twoheadrightarrow_{\mathcal{A}} T')\big) \quad \textbf{(minimal types)} \tag{10.5}$$

Let us now assume that types are computed from graphs in the following way: there is a linear mapping t, such that $H \triangleright f(t(H))$, if $f(t(H))$ is defined, and all other types of H are derived by the weakening rule, i.e. $f(t(H))$ is the minimal type of H.

The meaning of the mappings t and f can be explained as follows: t is a transformation *local* to edges, abstracting from irrelevant details and adding annotation information to a graph. The mapping f on the other hand, is a *global* operation, merging or removing parts of a graph in order to anticipate future reductions and thus ensure the subject reduction property. In the example in section 10.4.1 f "folds" a graph into itself, hence the letter f. In order to obtain compositionality, it is required that f can be applied arbitrarily often at any stage of type inference, without losing information (see condition (10.6) of Theorem 10.3.1).

In this setting it is sufficient to prove some simpler conditions, especially the proof of (10.2) can be conducted locally.

Theorem 10.3.1 *Let $\mathcal{A}$ be a fixed annotation mapping, let f be an arity-preserving mapping as above, let t be a linear mapping, let X be a predicate on hypergraphs and let $H \triangleright T$ if and only if $f(t(H)) \twoheadrightarrow_{\mathcal{A}} T$. Let us further assume that f satisfies*[4]

$$f(ⓓ_{i=1}^{n}(T_i,\zeta_i)) \cong f(ⓓ_{i=1}^{n}(f(T_i),\zeta_i)) \tag{10.6}$$
$$T \twoheadrightarrow_{\mathcal{A}} T' \Rightarrow f(T) \twoheadrightarrow_{\mathcal{A}} f(T') \tag{10.7}$$

Then the relation $\triangleright$ satisfies conditions (10.1)–(10.5) if and only if it satisfies

$$f(t(H)) \textit{ defined} \Rightarrow X(H) \tag{10.8}$$
$$(L,R) \in \mathcal{R} \Rightarrow f(t(R)) \twoheadrightarrow_{\mathcal{A}} f(t(L)) \tag{10.9}$$

The operation f can often be characterised by a universal property with the intuitive notion that $f(T)$ is the "smallest" type graph (wrt. the preorder $\twoheadrightarrow_{\mathcal{A}}$) for which $T \twoheadrightarrow_{\mathcal{A}} f(T)$ and a property C hold.

Proposition 10.3.2 *Let C be a property on type graphs such that $f(T)$ can be characterised in the following way: $f(T)$ satisfies C, there is a morphism $\varphi : T \twoheadrightarrow_{\mathcal{A}} f(T)$ and for every other morphism $\varphi' : T \to_{\mathcal{A}} T'$ where $C(T')$ holds, there is a unique morphism $\psi : f(T) \to_{\mathcal{A}} T'$ such that $\psi \circ \varphi = \varphi'$. Furthermore we demand that if there exists a morphism $\varphi : T \to_{\mathcal{A}} T'$ such that $C(T')$ holds, then $f(T)$ is defined.*

[4]In an equation $T \cong T'$ we assume that T is defined if and only if T' is defined. And in a condition of the form $T \twoheadrightarrow_{\mathcal{A}} T'$ we assume that T is defined if T' is defined.

Then if $f(T)$ is defined, it is unique up to isomorphism. Furthermore f satisfies conditions (10.6) and (10.7).

10.4 Case Studies

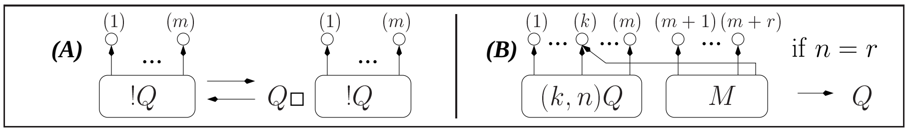
10.4.1 A Type System for the Polyadic π-Calculus

We present a graph rewriting semantics for the asynchronous polyadic π-calculus [Mil93] without choice and matching, already introduced in [Kön00c]. Different ways of encoding the π-calculus into graph rewriting can be found in [Yos94, Gar99, GM02].

We apply the theory presented in section 10.3, introduce a type system avoiding runtime errors produced by mismatching arities and show that it satisfies the conditions of Theorem 10.3.1. Afterwards we show that a graph has a type if and only if the corresponding π-calculus process has a type in a standard type system with infinite regular trees.

Definition 10.4.1 (Process Graphs) *A* process graph *P is inductively defined as follows: P is a hypergraph with a duplicate-free string of external nodes. Furthermore each edge e is either labelled with $(k,n)Q$ where Q is again a process graph, $1 \leq n \leq ar(Q)$ and $1 \leq k \leq ar(e) = ar(Q) - n$ (e is a process waiting for a message with n ports arriving at its k-th node), with $!Q$ where $ar(Q) = ar(e)$ (e is a process which can replicate itself) or with the constant M (e is a message sent to its last node).*

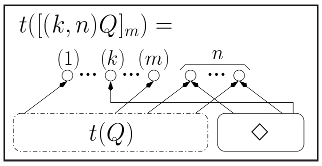
The reduction relation is generated by the rules in (A) (replication) and by rule (B) (reception of a message by a process) and is closed under isomorphism and graph construction.



A process graph may contain a bad redex, if it contains a subgraph corresponding to the left-hand side of rule (B) with $n \neq r$, so we define the predicate X as follows: $X(P)$ if and only if P does not contain a bad redex.

We now propose a type system for process graphs by defining the mappings t and f. (Note that in this case, the type graphs are trivially annotated by $\perp$, and so we omit the annotation mapping.)

The linear mapping t is defined on the hyperedges as follows: $t([M]_n) = [\diamond]_n$ ($\diamond$ is a new edge label), $t([!Q]_m) = t(Q)$ and $t([(k,n)Q]_m)$ is defined as in the image to the right (in the notation explained after Definition 10.2.3). It is only defined if $n + m = ar(Q)$.



The mapping f is defined as in Proposition 10.3.2 where C is defined as follows[5]

$$C(T) \iff \forall e_1, e_2 \in E_T \colon (\lfloor s_T(e_1) \rfloor_{ar(e_1)} = \lfloor s_T(e_2) \rfloor_{ar(e_2)} \;\Rightarrow\; e_1 = e_2)$$

The linear mapping t extracts the communication structure from a process graph, i.e. an edge of the form $[\Diamond]_n$ indicates that its nodes (except the last) might be sent or received via its last node. Then f makes sure that the arity of the arriving message matches the expected arity and that nodes that might get fused during reduction are already fused in $f(t(H))$.

Proposition 10.4.2 *The trivial annotation mapping $\mathcal{A}$ (where every lattice consists of a single element $\bot$), the mappings f and t and the predicate X defined above satisfy conditions (10.6)–(10.9) of Theorem 10.3.1. Thus if $P \triangleright T$, then P will never produce a bad redex during reduction.*

We now compare our type system to a standard type system of the π-calculus. An encoding of process graphs into the asynchronous π-calculus can be defined as follows.

Definition 10.4.3 (Encoding) *Let P be a process graph, let $\mathcal{N}$ be the name set of the π-calculus and let $\tilde{t} \in \mathcal{N}^*$ such that $|\tilde{t}| = ar(P)$. We define $\Theta_{\tilde{t}}(P)$ inductively as follows:*

$$\Theta_{a_1 \dots a_{n+1}}([M]_{n+1}) = \overline{a_{n+1}}\langle a_1, \dots, a_n \rangle \qquad \Theta_{\tilde{t}}([!Q]_m) = !\,\Theta_{\tilde{t}}(Q)$$

$$\Theta_{a_1 \dots a_m}([(k,n)Q]_m) = a_k(x_1, \dots, x_n).\Theta_{a_1 \dots a_m x_1 \dots x_n}(Q)$$

$$\Theta_{\tilde{t}}(\textcircled{\scriptsize D}_{i=1}^{n}(P_i, \zeta_i)) = (\nu\, \mu(V_D \backslash Set(\chi_D)))(\Theta_{\mu(\zeta_1(\chi_{\mathbf{m_1}}))}(P_1) \mid \dots \mid \Theta_{\mu(\zeta_n(\chi_{\mathbf{m_n}}))}(P_n))$$

where $\zeta_i : \mathbf{m_i} \to D$, $1 \le i \le n$ and $\mu : V_D \to \mathcal{N}$ is a mapping such that μ restricted to $V_D \backslash Set(\chi_D)$ is injective, $\mu(V_D \backslash Set(\chi_D)) \cap \mu(Set(\chi_D)) = \emptyset$ and $\mu(\chi_D) = \tilde{t}$. Furthermore the $x_1, \dots, x_n \in \mathcal{N}$ are fresh names.

The encoding of a discrete graph is included in the last case, if we set $n = 0$ and assume that the empty parallel composition yields the nil process $\mathbf{0}$.

An operational correspondence can be stated as follows:

Proposition 10.4.4 *Let p be an arbitrary expression in the asynchronous polyadic π-calculus without summation. Then there exists a process graph P and a duplicate-free string $\tilde{t} \in \mathcal{N}^*$ such that $\Theta_{\tilde{t}}(P) \equiv p$. Furthermore for process graphs P, P' and for every duplicate-free string $\tilde{t} \in \mathcal{N}^*$ with $|\tilde{t}| = ar(P) = ar(P')$ it is true that:*

- *$P \cong P'$ implies $\Theta_{\tilde{t}}(P) \equiv \Theta_{\tilde{t}}(P')$*
- *$P \to^* P'$ implies $\Theta_{\tilde{t}}(P) \to^* \Theta_{\tilde{t}}(P)$*
- *$\Theta_{\tilde{t}}(P) \to^* p \neq wrong$ implies that $P \to^* Q$ and $\Theta_{\tilde{t}}(Q) \equiv p$ for some process graph Q.*

[5] $\lfloor s \rfloor_i$ extracts the i-th element of a string s.

- $\Theta_{\tilde{t}}(P) \to^* wrong$ *if and only if* $P \to^* P'$ *for some process graph* P' *containing a bad redex*

We now compare our type system with a standard type system of the π-calculus: a *type tree* is a potentially infinite ordered tree with only finitely many non-isomorphic subtrees. A type tree is represented by the tuple $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are again type trees, the children of the root. A type assignment $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ assigns names to type trees where $\Gamma(x_i) = t_i$. The rules of the type system are simplified versions of the ones from [PS93], obtained by removing the subtyping annotations.

$$\Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \mid q} \qquad \frac{\Gamma \vdash p}{\Gamma \vdash\, !p} \qquad \frac{\Gamma, a : t \vdash p}{\Gamma \vdash (\nu a)p}$$

$$\frac{\Gamma(a) = [t_1, \ldots, t_m] \quad \Gamma, x_1 : t_1, \ldots, x_m : t_m \vdash p}{\Gamma \vdash a(x_1, \ldots, x_m).p} \qquad \frac{\Gamma(a) = [\Gamma(a_1) \ldots, \Gamma(a_m)]}{\Gamma \vdash \overline{a}\langle a_1, \ldots, a_m\rangle}$$

We will now show that if a process graph has a type, then its encoding has a type in the π-calculus type system and vice versa. In order to express this we first describe the unfolding of a type graph into type trees.

Proposition 10.4.5 *Let T be a type graph and let σ be a mapping from V_T into the set of type trees. The mapping σ is called consistent, if it satisfies for every edge $e \in E_T$: $s_T(e) = v_1 \ldots v_n v \ \Rightarrow\ \sigma(v) = [\sigma(v_1), \ldots, \sigma(v_n)]$. Every type graph of the form $f(t(P))$ has such a consistent mapping.*

Let $P \triangleright T$ with $n = ar(T)$ and let σ be a consistent mapping for T. Then it holds for every duplicate-free string $\tilde{t}$ of length n that $\lfloor \tilde{t} \rfloor_1 : \sigma(\lfloor \chi_T \rfloor_1), \ldots, \lfloor \tilde{t} \rfloor_n : \sigma(\lfloor \chi_T \rfloor_n) \vdash \Theta_{\tilde{t}}(P)$.

Now let $\Gamma \vdash \Theta_{\tilde{t}}(P)$. Then there exists a type graph T such that $P \triangleright T$ and a consistent mapping σ such that for every $1 \leq i \leq |\tilde{t}|$ it holds that $\sigma(\lfloor \chi_T \rfloor_i) = \Gamma(\lfloor \tilde{t} \rfloor_i)$.

10.4.2 Concurrent Object-Oriented Programming

We now show how to model a concurrent object-oriented system by graph rewriting and then present a type system. In our model, several objects may compete in order to receive a message, and several messages might be waiting at the same object. Typically, type systems in object-oriented programming are there to ensure that an object that receives a message is able to process it.

Definition 10.4.6 (Concurrent object-oriented rewrite system) *Let $(\mathcal{C}, <:)$ be a lattice of classes with a top class*[6] *$\top$ and a bottom class $\bot$. We denote classes by the letters $A, B, C, \ldots$. Furthermore let $\mathcal{M}$ be a set of method names. The function $ar : \mathcal{C} \cup \mathcal{M} \to \mathbb{N}\backslash\{0\}$ assigns an arity to every class or method name.*

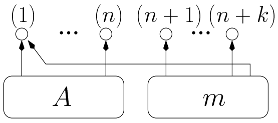
An object graph G is a hypergraph with a duplicate-free string of external nodes, labelled with elements of $\mathcal{C}\backslash\{\bot\} \cup \mathcal{M}$ where for every edge e it holds that

[6] This corresponds to the class Object in Java

$ar(e) = ar(l_G(e))$. A concurrent object-oriented rewrite system (specifying the semantics) consists of a set of rules $\mathcal{R}$ satisfying the following conditions:

- *the left-hand side of a rule always has the form shown in Figure (C) below (where $A \in \mathcal{C}\backslash\{\perp\}$, $ar(A) = n$, $m \in \mathcal{M}$, $ar(m) = k + 1$).*

The right-hand side is again an object graph of arity $n + k$. If a left-hand side $R_{A,m}$ exists, we say that A understands m.



(C)

$= R_{A,m}$

- *If $A <: B$, $A \neq \perp$ and B understands m, then A also understands m.*

- *For all $m \in \mathcal{M}$, either $\{A \mid A$ understands $m\}$ is empty or it contains a greatest element.*

An object graph G contains a "message not understood"-error if G contains a subgraph $R_{A,m}$, but A does not understand m.

Thus the predicate X for this section is defined as follows: $X(G)$ if and only if G does not contain a "message not understood"-error.

In contrast to the previous section, we now use annotated type graphs: the annotation mapping $\mathcal{A}$ assigns a lattice $(\{a : V_H \to \mathcal{C} \times \mathcal{C}\}, \leq))$ to every hypergraph H. The partial order is defined as follows: $a_1 \leq a_2 \iff \forall v\colon (a_1(v) = (A_1, B_2) \wedge a_2(v) = (A_2, B_2) \Rightarrow A_1 <: A_2 \wedge B_1 :> B_2)$, i.e. we have covariance in the first and contravariance in the second position. If a node v is labelled (A, B), this has the following intuitive meaning: we can accept at least as many messages as an object of class A on this node *and* we can send at most as many messages as an object of class B can accept.

Furthermore we define $\mathcal{A}_\varphi(a)(v') = \bigvee_{\varphi(v)=v'} a(v)$ where $\varphi : H \to H'$, a is an element of $\mathcal{A}(H)$ and $v' \in V_{H'}$.

We now define the operator f: let $T[a]$ be a type graph of arity n where it holds for all nodes v that $a(v) = (A, B)$ implies $A <: B$ (otherwise f is undefined). Then f reduces the graph to its string of external nodes, i.e $f(T[a]) = \mathbf{n}[b]$ where $b(\lfloor \chi_{\mathbf{n}} \rfloor_i) = a(\lfloor \chi_T \rfloor_i)$.

The linear mapping t determines the type of a class or method. It is necessary to choose a linear mapping that preserves the interface of left-hand and right-hand sides, i.e. we can use any t that satisfies condition (10.9) and the following two conditions below for $A \in \mathcal{C}\backslash\{\perp\}$ and $m \in \mathcal{M}$:

$$\begin{aligned} t([A]_n) &= [A]_n[a] \quad \text{where} \quad a(\lfloor \chi_{[A]_n} \rfloor_1) \geq (A, \top) \\ t([m]_n) &= [m]_n[a] \quad \text{where} \quad a(\lfloor \chi_{[m]_n} \rfloor_n) \geq (\perp, \max\{B \mid B \text{ understands } m\}) \end{aligned}$$

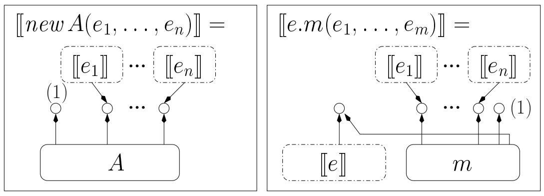
Proposition 10.4.7 *The annotation mapping $\mathcal{A}$, the mappings f and t and the predicate X defined above satisfy conditions (10.6)–(10.9) of Theorem 10.3.1. Thus if $G \triangleright T$, then G will never produce a "message not understood"-error during reduction.*

In this case we do not prove that this type systems corresponds to an object-oriented type system, but rather present a semi-formal argument: we give the syntax and a type system for a small object calculus, and furthermore an encoding into hypergraphs, without really defining the semantics. For the formal semantics of object calculi see [Wal95, IPW99], among others.

An expression e in the object calculus either has the form $new\ A(e_1, \ldots, e_n)$ where $A \in \mathcal{C}\backslash\{\perp\}$ and $ar(A) = n + 1$ or $e.m(e_1, \ldots, e_n)$ where $m \in \mathcal{M}$ and $ar(m) = n + 2$. The e_i are again expressions. Every class A is assigned an $(ar(A)-1)$-tuple of classes defining the type of the fields of A ($A : (A_1, \ldots, A_n)$) and every method m with $ar(m) = n + 2$ defined in class B is assigned a type $B.m : C_1, \ldots, C_n \to C$. If a method is overwritten in a subclass it is required to have the same type. A simple type systems looks as follows:

$$\frac{e : A,\ A <: B}{e : B} \qquad \frac{A : (A_1, \ldots, A_n),\ e_i : A_i}{new\ A(e_1, \ldots, e_n) : A}$$
$$\frac{e : B,\ B.m : C_1, \ldots, C_n \to C,\ e_i : C_i}{e.m(e_1, \ldots, e_n) : C}$$

Now an encoding $[\![\cdot]\!]$ can be defined as shown in the figure to the right. We introduce the convention that the penultimate node of a message can be used to access the result after the rewriting step.



If $A : (A_1, \ldots, A_n)$ we define t in such a way that the $n + 1$ external nodes of $t([A]_{n+1})$ are annotated by $(A, \top)$, $(\perp, A_1)$, $\ldots$, $(\perp, A_n)$. And if $B.m : C_1, \ldots, C_n \to C$ (where B is the maximal class which understands method m), we annotate the external nodes of $t([m]_{n+2})$ by $(\perp, C_1)$, $\ldots$, $(\perp, C_n)$, $(C, \top)$, $(\perp, B)$. Now we can show by induction on the typing rules that if $e : A$, then there exists a type graph $T[a]$ such that $[\![e]\!] \triangleright T[a]$ and $a(\lfloor \chi_T \rfloor_1) = (A, \top)$.

10.5 Conclusion and Comparison to Related Work

This is a first tentative approach aimed at developing a general framework for the static analysis of graph rewriting in the context of type systems. It is obvious that there are many type systems which do not fit well into our proposal. But since we are able to capture the essence of two important type systems, we assume to be on the right track.

Types are often used to make the connection of components and the flow of information through a system explicit (see e.g. the type system for the π-calculus, where the type trees indicate which tuple of channels is sent via which channel). Since connections are already explicit in graphs, we can use them both as type and as the expression to be typed. Via morphisms we can establish a clear connection between an expression and its type. Graphs are furthermore useful since we can easily add an extra layer of annotation.

Work that is very close in spirit to ours is [Hon96] by Honda which also presents a general framework for type systems. The underlying model is closer to standard process algebras and the main focus is on the characterisation and classification of type systems.

The idea of composing graphs in such a way that they satisfy a certain property was already presented by Lafont in [Laf90] where it is used to obtain deadlock-free nets.

In graph rewriting there already exists a concept of typed graphs [CMR96], related to ours, but nevertheless different. In that work, a type graph is fixed a priori and there is only one type graph for every set of productions. Graphs are considered valid only if they can be mapped into the type graph by a graph morphism (this is similar to our proposal). In our case, we compute the type graphs a posteriori and it is a crucial point in the design of every type system to distinguish as many graphs as possible by assigning different type graphs to them.

This paper is a continuation of the work presented in [Kön99b] where the idea of generic type systems for process graphs (as defined in section 10.4.1) was introduced, but no proof of the equivalence of our type system to the standard type system for the π-calculus was given. The ideas presented there are now extended to general graph rewriting systems.

Further work will consist in better understanding the underlying mechanism of the type system. An interesting question in this context is the following: given a set of rewrite rules, is it possible to automatically derive mappings f and t satisfying the conditions of Theorem 10.3.1?

Acknowledgements: I would like to thank Reiko Heckel and Andrea Corradini for their comments on drafts of this paper, and Tobias Nipkow for his advice. I am also grateful to the anonymous referees for their valuable comments.

Chapter 11

Analysing Input/Output-Capabilities of Mobile Processes with a Generic Type System

Abstract

We introduce a generic type system for the synchronous polyadic π-calculus, allowing us to mechanise the analysis of input/output capabilities of mobile processes. The parameter of the generic type system is a lattice-ordered monoid, the elements of which are used to describe the capabilities of channels with respect to their input/output-capabilities. The type system can be instantiated in order to check process properties such as upper and lower bounds on the number of processes concurrently using a channel, confluence and absence of blocked processes.

11.1 Introduction

With the increasing connection of computers and networks, the treatment of code and process mobility is becoming more and more important. The increasing importance of this topic brings about many challenges concerning analysis and verification of mobile processes.

For the analysis and verification of processes and programs in general there are basically two approaches: methods that are complete but cannot be fully mechanised, and fully automatic methods, which—because of undecidability issues—cannot be complete, i.e., not all processes satisfying the property to be checked are accepted. There is a variety of methods which can be seen as variants of the latter approach and which can be summarised as static analysis techniques [NNH99].

One promising direction, especially for the π-calculus—a calculus describing communicating mobile processes—is to use type or sort systems and type inference with rather complex types abstracting from process behaviour. In the last few years there have been several papers presenting such type systems for the polyadic π-calculus and other process calculi, checking e.g., input/output behaviour [PS96, KPT99], absence of deadlocks and livelocks [Kob98, Kob02], security properties [Aba99, BDNN98, HVY00, HR00], allocation of permissions to names [RH97] and many others. There are also interesting type systems for higher-order variants of the π-calculus [YH02]. Types are compositional and thus allow reuse of information obtained in the analysis of smaller subsystems. Because of name mobility in the π-calculus, the difficulty in typing processes lies in tracking the capabilities of names in a mobile environment.

One drawback of the type systems mentioned above is the fact that they are specialised to check very specific properties. A much more general approach is a theory of types by Honda [Hon96], which is based on typed algebras and gives a classification of type systems. This theory is very general and it is thus necessary to prove the subject reduction property and the correctness of a type system for every instance.

Another generic approach is a type system by Kobayashi and Igarashi [IK01], which assigns to each π-calculus process an abstract process of a simpler process calculus, from the possible reductions of which one can infer properties of the original process. This technique seems to be very powerful, but the analysis of these abstract processes could be rather costly from a complexity point of view. A generic approach to resource usage analysis is shown in [KSS00].

Our contribution is to present a generic type system where the subject reduction property can be shown for the general case, and by instantiating the type system specific properties of processes can be analysed. With the introduction of residuation (explained below) we manage to derive tight upper and lower bounds for channel usage. The only paper we are aware of where this technique is also used is [KNY95].

We concentrate on properties connected to input/output capabilities of processes in the synchronous polyadic π-calculus. Our types can be seen as a generalization of the linear types presented in [KPT99].

In the examples (see Section 11.6) we check properties such as upper and lower bounds on the number of certain prefixes, confluence, absence of blocked input or output prefixes. Determining these capabilities of a process involves counting and we attempt to keep this concept as general as possible by basing the generic type system on commutative monoids. Instantiating a type system mainly involves choosing an appropriate monoid, and monoid elements associated with input and output prefixes (e.g. for counting the number of prefixes with a certain subject).

Instead of giving the precise answer to every question, the type system uses over-approximation (e.g. we can expect results of the form "there are at most two processes using channel x at any given time"). Hence plain monoids are not sufficient, but we need ordered monoids (so-called lattice-ordered monoids or ℓ-monoids), equipped with a partial order compatible with summation.

There is a huge class of lattice-ordered monoids which are residuated, i.e., some limited form of subtraction can be defined. Residuation can be put to good use in process analysis. Consider, e.g., the process $P = \overline{x}.x.\mathbf{0}$. While P increases the number of (future) occurrences of the output prefix $\overline{x}$ by one, it does not do so for the input prefix x, since we are interested in the number of prefixes not located underneath another prefix (i.e. in the maximal number of prefixes which are active at the same time) and x can only be reached by a communication with $\overline{x}$, which decreases the number of input prefixes in the environment by one. This decrease can be anticipated when typing P, and is taken into consideration by subtracting one from the number of input prefixes. This is a new feature, which does not occur in related type systems such as [KPT99] and which guarantees sharper bounds on the current capabilities of a process.

The type assigned to a channel x occurring free in process P will be of the form $[\tilde{t}]^a$, similar to the types in [KPT99]. The letter a stands for a monoid element which is expected to be an upper bound for the capabilities of channel x. Furthermore $\tilde{t} = t_1 \dots t_n$ is a sequence of types, telling us that n-tuples of channel names are communicated via x, where the i-th component of this tuple should have type t_i.

The aim of this paper is twofold: to show how capabilities of processes can be parameterised and bounded by using algebraic structures and to start developing a framework which can be used for verification and static analysis of mobile processes.

The rest of this paper is structured as follows: in Section 11.2 we discuss the general principles and ideas behind this work. Then, in Section 11.3, we introduce some preliminaries, by giving a short summary of the π-calculus, presenting lattice-ordered commutative monoids and defining the notion of type (and some operations on types). The type system itself is presented in Section 11.4 and the subject reduction property is shown for its most general version, i.e. for arbitrary ℓ-monoids. We then demonstrate how the type system can be used for process analysis (Section 11.5), by giving the connection between the type of a process and its input/output capabilities and by showing how type systems can be composed to form new type systems. Afterwards in Sections 11.6 and 11.7 we discuss some examples and related work.

11.2 General Ideas

In order to illustrate how our type system will look like and how algebraic structures can be used to describe capabilities of π-calculus processes, we give the following examples.

11.2.1 Upper Bounds

We regard the following π-calculus process $P = !a(x,y).\overline{x}\langle y\rangle \mid \overline{a}\langle b,c\rangle$, which reduces to $!a(x,y).\overline{x}\langle y\rangle \mid \overline{b}\langle c\rangle$, and wish to give upper bounds for its input/output behaviour.

The first step is to assign a type to each channel name which reflects the arity of the tuples communicated via this channel. The name c will never be used for communication, so we can assign an arbitrary type to c, for example

$c\colon []$, where $[]$ denotes the type of a channel over which empty sequences of values are sent. Furthermore c will be sent via channel b, so we expect the following type assignment: $b\colon [[]]$, meaning that a single name with type $[]$ will be sent via b. Finally this gives us the type assignment $a\colon [[[]],[]]$ for channel name a. This is a standard procedure, occurring in many type systems for the π-calculus.

We have so far neglected the monoid elements representing capabilities. Adding these elements m_a, m_b, m_c in retrospect gives us the type environment Γ shown below for process P.

$$\Gamma = a\colon [[[]^{m_c}]^{m_b}, []^{m_c}]^{m_a}, b\colon [[]^{m_c}]^{m_b}, c\colon []^{m_c}.$$

So in the second step we discuss which values these three monoid elements should have, if we attempt to derive information about the capabilities of the channel names of P.

We could, for example, suppose that, similar to [PS96], the capabilities are taken from a set $IO = \{\mathit{none}, \mathit{output}, \mathit{input}, \mathit{both}\}$ where the elements represent no input/output capability, output capability only, input capability only and both capabilities, respectively. These capabilities naturally form a lattice with an order $\leq$ satisfying $\mathit{none} \leq \mathit{output} \leq \mathit{both}$ and $\mathit{none} \leq \mathit{input} \leq \mathit{both}$. In this case the monoid operation coincides with the join operation of the lattice.

So in our example in a type environment for process P we expect the monoid elements m_a, m_b, m_c to have the following values:

$$m_a = \mathit{both} \quad m_b = \mathit{output} \quad m_c = \mathit{none}.$$

A somewhat more fine-grained analysis can be achieved by using pairs of natural numbers (including infinity) with the natural partial order to give an upper bound to the number of times a name can appear in an input or output prefix at any given point in any reduction sequence. The first element of a pair denotes output capability whereas the second element denotes input capability. In our example the best upper bound for the input capability of name a is ∞ since it is available infinitely often (because of the replication operator). For other names or other capabilities sharper bounds are possible:

$$m_a = (1,\infty) \quad m_b = (1,0) \quad m_c = (0,0).$$

Depending on the instantiation of the type system with the correct monoid, the typing rules that will be presented allow the derivation of the above types for process P.

11.2.2 Lower Bounds

One advantage of the general approach we are following here is the fact that the type system can be easily adapted, in this case to derive lower bounds instead of upper bounds. We consider the π-calculus process $Q = \overline{a}\langle b\rangle \mid a(x).\overline{x}.x \mid b$ with the reduction sequence $Q \to \overline{b}.b \mid b \to b$. We can observe that during the reduction of Q there will always be at least one input prefix with subject b.

By instantiating the type system with pairs of natural numbers as monoid elements (as above) and by using the inverted partial order $\geq$ instead of $\leq$, we obtain the following type environment:

$$\Gamma = a\colon [[\,]^{(0,0)}]^{(0,0)}, b\colon [\,]^{(-1,1)}.$$

That is, the type system is, in this case, able to correctly predict the lower bound 1 for the input capability of b. However, the lower bound for the output capability of b is -1 since Q is able to "snatch" away an output prefix of the form $\overline{b}$ from the environment by communication with b. The type of the names communicated via a is $[\,]^{(0,0)}$, which intuitively means that the input and output capabilities of x inside $a(x).\overline{x}.x$ "neutralise" each other.

We have presented very simple examples in order to give a flavour of the general ideas. More complex examples will be treated in Section 11.6.

11.3 Preliminaries

11.3.1 The π-Calculus

The π-calculus [MPW92, Mil93] is an influential paradigm describing communication and mobility of processes. In this paper we will consider the synchronous polyadic π-calculus without choice and matching where replication is only allowed for input prefixes. Its syntax is defined as follows:

$$P ::= \mathbf{0} \mid (\nu x\colon t)P \mid P_1|P_2 \mid \overline{x}\langle\tilde{z}\rangle.P \mid x(\tilde{y}).P \mid !x(\tilde{y}).P$$

where t is a type tree (see Definition 11.4.1) and x is taken from a fixed set of names N. Sequences of names from N are denoted by $\tilde{y} = y_1 \dots y_n$. We call $\overline{x}\langle\tilde{z}\rangle$ *output prefix* and $x(\tilde{y})$ *input prefix*.

The set of all free names (i.e. names not bound by either ν or by an input prefix) of a process P is denoted by $\mathit{fn}(P)$. The process obtained by replacing the free names y_i by x_i in P (and avoiding capture) is written $P\{\tilde{x}/\tilde{y}\}$.

Structural congruence is the smallest congruence obeying the rules in the upper part of Table 11.1, and equating processes that can be converted into one another by consistent renaming of bound names (α-conversion). The side condition of rule (C-Restr1) will become clear in Section 11.4 where we define type trees and the mapping *mon*. We use a reduction semantics as for the chemical abstract machine [BB92] instead of a labelled transition semantics (see Table 11.1).

Consider the following processes which we will use as an example in this paper (see Section 11.6). We omit the final $\mathbf{0}$.

$$F = c(r).\overline{d}\langle r\rangle.d(a).\overline{c}\langle a\rangle \qquad S = d(s).s(h_1,h_2).\overline{d}\langle h_1\rangle$$
$$T = \overline{c}\langle h\rangle.c(x) \qquad H = \overline{h}\langle i_1,i_2\rangle$$

There is a forwarder F which receives requests on a channel c, forwards them on a channel d to a server, receives the answer and sends it back on c. The server S receives requests on d. These requests come with a name s where the server

Structural Congruence:
(C-Com) $P_1 \mid P_2 \equiv P_2 \mid P_1$ (C-**0**) $P \mid \mathbf{0} \equiv P$
(C-Ass) $P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$
(C-Restr1) $(\nu x \colon t)\mathbf{0} \equiv \mathbf{0}$ if $mon(t) \geq 0$
(C-Restr2) $(\nu x \colon s)(\nu y \colon t)P \equiv (\nu y \colon t)(\nu x \colon s)P$ if $x \neq y$
(C-Restr3) $((\nu x \colon s)P_1) \mid P_2 \equiv (\nu x \colon s)(P_1 \mid P_2)$ if $x \notin fn(P_2)$
Reduction Rules:
(R-Comm) $\overline{x}\langle\tilde{z}\rangle.Q \mid x(\tilde{y}).P \to Q \mid P\{\tilde{z}/\tilde{y}\}$
(R-Rep) $\overline{x}\langle\tilde{z}\rangle.Q \mid\, !x(\tilde{y}).P \to Q \mid P\{\tilde{z}/\tilde{y}\} \mid\, !x(\tilde{y}).P$
(R-Par) $\dfrac{P \to P'}{P \mid Q \to P' \mid Q}$ (R-Restr) $\dfrac{P \to P'}{(\nu x \colon t)P \to (\nu x \colon t)P'}$
(R-Equ) $\dfrac{Q \equiv P, P \to P', P' \equiv Q'}{Q \to Q'}$

Table 11.1: Operational semantics of the π-calculus

can get further information. The server obtains this information, processes it and sends the answer back on d (in our example we keep the "processing part" very simple, the server just sends back the first component). Finally, T is a trigger process, starting the execution of F and receiving the result in the end, and H delivers information to the server.

We can combine the processes F, S, T, H to obtain P as the entire system. If we want F and S to be persistent, we use P'.

$$P = T \mid H \mid (\nu d \colon t)(F \mid S) \qquad P' = T \mid H \mid (\nu d \colon t)(!F \mid !S)$$

A programmer analysing this piece of code might be interested in the following properties: input/output behaviour, upper and lower bounds on the number of messages in a channel, confluence properties and absence of blocked prefixes that never find a communication partner. E.g., examining P will reveal that at any given time every name is used for input and output at most once and that P is therefore confluent.

11.3.2 Residuated Lattice-ordered Monoids

Lattice-ordered monoids are a well-developed mathematical concept (see e.g. [Bir67]). We are interested in commutative residuated ℓ-monoids in order to represent input/output capabilities.

Definition 11.3.1 (Lattice-ordered Monoid)

A commutative lattice-ordered monoid (ℓ-monoid) *is a tuple* $(I, +, \leq)$ *where* I *is a set,* $+ \colon I \times I \to I$ *is a binary operation and* $\leq$ *is a partial order which satisfy:*

- $(I, +)$ *is a commutative monoid, i.e.,* $+$ *is associative and commutative, and there is a unit* 0 *with* $0 + a = a$ *for every monoid element* $a \in I$*.*

- $(I, \leq)$ *is a lattice, i.e.,* $\leq$ *is a partial order, where two elements* $a, b \in I$ *have a* join *(or least upper bound)* $a \vee b$ *and a* meet *(or greatest lower bound)* $a \wedge b$*.*

- I *contains a bottom element* $\perp$*, the smallest element in* I*, and a top element* $\top$*, the greatest element in* I*.*

- *For* $a, b, c \in I$*:* $a + (b \vee c) = (a + b) \vee (a + c)$ *and* $a + (b \wedge c) = (a + b) \wedge (a + c)$*.*

For an element $a \in I$ *we define:* $sig(a) = \begin{cases} \perp & \text{if } a < 0, \\ 0 & \text{if } a = 0 \\ \top & \text{if } a > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$

If the partial order $\leq$ and the operator $+$ can be derived from the context, we will sometimes write I instead of $(I, +, \leq)$. For many of our examples a semi-lattice, i.e., a lattice for which only the join operation is defined, will be sufficient.

Definition 11.3.2 (Residuated ℓ-monoid) *Let* $(I, +, \leq)$ *be an* ℓ*-monoid and let* $a, b \in I$*. The* residual $a - b$ *is the smallest* x *(if it exists) such that* $a \leq x + b$*. A monoid* I *is called* residuated *if all residuals* $a - b$ *exist in* I *for* $a, b \in I$*.*

Examples: An ℓ-monoid which has already been introduced in Section 11.2.1, is $IO = (\{none, input, output, both\}, \vee, \leq)$ where $none \leq input \leq both$, $none \leq output \leq both$ and the monoid operation is the join, i.e., the ℓ-monoid degenerates to a lattice. This ℓ-monoid is residuated and it holds for example that $input - input = none$, $output - input = output$ and $both - input = output$.

In order to count the number of inputs or outputs we use the ℓ-monoid $\mathbb{Z}^\infty = (\mathbb{Z} \cup \{\infty, -\infty\}, +, \leq)$ with all integers including ∞ and $-\infty$. We define $\infty + (-\infty) = -\infty$ and otherwise summation works as expected. It is residuated and residuation corresponds to subtraction for all monoid elements different from ∞ and $-\infty$.

The cartesian product of two residuated ℓ-monoids, e.g. $\mathbb{Z}^\infty \times \mathbb{Z}^\infty$, is also a residuated ℓ-monoid.

We need the following laws concerning residuated ℓ-monoids.

Lemma 11.3.3 *For all elements* a, b, c *of a residuated* ℓ*-monoid it holds that*

$$a \leq (a - b) + b \qquad (a + b) - b \leq a \qquad (a + b) - c \leq (a - c) + b$$
$$(a + b) \vee 0 \leq (a \vee 0) + (b \vee 0) \qquad \perp + \perp = \perp \qquad \top + \top = \top$$

Furthermore the monoid operation $+$ is monotone, i.e., $a \leq a'$ and $b \leq b'$ imply $a + b \leq a' + b'$.

Proof: Most of the equations and inequations are straightforward to prove. We show only a proof of $(a + b) - b \leq a$ and refer the reader to [Bir67] for proofs of the other laws.

$(a + b) - b$ is the smallest x such that $a + b \leq x + b$. But if we set $x = a$ the inequation holds. So it must be the case that $(a + b) - b \leq a$.

There are cases where the inequation is strict. If we consider, for example, the ℓ-monoid IO defined above and set $a = \mathit{input}$, $b = \mathit{input}$, we obtain $(\mathit{input} + \mathit{input}) - \mathit{input} = \mathit{none} < \mathit{input}$.

Monotonicity follows from the ℓ-monoid law $a + (b \vee c) = (a + b) \vee (a + c)$. If $b \leq b'$, then $b \vee b' = b'$, therefore $a + b \leq (a + b) \vee (a + b') = a + (b \vee b') = a + b'$. $\square$

11.4 The Type System and its Properties

We will first define the notion of type tree and type environment and introduce some simple operations on types. We assume that a fixed ℓ-monoid $(I, +, \leq)$ is given.

Definition 11.4.1 (Type Tree) *A type tree (or type) t is of the form $[t_1, \ldots, t_n]^a$ where $a \in I$ and $t_1, \ldots, t_n$ are again type trees. This structure is potentially infinite. Furthermore we will often abbreviate the sequence $t_1, \ldots, t_n$ by $\tilde{t}$. We define $mon([\tilde{t}]^a) = a$.*

Summation on type trees is defined as follows:

$$[t_1, \ldots, t_n]^a \circledast [t_1, \ldots, t_n]^b = [t_1, \ldots, t_n]^{a+b}.$$

In this case we say that the two types are compatible. In all other cases summation is undefined.

The following definition of type environments is standard. The summation operation is taken from [KPT99]. Note that in the definition of $\Gamma, x\colon t$ below we do not require that x does not occur in Γ. Instead the assignment to x is overwritten by this operation, a fact that will be important for the typing rules introduced in Table 11.2. This choice allows us to avoid an explicit weakening rule and thus enables an easier handling of the induction in the proof of the subject reduction property (see Theorem 11.4.8).

Definition 11.4.2 (Type Environment) *A type environment Γ is of the form*

$$\Gamma = x_1\colon t_1, \ldots, x_n\colon t_n,$$

where $x_1, \ldots, x_n$ are distinct names and $t_1, \ldots, t_n$ are type trees. We also write $\Gamma(x_i) = t_i$ and define $dom(\Gamma) = \{x_1, \ldots, x_n\}$. By $\Gamma, x\colon t$ we denote the type environment Γ where any assignment to the variable x, if it should exist, is *overwritten by $x: t$. For $x \in N$, we denote by $\Gamma \backslash x$ the type environment Γ from which any assignment to the name x is deleted.*

Now let Γ_1, Γ_2 be two type environments. We define $dom(\Gamma_1 \circledast \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$ and

$$(\Gamma_1 \circledast \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \circledast \Gamma_2(x) & \text{if } x \in dom(\Gamma_1(x)) \cap dom(\Gamma_2(x)) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1) \end{cases}$$

This summation operation is defined only if $\Gamma_1(x) \circledast \Gamma_2(x)$ is defined for all $x \in dom(\Gamma_1(x)) \cap dom(\Gamma_2(x))$. In this case Γ_1 and Γ_2 are called compatible.

In some cases (see rule (T-OUT) in Table 11.2) we will use a slightly extended definition of a type environment, allowing that a name x appears more than once. This is only permissible if the different types of x are compatible. Such an extended type environment can be converted into an equivalent standard type environment using the following law: $\Gamma, x: t_1, x: t_2 = \Gamma, x: (t_1 \circledast t_2)$.

We also need the following join of a type environment with zero:

$$(x_1: [\tilde{t_1}]^{a_1}, \ldots, x_n: [\tilde{t_n}]^{a_n}) \vee 0 = x_1: [\tilde{t_1}]^{a_1 \vee 0}, \ldots, x_n: [\tilde{t_n}]^{a_n \vee 0}.$$

Finally let Γ, Γ' be two type environments with $dom(\Gamma) = dom(\Gamma')$. We write $\Gamma \leq \Gamma'$ if Γ and Γ' are compatible and for every $x \in dom(\Gamma)$ it holds that $mon(\Gamma(x)) \leq mon(\Gamma'(x))$.

The operations on type assignments satisfy the following laws:

Lemma 11.4.3

- *The operation $\circledast$ is associative and commutative.*
- *It holds that $(\Gamma_1 \circledast \Gamma_2) \vee 0 \leq (\Gamma_1 \vee 0) \circledast (\Gamma_2 \vee 0)$.*
- *It holds that $\Gamma_1 \leq \Gamma_1'$ and $\Gamma_2 \leq \Gamma_2'$ imply $\Gamma_1 \circledast \Gamma_2 \leq \Gamma_1' \circledast \Gamma_2'$ whenever either of the two sides of the last inequation is defined.*

Proof: Straightforward by the definition of the operations and Lemma 11.3.3. □

We are now ready to define the rules of the type system (see Table 11.2). We assume that there are two fixed monoid elements *out* and *in* (where *in* must be comparable[1] to 0) representing the capabilities of output and input prefixes respectively.

The intuitive meaning of the rules is as follows:

(T-$\leq$) We can always over-approximate the capabilities of a process.

[1]This is due to the fact that $sig(in)$ must be defined, since it will be used in typing replication of input prefixes.

$$\frac{\Gamma \vdash P,\ \Gamma \leq \Delta}{\Delta \vdash P} \ (\text{T-}\leq)$$

$$\Gamma \vee 0 \vdash \mathbf{0} \ (\text{T-Nil}) \qquad \frac{\Gamma_1 \vdash P_1,\ \Gamma_2 \vdash P_2}{\Gamma_1 \circledast \Gamma_2 \vdash P_1 \mid P_2} \ (\text{T-Par})$$

$$\frac{\Gamma, x\colon t \vdash P}{\Gamma \vdash (\nu x\colon t)P} \ (\text{T-Restr}) \quad \text{if } mon(\Gamma(x)) \geq 0 \text{ whenever } x \in dom(\Gamma)$$

$$\frac{\Gamma, x\colon [\tilde{t}]^a, \tilde{y}\colon \tilde{t} \vdash P}{\Gamma \vee 0, x\colon [\tilde{t}]^{(a-out)\vee 0+in} \vdash x(\tilde{y}).P} \ (\text{T-In})$$

$$\frac{\Gamma, x\colon [\tilde{t}]^a \vdash P}{\Gamma' \vee 0, x\colon [\tilde{t}]^{(a'-in)\vee 0+out} \vdash \overline{x}\langle \tilde{z}\rangle.P} \ (\text{T-Out}) \quad \text{if } \Gamma', x\colon [\tilde{t}]^{a'} = (\Gamma, x\colon [\tilde{t}]^a) \circledast \tilde{z}\colon \tilde{t}$$

$$\frac{\Gamma, x\colon [\tilde{t}]^a \vdash x(\tilde{y}).P}{\Gamma, x\colon [\tilde{t}]^{a+sig(in)} \vdash !x(\tilde{y}).P} \ (\text{T-Rep}) \quad \text{if } \Gamma \circledast \Gamma \leq \Gamma \text{ and } a + a \leq a$$

Table 11.2: Typing rules

(T-Nil) The nil process can have an arbitrary type assignment, provided the monoid elements of the free names are greater than 0. This stems from the fact that we only over-approximate but never under-approximate. Otherwise we could assign a negative monoid element to a name x in the type environment, which would decrease the capability of x whenever two type environments are added.

(T-Par) The parallel composition of two processes can be typed by adding their respective type assignments.

(T-Restr) If a name is restricted, we remove the assumption on it, but retain its type by integrating it into the process description. Note that Γ might still contain an assignment to x, in which case we have to make sure that its corresponding monoid element is greater than or equal to 0. This is consistent with the fact that assumptions on names that do not occur free in a process must have capabilities greater or equal to 0 due to over-approximation (see also rule (T-Nil) or Lemma 11.4.5).

(T-In) In this rule we make sure that the types of the names in the sequence $\tilde{y}$ match the types of the names that are sent via channel x. If this is the case we remove the assumptions on $\tilde{y}$ since these names are bound by the input prefix. Now since removing the input prefix $x(\tilde{y})$ would also mean the removal of a corresponding output prefix in the environment, we can anticipate this by subtracting *out* from the capability a of x. Afterwards we take the join with 0 and add *in* in order to record the input capability of x. The join with 0 is necessary since we do not count capabilities underneath a prefix. Taking into account negative capbilities located underneath a prefix would therefore be under-approximation, whereas positive capabilites lead to over-approximation, which is allowed and even necessary in order to show the subject reduction property. For similar reasons, we take the join of Γ with 0.

(T-OUT) This rule is slightly more complicated than the rule for the input prefix. First, a tuple of names $\tilde{z}$ that is sent via x can be used by a receiver in accordance with the types $\tilde{t}$. We anticipate this by adding $\tilde{z}\colon \tilde{t}$ to the type environment of P. Whenever one of the z_i already occurs in $\Gamma, x\colon [\tilde{t}]^a$, this makes sure that the respective types are compatible. Note that x might occur in the sequence $\tilde{z}$, which changes its monoid element from a to a' during the operation.

The monoid element a' is treated analogously to the way described for input prefixes by subtracting *in*, taking the join with 0 and adding *out*. For the rest of the monoid elements we are also required to take the join with 0 as explained above.

(T-REP) In this rule we have to make sure that a replicated process has a type assignment which is either idempotent or gets smaller when added to itself. This can be achieved if Γ contains only negative or idempotent monoid elements on the top level. Furthermore, since we know that infinitely many copies of the input prefix with subject x are available, we add $\bot$, $\top$ or 0, according to the value of *in*.

Example: In Table 11.3 we show how to derive a type assignment Γ for the process $P = !a(x,y).\overline{x}\langle y\rangle \mid \overline{a}\langle b,c\rangle$ introduced in Section 11.2.1. We use the ℓ-monoid $\mathbb{Z}^\infty \times \mathbb{Z}^\infty$ and set $in = (0,1)$, $out = (1,0)$.

Typing the replication operator involves the application of typing rule (T-$\leq$), replacing the monoid element $(0,1)$ of channel name a by $(0,\infty)$ in order to make it idempotent.

$$\dfrac{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(0,0)}, b\colon [[\,]^{(0,0)}]^{(0,0)}, c\colon [\,]^{(0,0)} \vdash \mathbf{0}}{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(1,0)^{a}}, b\colon [[\,]^{(0,0)}]^{(1,0)}, c\colon [\,]^{(0,0)} \vdash \overline{a}\langle b,c\rangle}$$

$$\dfrac{\dfrac{\dfrac{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(0,0)}, x\colon [[\,]^{(0,0)}]^{(0,0)}, y\colon [\,]^{(0,0)} \vdash \mathbf{0}}{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(0,0)}, x\colon [[\,]^{(0,0)}]^{(1,0)^{b}}, y\colon [\,]^{(0,0)} \vdash \overline{x}\langle y\rangle}}{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(0,1)^{c}} \vdash a(x,y).\overline{x}\langle y\rangle}}{a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(0,\infty)^{d}} \vdash !a(x,y).\overline{x}\langle y\rangle}$$

$$a\colon [[[\,]^{(0,0)}]^{(1,0)}, [\,]^{(0,0)}]^{(1,\infty)}, b\colon [[\,]^{(0,0)}]^{(1,0)}, c\colon [\,]^{(0,0)} \vdash P$$

[a] $(1,0) = ((0,0)-(0,1)) \vee (0,0) + (1,0) = (0 - in) \vee 0 + out$
[b] $(1,0) = ((0,0)-(0,1)) \vee (0,0) + (1,0) = (0 - in) \vee 0 + out$
[c] $(0,1) = ((0,0)-(1,0)) \vee (0,0) + (0,1) = (0 - out) \vee 0 + in$
[d] Smallest element a larger than $(0,1)$ which satisfies $a + a \leq a$ is $(0,\infty)$. Also $(0,\infty) + sig(in) = (0,\infty) + (0,\infty) = (0,\infty)$.

Table 11.3: Deriving a type assignment

In order to better motivate the rules in Table 11.2, we will sketch in advance the central case of the Subject Reduction Theorem (see Theorem 11.4.8) in a simplified form. Let us assume that $\Gamma \vdash P$ where $P = \overline{x}\langle z\rangle.Q \mid x(y).R$. Obviously P reduces to $P' = Q \mid R\{z/y\}$. Forgetting rule (T-$\leq$) for the moment this implies that

$$\Gamma = \Gamma'_Q \circledast \Gamma'_R \qquad \Gamma'_Q \vdash \overline{x}\langle z\rangle.Q \qquad \Gamma'_R \vdash x(y).R.$$

Now, by looking at the typing rules in Table 11.2 we can infer that

$$\Gamma'_Q = (\Gamma_Q \circledast z : t) \vee 0, x \colon [t]^{(a-in)\vee 0+out} \quad \text{and} \quad \Gamma_Q, x \colon [t]^a \vdash Q.$$

Furthermore it holds that

$$\Gamma'_R = \Gamma_R \vee 0, x \colon [t]^{(b-out)\vee 0+in} \quad \text{and} \quad \Gamma_R, x \colon [t]^b, y \colon t \vdash R.$$

We can argue that $\Gamma_R(z)$ is compatible with t if there should be an assignment to z in Γ_R. Therefore the Substitution Lemma (Lemma 11.4.7), which will be shown below, gives us $\Gamma_R, x \colon [t]^b \circledast z \colon t \vdash R\{z/y\}$. Everything combined we obtain

$$\Gamma_Q \circledast \Gamma_R \circledast z \colon t, x \colon [t]^{a+b} \vdash P'.$$

It is now left to show that this is smaller than Γ and (T-$\leq$) will give us the desired result. This can be shown with Lemma 11.4.3 and by observing that $(a - in) \vee 0 + out + (b - out) \vee 0 + in \geq a + b$, which can be shown by using some of the inequations for ℓ-monoids introduced in Lemma 11.3.3.

We can now define how an instantiation of this general framework, a specific type systems, looks like.

Definition 11.4.4 (Instance Type System) *Let I be a residuated lattice-ordered monoid and let in and out be two fixed monoid elements, where either $in \leq 0$ or $in \geq 0$, i.e., in is comparable to 0.*

Then we call the tuple $T = (I, in, out)$ an instance type system. We write $\Gamma \vdash_T P$ if the type assignment Γ can be derived for the π-calculus process P, using the rules given in Table 11.2 and the components of T. We will often omit the index T if it is obvious from the context.

In order to show the subject reduction property of the type system, i.e. the fact that the type is invariant under reductions, we must first show the following lemmas.

Lemma 11.4.5 (Weakening and Strengthening) *If $\Gamma \vdash P$ with $x \notin fn(P)$, a is a monoid element with $a \geq 0$ and $\tilde{t}$ is a sequence of type trees, then it holds that $\Gamma, x \colon [\tilde{t}]^a \vdash P$. Furthermore it holds that $\Gamma\backslash x \vdash P$ and $mon(\Gamma(x)) \geq 0$ whenever $x \in dom(\Gamma)$.*

Proof: Proof by induction on the type derivation. □

Lemma 11.4.6 *Let P, Q be two processes that are equivalent with respect to α-conversion. Then it holds that $\Gamma \vdash P \iff \Gamma \vdash Q$.*

Proof: By induction on the typing rules. Whenever a name x is replaced by y this amounts to replacing the occurrences of x by y in the type derivation. Note that names that do not occur free in P and Q can be removed from the type environment according to Lemma 11.4.5 and do not get in the way. $\square$

Every instance type system satisfies the following substitution lemma, which is central for proving the subject reduction property:

Lemma 11.4.7 (Substitution) *Let $v \neq w$ be two names.*

If $\Sigma, w\colon [\tilde{s}]^c \vdash P$ and $w \notin dom(\Sigma)$, then $\Sigma' = \Sigma \circledast v\colon [\tilde{s}]^c \vdash P\{v/w\}$, whenever Σ' is defined.

Proof: We show this lemma by induction on the typing rules:

(T-$\leq$) In this case $\Gamma, w\colon [\tilde{s}]^{c'} \leq \Sigma, w\colon [\tilde{s}]^c$ and $\Gamma, w\colon [\tilde{s}]^{c'} \vdash P$ with $c' \leq c$. It holds that Σ' is defined if and only if $\Gamma \circledast v\colon [\tilde{s}]^{c'}$ is defined. The induction hypothesis implies that $\Gamma \circledast v\colon [\tilde{s}]^{c'} \vdash P\{v/w\}$, and clearly $\Gamma \circledast v\colon [\tilde{s}]^{c'} \leq \Sigma'$, which implies $\Sigma' \vdash P\{v/w\}$.

(T-Nil) In this case $P = \mathbf{0}$ and $\Sigma, w\colon [\tilde{s}]^c = \Gamma \vee 0$. This implies $c \geq 0$. Now $\Sigma'(v) = [\tilde{s}]^{c'+c}$ where $c' \geq 0$ (we set $c' = 0$ if $v \notin dom(\Sigma)$) and therefore $c + c' \geq 0$. This holds also for all other topmost monoid elements in Σ'. Therefore $\Sigma' \vdash \mathbf{0} = P\{v/w\}$.

(T-Par) In this case $P = P_1 \mid P_2$ with $\Gamma_i \vdash P_i$, $i \in \{1,2\}$ and $\Sigma, w\colon [\tilde{s}]^c = \Gamma_1 \circledast \Gamma_2$. It holds that $c = c_1 + c_2$ with $\Gamma_i = \Delta_i, w\colon [\tilde{s}]^{c_i}$ or $c_i = 0$ and $w \notin dom(\Gamma_i)$. In the second case we set $\Delta_i = \Gamma_i$.

In the first case, we can infer with the induction hypothesis that $\Delta_i \circledast v\colon [\tilde{s}]^{c_i} \vdash P_i\{x/y\}$. In the second case Lemma 11.4.5 gives us $\Delta_i, w\colon [\tilde{s}]^{c_i} \vdash P_i$, from which we can again infer with the induction hypothesis that $\Delta_i \circledast v\colon [\tilde{s}]^{c_i} \vdash P_i\{x/y\}$. Combined this yields:

$$\begin{aligned}\Sigma' &= (\Delta_1 \circledast \Delta_2) \circledast v\colon [\tilde{s}]^c \\ &= (\Delta_1 \circledast v\colon [\tilde{s}]^{c_1}) \circledast (\Delta_2 \circledast v\colon [\tilde{s}]^{c_2}) \vdash \underbrace{P_1\{v/w\} \mid P_2\{v/w\}}_{=P\{v/w\}}.\end{aligned}$$

(T-Restr) In this case $P = (\nu x\colon t)P'$. Since x is bound we can assume that $x \neq v$ and $x \neq w$, due to Lemma 11.4.6. It follows that $\Sigma, w\colon [\tilde{s}]^c, x\colon t \vdash P'$ where $mon(\Sigma(x)) \geq 0$ if $x \in dom(\Sigma)$. The induction hypothesis implies that $\Sigma \circledast v\colon [\tilde{s}]^c, x\colon t \vdash P'\{v/w\}$. Furthermore it holds that $mon((\Sigma \circledast v\colon [\tilde{s}]^c)(x)) = mon(\Sigma(x)) \geq 0$ whenever x occurs in that type environment. So we can infer that $\Sigma' = \Sigma \circledast v\colon [\tilde{s}]^c \vdash (\nu x\colon t)(P'\{v/w\}) = P\{v/w\}$.

(T-In) In this case it holds that $P = x(\tilde{y}).P'$. Since all the y_i are bound, we can assume because of Lemma 11.4.6 that $y_i \neq w$ and $y_i \neq v$ for each i. It holds that

$$\begin{aligned}\Gamma, x\colon [\tilde{t}]^a, \tilde{y}\colon \tilde{t} &\vdash P \\ \Sigma, w\colon [\tilde{s}]^c &= \Gamma \vee 0, x\colon [\tilde{t}]^{(a-out)\vee 0+in}.\end{aligned}$$

Let σ be the mapping on names that maps w to v and is the identity on all other names. Furthermore we set $\Delta = \Gamma, x\colon [\tilde{t}]^a, \tilde{y}\colon \tilde{t}$.

Obviously an assignment of the form $w\colon [\tilde{s}]^{\hat{c}}$ occurs in Δ. The induction hypothesis gives us $\Delta\backslash w \circledast v\colon [\tilde{s}]^{\hat{c}} \ \vdash \ P'\{v/w\}$. This type environment contains an assignment to $\sigma(x)$ in any case (since either ($\sigma(x) = x$ and $w \neq x$) or $\sigma(x) = v$). The type environment is therefore of the form

$$\hat{\Gamma}, \sigma(x)\colon [\tilde{t}]^{\hat{a}}, \tilde{y}\colon \tilde{t} = \Delta\backslash w \circledast v\colon [\tilde{s}]^{\hat{c}}$$

We can also set

$$\hat{\Sigma} = \hat{\Gamma} \vee 0, \sigma(x)\colon [\tilde{t}]^{(\hat{a}-out)\vee 0+in}.$$

Because of rule (T-In) it holds that $\hat{\Sigma} \vdash \sigma(x)(\tilde{y}).(P'\{v/w\}) = P\{v/w\}$.

It remains to show that $\hat{\Sigma} \leq \Sigma'$. First $\hat{\Sigma}$ and Σ' have the same domain and the corresponding type trees in the type environment coincide in everything but the topmost monoid elements. Let us now consider the topmost monoid elements.

First, nothing changes for all type trees different from the type tree corresponding to v. We now consider the following cases in order to show that $mon(\hat{\Sigma}(v)) \leq mon(\Sigma'(v))$. Furthermore we set $b = mon(\Delta(v))$ whenever $v \in dom(\Delta)$. Otherwise we set $b = 0$.

$w \neq x \ \wedge \ v \neq x$: In this case it holds that

$$\begin{array}{rcl} mon(\hat{\Sigma}(v)) & = & (b + \hat{c}) \vee 0 \\ & \leq & (\hat{c} \vee 0) + (b \vee 0) \\ & = & c + (b \vee 0) \\ & = & mon(\Sigma'(v)). \end{array}$$

$w = x$: In this case $\sigma(x) = v$ and we obtain

$$\begin{array}{rcl} mon(\hat{\Sigma}(v)) & = & ((b + \hat{c}) - out) \vee 0 + in \\ & \leq & ((\hat{c} - out) + b) \vee 0 + in \\ & \leq & (\hat{c} - out) \vee 0 + in + (b \vee 0) \\ & = & c + (b \vee 0) \\ & = & mon(\Sigma'(v)). \end{array}$$

$v = x$: Analogous to the case $w = x$ above.

(T-Out) In this case it holds that $P = x\langle\tilde{z}\rangle.P'$ and

$$\begin{array}{rcl} \Gamma, x\colon [\tilde{t}]^a & \vdash & P' \\ (\Gamma, x\colon [\tilde{t}]^a) \circledast \tilde{z}\colon \tilde{t} & = & \Gamma', x\colon [\tilde{t}]^{a'} \\ \Sigma, w\colon [\tilde{s}]^c & = & \Gamma' \vee 0, x\colon [\tilde{t}]^{(a'-in)\vee 0+out}. \end{array}$$

Let σ be the mapping on names that maps w to v and is the identity on all other names. Furthermore we set $\Delta = \Gamma, x\colon [\tilde{t}]^a$.

Whenever an assignment of the form $w\colon [\tilde{s}]^{\hat{c}}$ occurs in Δ we set $\hat{\Delta} = \Delta$, otherwise we set $\hat{c} = 0$ and define $\hat{\Delta} = \Delta, w\colon [\tilde{s}]^{\hat{c}}$. In the latter case we can conclude with Lemma 11.4.5 that $\hat{\Delta} \vdash P'$. The induction hypothesis gives us $\Delta\backslash w \circledast v\colon [\tilde{s}]^{\hat{c}} \ \vdash \ P'\{v/w\}$ in both cases. This type environment contains an assignment to $\sigma(x)$ in any case.

Whenever $w = x$, then we know that $\tilde{t} = \tilde{s}$. If $w = z_i$, then we can be sure that the types of v and z_i are compatible, since the types of v and w are compatible. So we can set:

$$\hat{\Gamma}, \sigma(x)\colon [\tilde{t}]^{\hat{a}} = (\Delta\backslash w \circledast v\colon [\tilde{s}]^{\hat{c}}) \circledast \sigma(\tilde{z})\colon \tilde{t}$$

We can also set

$$\hat{\Sigma} = \hat{\Gamma} \vee 0, \sigma(x)\colon [\tilde{t}]^{(\hat{a}-in)\vee 0+out}.$$

Because of rule (T-OUT) it holds that $\hat{\Sigma} \ \vdash \ \sigma(\overline{x})\langle\sigma(\tilde{z})\rangle.(P'\{v/w\}) \ = \ P\{v/w\}$.

It remains to show that $\hat{\Sigma} \leq \Sigma'$. First $\hat{\Sigma}$ and Σ' have the same domain and the corresponding type trees in the type environment coincide in everything but the topmost monoid element. Let us now consider the topmost monoid elements.

First, nothing changes for all type trees different from the type tree corresponding to v. We now consider the following cases in order to show that $mon(\hat{\Sigma}(v)) \leq mon(\Sigma'(v))$. We assume that each t_i in the sequence $\tilde{t}$ is of the form $[\tilde{u_i}]^{b_i}$. Furthermore we set $b = mon(\Delta(v))$ whenever $v \in dom(\Delta)$. Otherwise we set $b = 0$.

$w \neq x \ \wedge \ v \neq x$**:** In this case it holds that

$$\begin{aligned} mon(\hat{\Sigma}(v)) &= \left(b + \hat{c} + \sum_{\sigma(z_i)=v} b_i\right) \vee 0 \\ &\leq \left(\hat{c} + \sum_{z_i=w} b_i\right) \vee 0 + (b \vee 0) \\ &= c + (b \vee 0) \\ &= mon(\Sigma'(v)). \end{aligned}$$

$w = x$: In this case we obtain

$$\begin{aligned}
mon(\hat{\Sigma}(v)) &= \left(\left(b + \hat{c} + \sum_{\sigma(z_i)=v} b_i\right) - in\right) \vee 0 + out \\
&= \left(\left(b + \hat{c} + \sum_{z_i=w} b_i\right) - in\right) \vee 0 + out \\
&\leq \left(\left(\left(\hat{c} + \sum_{z_i=w} b_i\right) - in\right) + b\right) \vee 0 + out \\
&\leq \left(\left(\hat{c} + \sum_{z_i=w} b_i\right) - in\right) \vee 0 + out + (b \vee 0) \\
&= c + (b \vee 0) \\
&= mon(\Sigma'(v)).
\end{aligned}$$

$v = x$: Analogous to the case $w = x$ above.

(T-Rep) In this case $P = !x(\tilde{y}).P'$. Furthermore

$$\begin{aligned}
\Gamma, x\colon [\tilde{t}]^a &\vdash x(\tilde{y}).P' \\
\Sigma, w\colon [\tilde{s}]^c &= \Gamma, x\colon [\tilde{t}]^{a+sig(in)}
\end{aligned}$$

and $\Gamma \circledast \Gamma \leq \Gamma$ and $a + a \leq a$. Let σ be the mapping on names that maps w to v and is the identity on all other names. Again we can assume that $y_i \neq v$ and $y_i \neq w$ for each i. We distinguish the following cases:

$w \neq x \ \wedge \ v \neq x$: In this case it holds that $\Gamma \backslash w, w\colon [\tilde{s}]^c, x\colon [\tilde{t}]^a \vdash x(\tilde{y}).P'$ and from $\Gamma \circledast \Gamma \leq \Gamma$ it follows that $c + c \leq c$. From the induction hypothesis we can infer that

$$\underbrace{(\Gamma \backslash w, x\colon [\tilde{t}]^a) \circledast v\colon [\tilde{s}]^c}_{=(\Gamma \backslash w \circledast v\colon [\tilde{s}]^c), x\colon [\tilde{t}]^a} \vdash \sigma(x)(\tilde{y}).(P'\{v/w\})$$

Because of $c + c \leq c$ and $\Gamma \circledast \Gamma \leq \Gamma$ it holds that $(mon(\Gamma(v)) + c) + (mon(\Gamma(v)) + c) \leq mon(\Gamma(v)) + c$ whenever $v \in dom(\Gamma)$ and therefore $(\Gamma \backslash w \circledast v\colon [\tilde{s}]^c) \circledast (\Gamma \backslash w \circledast v\colon [\tilde{s}]^c) \leq \Gamma \backslash w \circledast v\colon [\tilde{s}]^c$.
From the typing rule (T-Rep) we can infer that

$$\Sigma' = (\Gamma \backslash w \circledast v\colon [\tilde{s}]^c), x\colon [\tilde{t}]^{a+sig(in)} \vdash P\{v/w\}.$$

$w = x$: In this case it holds that $a + sig(in) = c$, $\tilde{s} = \tilde{t}$ and $\Sigma = \Gamma \backslash w$. From the induction hypothesis we can infer that

$$\Sigma \circledast v\colon [\tilde{t}]^a \vdash v(\tilde{y}).(P'\{v/w\}).$$

As above we can show that $(\Sigma \circledast v\colon [\tilde{t}]^a) \circledast (\Sigma \circledast v\colon [\tilde{t}]^a) \leq \Sigma \circledast v\colon [\tilde{t}]^a$ and therefore rule (T-Rep) gives us

$$\Sigma' = \Sigma \circledast v\colon [\tilde{t}]^{a+sig(in)} \vdash P\{v/w\}.$$

$v = x$**:** In this case it holds that $\Gamma\backslash w, w\colon [\tilde{s}]^c, v\colon [\tilde{t}]^a \ \vdash \ v(\tilde{y}).P'$ and furthermore $\tilde{t} = \tilde{s}$. From the induction hypothesis we can infer that

$$\Gamma\backslash w, v\colon [\tilde{t}]^{a+c} \ \vdash \ v(\tilde{y}).(P'\{v/w\}).$$

Again we can infer that $(\Gamma\backslash w, v\colon [\tilde{t}]^{a+c}) \circledast (\Gamma\backslash w, v\colon [\tilde{t}]^{a+c}) \leq \Gamma\backslash w, v\colon [\tilde{t}]^{a+c}$ and therefore it follows with rule (T-Rep) that

$$\Sigma' = \Gamma\backslash w, v\colon [\tilde{t}]^{a+c+sig(in)} \ \vdash \ P\{v/w\}.$$

□

Now we can state the subject reduction property in the following way.

Theorem 11.4.8 (Subject Reduction) *If $P \equiv Q$, then $\Sigma \ \vdash \ P \iff \Sigma \ \vdash \ Q$. If $P \to P'$ and $\Sigma \ \vdash \ P$ then $\Sigma \ \vdash \ P'$.*

Proof: We show the first half of the proposition by induction on the rules of structural congruence, taking into account that $\equiv$ is a congruence. The cases for reflexivity, transitivity, symmetry and contextualization are straightforward.

(C-Com) Follows immediately from the commutativity of $\circledast$.

(C-$\mathbf{0}$) Follows immediately from the fact that $\mathbf{0}$ can be typed with the empty type environment $\emptyset$, which is the unit of $\circledast$.

(C-Ass) Follows immediately from the associativity of $\circledast$.

(C-Restr1) Let $(\nu x\colon t)\mathbf{0} \equiv \mathbf{0}$ and we assume first that $\Sigma \vdash (\nu x)\mathbf{0}$. From the typing rules we can infer that $\Sigma \geq \Gamma$, $\Gamma \vdash (\nu x\colon t)\mathbf{0}$ and $\Gamma, x\colon t \vdash \mathbf{0}$. Since x does not occur free in $\mathbf{0}$, Lemma 11.4.5 implies $\Gamma \vdash \mathbf{0}$. Finally rule (T-$\leq$) gives us $\Sigma \vdash \mathbf{0}$.

In the other direction we have $\Sigma \vdash \mathbf{0}$ and rule (C-Restr1) implies that $mon(t) \geq 0$. Therefore Lemma 11.4.5 implies that $\Sigma, x\colon t \vdash \mathbf{0}$, from which we can infer $\Sigma \vdash (\nu x\colon t)\mathbf{0}$ with rule (T-Restr).

(C-Restr2) Immediate from the fact that $\Gamma, x\colon s, y\colon t = \Gamma, y\colon t, x\colon s$.

(C-Restr3) Let $((\nu x\colon t)P_1) \mid P_2 \equiv (\nu x\colon t)(P_1 \mid P_2)$ and $x \notin fn(P_2)$.

Let us first assume that $\Sigma \vdash ((\nu x\colon t)P_1) \mid P_2$. This implies that $\Sigma \geq \Gamma_1 \circledast \Gamma_2$ where $\Gamma_1 \vdash (\nu x\colon t)P_1$, $\Gamma_2 \vdash P_2$. Since $x \notin fn(((\nu x\colon t)P_1) \mid P_2)$ it follows from Lemma 11.4.5 that $mon(\Sigma(x)) \geq 0$ whenever $x \in dom(\Sigma)$. Furthermore $\Gamma_1 \geq \Gamma_1'$ where $\Gamma_1' \vdash (\nu x\colon t)P_1$ and $\Gamma_1', x\colon t \vdash P_1$. From Lemma 11.4.5 it follows that $\Gamma_2\backslash x \vdash P_2$ and typing rules (T-Par) then implies that

$$(\Gamma_1', x\colon t) \circledast \Gamma_2\backslash x \ \vdash \ P_1 \mid P_2.$$

Summation of type environments is defined since $\Gamma_1 \circledast \Gamma_2$ is defined. With rule (T-Restr) it follows that $\Gamma_1'\backslash x \circledast \Gamma_2\backslash x \ \vdash \ (\nu x\colon t)(P_1 \mid P_2)$. It holds that $\Sigma\backslash x \geq \Gamma_1'\backslash x \circledast \Gamma_2'\backslash x$. By adding a possible assumption on x with Lemma 11.4.5 we obtain $\Sigma \vdash (\nu x\colon t)(P_1 \mid P_2)$.

In the other direction we can infer that $\Sigma \geq \Gamma$, $\Gamma \vdash (\nu x\colon t)(P_1 \mid P_2)$, $\Gamma, x\colon t \vdash P_1 \mid P_2$ and finally $\Gamma, x\colon t \geq \Gamma_1 \circledast \Gamma_2$ with $\Gamma_i \vdash P_i$, $i \in \{1,2\}$. We assume that $t = [\tilde{s}]^a$. The type environment Γ_i has either the form $\Gamma'_i, x\colon [\tilde{s}]^{a_i}$ or Γ'_i where $x \notin dom(\Gamma'_i)$. If the latter is the case we set $a_i = 0$ and it follows that $\Gamma'_i, x\colon [\tilde{s}]^{a_i} \vdash P_i$.

From Lemma 11.4.5 we can infer that $a_2 \geq 0$ since x does not occur free in P_2. Because of $a_1 + a_2 \leq a$ it follows that $a_1 \leq a$. With rule (T-$\leq$) we can infer that $\Gamma'_1, x\colon [\tilde{s}]^a \vdash P_1$. Then rule (T-RESTR) implies $\Gamma'_1 \vdash (\nu x\colon t)P_1$. Rule (T-PAR) implies that $\Gamma'_1 \circledast \Gamma'_2 \vdash ((\nu x\colon t)P_1) \mid P_2$. It holds that $\Sigma \backslash x \geq \Gamma'_1 \circledast \Gamma'_2$. By adding a possible assumption on x with Lemma 11.4.5 we obtain $\Sigma \vdash ((\nu x\colon t)P_1) \mid P_2$.

The second part of the proposition is shown by induction on the reduction rules. We only show the following two cases. The rest of the cases is obvious with the induction hypothesis and the first part of this theorem.

(R-COMM) $P = \overline{x}\langle\tilde{z}\rangle.Q \mid x(\tilde{y}).R$, $P' = Q \mid R\{\tilde{z}/\tilde{y}\}$ and $\Sigma \vdash P$. We can assume that x does not occur in $\tilde{y}$.

This implies that $\Sigma \geq \Sigma_Q \circledast \Sigma_R$ where $\Sigma_Q \vdash \overline{x}\langle\tilde{z}\rangle.Q$ and $\Sigma_R \vdash x(\tilde{y}).R$. Furthermore $\Sigma_Q \geq \Upsilon_Q$ where

$$\begin{array}{rcl} \Upsilon_Q & = & \Gamma'_Q \vee 0, x\colon [\tilde{t}]^{(a'-in)\vee 0+out} \\ \Gamma'_Q, x\colon [\tilde{t}]^{a'} & = & (\Gamma_Q, x\colon [\tilde{t}]^a) \circledast \tilde{z}\colon \tilde{t} \\ \Gamma_Q, x\colon [\tilde{t}]^a & \vdash & Q. \end{array}$$

Additionally $\Sigma_R \geq \Upsilon_R$ where

$$\begin{array}{rcl} \Upsilon_R & = & \Gamma_R \vee 0, x\colon [\tilde{t}]^{(b-out)\vee 0+in} \\ \Gamma_R, x\colon [\tilde{t}]^b, \tilde{y}\colon \tilde{t} & \vdash & R. \end{array}$$

Since $\circledast$ respects $\leq$ it holds that $\Sigma \geq \Upsilon_Q \circledast \Upsilon_R$. From Lemma 11.4.7 we can infer that

$$(\Gamma_R, x\colon [\tilde{t}]^b) \circledast \tilde{z}\colon \tilde{t} \ \vdash \ R\{\tilde{z}/\tilde{y}\},$$

which is well-defined, since $(\Gamma_Q, x\colon [\tilde{t}]^a) \circledast \tilde{z}\colon \tilde{t}$ is well-defined and therefore every name z_i must have a type compatible with t_i. From (T-PAR) we can infer that

$$(\Gamma_Q, x\colon [\tilde{t}]^a) \circledast ((\Gamma_R, x\colon [\tilde{t}]^b) \circledast \tilde{z}\colon \tilde{t}) \ \vdash \ Q \mid R\{\tilde{z}/\tilde{y}\}.$$

The type environment on the left-hand side is equal to

$$\begin{array}{rl} & ((\Gamma_Q, x\colon [\tilde{t}]^a) \circledast \tilde{z}\colon \tilde{t}) \circledast (\Gamma_R, x\colon [\tilde{t}]^b) \\ = & (\Gamma'_Q, x\colon [\tilde{t}]^{a'}) \circledast (\Gamma_R, x\colon [\tilde{t}]^b) \\ = & \Gamma'_Q \circledast \Gamma_R, x\colon [\tilde{t}]^{a'+b} \\ \leq & (\Gamma'_Q \vee 0) \circledast (\Gamma_R \vee 0), x\colon [\tilde{t}]^{(a'-in)\vee 0+out+(b-out)\vee 0+in} \\ = & \Upsilon_Q \circledast \Upsilon_R. \end{array}$$

The inequation holds because of Lemma 11.4.3 and $(a' - in) \vee 0 + out + (b - out) \vee 0 + in \geq (a' - in) + in + (b - out) + out \geq a' + b$ for two elements a', b of an ℓ-monoid (see Lemma 11.3.3). Then $\Sigma \geq \Upsilon_Q \circledast \Upsilon_R$ implies $\Sigma \vdash P'$.

(R-Rep) $P = \overline{x}\langle\tilde{z}\rangle.Q \mid !x(\tilde{y}).R$, $P' = Q \mid R\{\tilde{z}/\tilde{y}\} \mid !x(\tilde{y}).R$ and $\Sigma \vdash P$. We show that $\Upsilon \vdash !x(\tilde{y}).R$ implies $\Upsilon \vdash x(\tilde{y}).R \mid !x(\tilde{y}).R$, the rest follows from case (R-Comm).

From $\Upsilon \vdash !x(\tilde{y}).R$ it follows that $\Upsilon \geq \Gamma, x\colon [\tilde{t}]^{a+sig(in)}$ where $\Gamma, x\colon [\tilde{t}]^{a+sig(in)} \vdash !x(\tilde{y}).R$ and $\Gamma, x\colon [\tilde{t}]^{a} \vdash x(\tilde{y}).R$, $\Gamma \circledast \Gamma \leq \Gamma$ and $a + a \leq a$.

From (T-Par) we can infer that

$$(\Gamma, x\colon [\tilde{t}]^{a+sig(in)}) \circledast (\Gamma, x\colon [\tilde{t}]^{a}) \vdash x(\tilde{y}).R \mid !x(\tilde{y}).R.$$

This type environment is equal to

$$\Gamma \circledast \Gamma, x\colon [\tilde{t}]^{a+sig(in)+a} \leq \Gamma, x\colon [\tilde{t}]^{a+sig(in)} \leq \Upsilon.$$

Finally this implies $\Upsilon \vdash x(\tilde{y}).R \mid !x(\tilde{y}).R$.

□

11.5 Using the Type System for Process Analysis

As in other type systems for mobile processes, a type guarantees absence of runtime errors which may appear in the form of arity mismatches in the communication rules (R-Comm) and (R-Rep), but it also enables us to perform more detailed process analysis.

11.5.1 Process Capabilities

The aim of this paper is to construct type systems yielding useful results for the analysis and verification of parallel processes. In our case the generic type system gives information about the properties of a process, especially concerning its input and output capabilities. We will now formally define the connection between the type of a process and its capabilities.

Definition 11.5.1 *Let P be a process and let w be a free name occurring in P. We define P's* capability *wrt. w, denoted by $C_w(P)$ by adding the following monoid elements: for every use of w as an output port we add out and for every use of w as an input port we add in. Notice that we do not continue summation after prefixes (see Table 11.4).*

We can now show that the type system is sound, in the sense that it gives appropriate upper bounds for the capabilities of processes.

Theorem 11.5.2 (Type Safety) *If $\Gamma \vdash P$, $P \to^* P'$ and w is a free name of P it follows that $C_w(P') \leq mon(\Gamma(w))$. If P' contains a subexpression $(\nu x\colon t)Q$ which is not located underneath a prefix it follows that $C_x(Q) \leq mon(t)$.*

$C_w(\mathbf{0}) = 0 \quad C_w(P \mid Q) = C_w(P) + C_w(Q)$	
$C_w(\overline{x}\langle\tilde{z}\rangle.P) = \begin{cases} out & \text{if } x = w \\ 0 & \text{otherwise} \end{cases}$	$C_w(x(\tilde{y}).P) = \begin{cases} in & \text{if } x = w \\ 0 & \text{otherwise} \end{cases}$
$C_w(!x(\tilde{y}).P) = \begin{cases} sig(in) & \text{if } x = w \\ 0 & \text{otherwise} \end{cases}$	$C_w((\nu x\colon s)P) = \begin{cases} C_w(P) & \text{if } x \neq w \\ 0 & \text{otherwise} \end{cases}$

Table 11.4: Determining the capabilities of a process

Proof: We show the proposition in two steps:

- First we show that for every process P, $\Gamma \vdash P$ implies $C_w(P) \leq mon(\Gamma(w))$.

  This can easily be done by induction on P, the only problematic case being replication: If $\Sigma = \Gamma, x\colon [\tilde{t}]^a \ \vdash !x(\tilde{y}).Q$, it follows from (T-Rep) and (T-$\leq$) that $\Gamma', x\colon [\tilde{t}]^b \ \vdash \ x(\tilde{y}).Q$, where $\Gamma \geq \Gamma'$ and $a \geq b + sig(in)$. Since $\Gamma', x\colon [\tilde{t}]^b$ was produced by typing rule (T-In) (and maybe rule (T-$\leq$)) it follows that $b \geq (a' - out) \vee 0 + in$ for some monoid element a'. This implies that $a \geq (a' - out) \vee 0 + in + sig(in) \geq in + sig(in) = sig(in)$.

  Therefore $C_x(P) = sig(in) \leq a = mon(\Sigma(x))$. For all other names $w \neq x$ it is straightforward to show that $C_w(P) = 0 \leq mon(\Sigma(w))$.

- Furthermore we can show by induction on the number of steps and with Theorem 11.4.8 that whenever $P \to^* P'$ and $\Gamma \vdash P$, then also $\Gamma \vdash P'$. Therefore $C_w(P') \leq mon(\Gamma(w))$.

The second part of the theorem can also be shown with Theorem 11.4.8. Note that for this part of the proof the side condition on rule (C-Restr1) in Table 11.1 is essential. □

The properties we can derive are of the form: "it is *always* the case that $C_w(P) \leq a$". In many cases more complex properties of processes can be derived from inequations of this form.

Definition 11.5.3 (Instance Type System for Property Y) *Let Y be a predicate on π-calculus processes. We call $T = (I, in, out, X)$ an instance type system for the predicate Y whenever $T' = (I, in, out)$ is an instance type system according to Definition 11.4.4 and X is a predicate on type assignments such that*

$$\Gamma \vdash_{T'} P \text{ and } X(\Gamma) \text{ imply } Y(P).$$

In Section 11.6 we give some examples for typical predicates X and Y.

11.5.2 Composition of Type Systems

Given two type systems checking certain capabilities of processes, it is not difficult to construct a type system computing upper bounds for tuples of capabilities. Let $T_i = (I_i, in_i, out_i, X_i), i \in \{1, 2\}$ be two instance type systems for predicates Y_1 and Y_2 respectively.

Then $T = (I_1 \times I_2, (in_1, in_2), (out_1, out_2), X)$ is an instance type system for the conjunction of Y_1 and Y_2 if $X(i_1, i_2) = X_1(i_1) \wedge X_2(i_2)$ and it is an instance type system for the disjunction of Y_1 and Y_2 if $X(i_1, i_2) = X_1(i_1) \vee X_2(i_2)$. (All monoid operations on $I_1 \times I_2$ are conducted pointwise.)

11.6 Examples

We now get back to the two example processes

$$P = T \mid H \mid (\nu d\colon t_d)(F \mid S) \qquad P' = T \mid H \mid (\nu d\colon t_d)(!F \mid !S)$$

introduced in Section 11.3.1 and type them with several instantiations of our type system, and thereby show how to mechanise process analysis in these cases.

First, in both cases, we obtain the same type environment from the point of view of structure. However the monoid elements differ with the various instantiations. The structure of type environment looks as follows:

$$\begin{aligned} & \Gamma = c\colon t_c,\ h\colon t_1,\ i_1 : t_1,\ i_2 : t_2 \\ \text{where} \quad & t_c = [t_1]^{m_c},\ t_1 = [t_1, t_2]^{m_1},\ t_2 = []^{m_2} \\ \text{and furthermore} \quad & t_d = [t_1]^{m_d}. \end{aligned}$$

Note that t_1 stands for an infinite, but regular, tree and $t_1 = [t_1, t_2]^{m_1}$ is its defining equation.

In the sequel we present several analyses of P and P' where the ℓ-monoids used appear in Table 11.5 and the results of the analysis, i.e. the monoid elements substituted for m_d, m_c, m_1, m_2, are shown in Table 11.6.

	property to be checked	underl. set	operation	order	*out*	*in*
1	input/output behaviour of P and P'	$\{none, output, input, both\}$	$\vee$	$\leq$	*output*	*input*
2	upper bounds on prefixes in P	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\leq$	$(1,0)$	$(0,1)$
3	upper bounds on prefixes in P'	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\leq$	$(1,0)$	$(0,1)$
4	lower bounds on prefixes in P	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\geq$	$(1,0)$	$(0,1)$
5	lower bounds on prefixes in P'	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\geq$	$(1,0)$	$(0,1)$
6	avoiding blocked output prefixes in P	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\sqsubseteq$[a]	$(1,0)$	$(0,1)$
7	avoiding blocked output prefixes in P'	$\mathbb{Z}^\infty \times \mathbb{Z}^\infty$	$+$	$\sqsubseteq$[a]	$(1,0)$	$(0,1)$

[a] $(i,j) \sqsubseteq (i',j')$ iff $i \leq i'$ and $j \geq j'$.

Table 11.5: The ℓ-monoids for different instantiations of the generic type system

11.6.1 Input/Output Behaviour of Channels

One simple application of our type system is to check whether channels are used for input or for output or for both. We use the monoid *IO* (with elements *none*, *output*–"output only", *input*–"input only" and *both*) introduced in Section 11.3.2 (see Table 11.5, row 1). We set $in = input$, $out = output$.

	property to be checked	m_d	m_c	m_1	m_2
1	input/output behaviour of P and P'	*both*	*both*	*both*	*none*
2	upper bounds on prefixes in P	$(1,1)$	$(1,1)$	$(1,1)$	$(0,0)$
3	upper bounds on prefixes in P'	(∞,∞)	$(1,\infty)$	$(1,\infty)$	$(0,0)$
4	lower bounds on prefixes in P	$(-1,0)$	$(-1,0)$	$(-1,-1)$	$(0,0)$
5	lower bounds on prefixes in P'	$(-\infty,\infty)$	$(-\infty,\infty)$	$(-\infty,-1)$	$(0,0)$
6	avoiding blocked output prefixes in P	$(1,0)$	$(1,0)$	$(1,-1)$	$(0,0)$
7	avoiding blocked output prefixes in P'	(∞,∞)	$(1,\infty)$	$(1,-1)$	$(0,0)$

Table 11.6: Resulting monoid elements for different instantiations of the generic type system

For both processes P and P' we obtain the same type assignments where the upper bounds are shown in Table 11.6 (row 1), i.e., the name i_2 is used neither for input nor output while all other names may be used for both. Note that, because of residuation, typing F on its own would yield capability *input* for name c, but no output capability. This is due to the fact that F alone will never reduce to a process with an output prefix with subject c.

This type system has some similarities to the one in [PS96], our type system however lacks a concept of co- and contravariance which is present in [PS96].

11.6.2 Upper Bounds on the Number of Prefixes

We attempt to define a type system, similar to the one presented in [KPT99] for our framework, i.e., we want to check how many processes try to input or output concurrently on the same channel.

We use the ℓ-monoid $\mathbb{Z}^\infty \times \mathbb{Z}^\infty$ (cartesian product of the set of integers with ∞ and $-\infty$) introduced in Section 11.3.2 (see Table 11.5, rows 2 & 3). The first component represents the number of active output prefixes (with a fixed subject) and the second component represents the number of active input prefixes.

We set $\mathit{out} = (1,0)$, $\mathit{in} = (0,1)$, and typing the processes P and P' yields the results given in Table 11.6 (rows 2 & 3). Since for P the upper bound is always $(1,1)$ or smaller we can conclude that there is at most one active input port and one active output port for any given subject at a time. For P' we can for example guarantee that $\overline{c}$ always occurs at most once as an output prefix, although it occurs underneath a replicated input prefix.

11.6.3 Confluence

As in [KPT99] we can use upper bounds on the number of active prefixes to guarantee confluence for π-calculus processes (see also [NS97]). Let Q be a process, and for every name x in Q which is either free or bound by the scope operator ν it holds that its capabilities never exceed $(1,1)$. Then we can guarantee that every channel (also bound channels) occurs at most once at any given time as the subject of an input or output prefix, and we thus have non-overlapping redexes in (R-Comm). Thus we can conclude that if $Q \rightarrow^* Q'$,

$Q' \to Q_1$ and $Q' \to Q_2$, then either $Q_1 \equiv Q_2$ or there is a process Q_3 such that $Q_1 \to Q_3$ and $Q_2 \to Q_3$.

Row 2 in Table 11.6 provides upper bound $(1, 1)$ for all capabilities in P. So we can state that P is confluent. Note that the same process would not be recognised as confluent by the type system in [KPT99].

11.6.4 Lower Bounds on the Number of Prefixes

The type system is not limited to statements of the form: "there *at most* n processes concurrently using channel x", we can also guarantee that there are *at least* m processes concurrently using channel x. In order to achieve this, we use the type system above and just invert the partial order, i.e. we take $\geq$ instead of $\leq$ (see Table 11.5, rows 4 & 5), *out* and *in* remain unchanged. This means also that the join $\vee$ in the new partial order is now the meet $\wedge$ of the original partial order. Typing P does not give us much information, since we cannot guarantee that there are any prefixes active at any given time (see Table 11.6, row 4) for any channel. In fact, some lower bounds are even (-1) stating that the respective channel "removes" input (or output) prefixes instead of making them available. In this case $P \to^* \mathbf{0}$, which means that no lower bounds can be guaranteed.

Typing P' yields the monoid elements given in Table 11.6 (row 5), which states that input prefixes with subjects c, d are available infinitely often.

Another lower bounds analysis will be presented in Section 11.6.6.

11.6.5 Avoiding Blocked Output Prefixes

Another interesting feature is to avoid blocked prefixes, i.e. prefixes which are waiting for a non-existing communication partner. We will first define—with the help of a lattice-ordered monoid—what it means for an output prefix to be blocked.

We take $\mathbb{Z}^\infty \times \mathbb{Z}^\infty$ as an ℓ-monoid and define a new partial order: $(i, j) \sqsubseteq (i', j')$ iff $i \leq i'$ and $j \geq j'$ (see Table 11.5, rows 6 & 7). The first component represents the number of output prefixes and the second the number of input prefixes of the same subject. Let $out = (1, 0)$ and $in = (0, 1)$. We say a name x is *output-blocked* in P, if $P \to^* P'$, $C_x(P') \sqsupseteq (1, 0)$ (i.e. there is at least one output prefix with subject x and no corresponding input prefixes) and for all P'' with $P' \to^* P''$ it follows that $C_x(P'') \sqsupseteq (1, 0)$ (no communication with x will ever take place).

We can, e.g., avoid this situation, by demanding that it is always the case that $C_x(P') = (a, b)$ and either $a \leq 0$ or $b \geq 1$ (i.e. $(a, b) \not\sqsupseteq (1, 0)$). We take the ℓ-monoid and out, in introduced above. This type system can be obtained by composing a type system establishing upper bounds for output prefixes and one establishing lower bounds for input prefixes (see Section 11.5.2). In this way we find out that no output prefixes with subjects c and d are output-blocked in P' (see Table 11.6, row 6, where the tuples are composed out of the first component of the tuples in row 3 and the second component of the tuples in row 5).

This type system is not the only way to check for blocked prefixes. There are alternatives which can be employed in case this version fails. We can take $\mathbb{Z}^\infty$ as a monoid with the ordinary partial order $\leq$, and define $out = 1$, $in = -1$. Now we can guarantee that a channel x in P is non-blocking if $C_x(P) \leq 0$, i.e. if there are always at least as many active input prefixes as output prefixes.

11.6.6 Availability of Printers

In order to give another example for the usefulness of the instance type system guaranteeing lower bounds, we consider the following scenario. We assume that there are two printers, printer 1 and printer 2, and several print services Pr_i, ready to accept tasks for either of the two printers. A print service Pr_i (see Table 11.7) receives a task via pr_i and then discards (or rather prints) it. Print services are not addressed directly by the process *User*, but via a print manager PM, which receives a task x, which should be printed, enquires for an available print service at the address server, sends the task x there and starts another print service for the same printer immediately afterwards. The address server AS distributes names of print services, in this case we assume that it outputs names pr_1 and pr_2 in an alternating order. The user constantly sends tasks to be printed to the print manager. Table 11.7 contains process *System*, which describes the entire system. Note that the address server and the user need extra triggers tr_1 and tr_2 since only input prefixes can be replicated.

Print manager:	$PM = !pm(x).as(y).\overline{y}\langle x\rangle.y(t)$
Address server:	$AS = !tr_1.\overline{as}\langle pr_1\rangle.\overline{as}\langle pr_2\rangle.\overline{tr_1}$
Print service:	$Pr_i = pr_i(t)$
User:	$User = !tr_2.\overline{pm}\langle t_1\rangle.\overline{pm}\langle t_2\rangle.\overline{tr_2}$
System:	$System = (\nu tr_1 : t_{tr_1})(\nu tr_2 : t_{tr_2})(\nu pm : t_{pm})((\nu as : t_{as})(PM \mid AS) \mid Pr_1 \mid Pr_1 \mid Pr_2 \mid Pr_2 \mid \overline{pr_1}\langle t\rangle \mid User \mid \overline{tr_1} \mid \overline{tr_2} \ )$

Table 11.7: Example: availability of printers

The names of the print services pr_1 and pr_2 are free and can still be addressed from the outside. We want to make sure that there are always sufficiently many print services available to the environment.

We apply an instance type system similar to the one described in Section 11.6.4, taking the monoid $\mathbb{Z}^\infty$ with $\geq$ as partial order and setting $out = 0$, $in = 1$. This means we only use the second component of each pair in the type system described in Section 11.6.4. We obtain the following type environment.

$$\begin{aligned} \Gamma \;=\;& pr_1 : [[\,]^0]^1, pr_2 : [[\,]^0]^2, t_1 : [\,]^0, t_2 : [\,]^0 \\ & \text{and } t_{tr_1} = t_{tr_2} = [\,]^0,\ t_{pm} = [[\,]^0]^\infty,\ t_{as} = [[[\,]^0]^0]^{-2} \end{aligned}$$

We can summarise the results of this analysis as follows (including also bound names):

name x	pr_1	pr_2	t_1	t_2	pm	as	tr_1	tr_2
$C_x(System) \geq$	1	2	0	0	∞	-2	0	0

In this way one can see that there is always at least one print service Pr_1 and at least two print services Pr_2 available at any time.

11.7 Related Work and Conclusion

We have presented a generic type system, which can be instantiated in order to analyse input/output capabilities of π-calculus processes. The notion of capabilities is kept very general, we only assume that they can be described by elements of a lattice-order monoid.

Inspiration for this work came from papers deriving information on the behaviour of a process by inspecting its input/output capabilities, such as [PS96, KPT99, NS97]. We now make a closer comparison.

- *Kobayashi, Pierce, Turner: "Linearity and the pi-calculus"* [KPT99]

  A type system that has close connections to ours is the linear type system by Kobayashi, Pierce and Turner [KPT99], since it also involves the typing of input/output capabilities of processes. A channel being linear means that the channel is used exactly once for input and output. The type system checks whether each channel is linear or not. (A slightly different variant of this type system is presented in [IK00], including a type inference algorithm.)

  One aim of [KPT99] is to study a process equivalence relation under the linear type system. This question has not been addressed in this paper, it is an interesting direction for future work.

  While the type system in [KPT99] counts all input and output prefixes of a process, we determine upper bounds on the number of prefixes which are concurrently active. Otherwise our type system can be seen as an extension of the variant of the linear type system given in [IK00]. The case of replication is handled in a slightly different way in our type system.

  Our type system still implies confluence and partial confluence respectively. The process P in our examples is identified as a confluent process (see Section 11.6.3), while this would not be the case in the type system in [KPT99].

  We could also directly use the cababilities of [IK00] in our framework. Consider the ℓ-monoid $\{0, 1, \omega\} \times \{0, 1, \omega\}$ where the first component of a tuple stands for input whereas the second component stands for output (see Section 11.6.2). Join and meet are defined component-wise where $0 \leq 1 \leq \omega$ is the partial order. Summation is also defined component-wise with $0 + x = x + 0 = x$, $x + \omega = \omega + x = \omega$ and $1 + 1 = \omega$ for every $x \in \{0, 1, \omega\}$. Residuation is defined and it holds that $x - y = 0$ whenever $x \leq y$, $x - 0 = 0$ and $\omega - 1 = 1$.

However, because of residuation and because of the fact that we count prefixes in a different way, the type systems may produce different results. In our framework we obtain $x\colon [\,]^{(1,0)} \vdash x.\overline{x}.\mathbf{0}$, whereas the type system of [IK00] typechecks $x\colon [\,]^{(1,1)} \vdash x.\overline{x}.\mathbf{0}$.

Using the capabilities of [KPT99] however, fails, since the structure defined there can be seen as an ℓ-monoid, but not as a residuated ℓ-monoid.

- *Naoki Kobayashi, Shin Saito, Eijiro Sumii: "An implicitly-typed deadlock-free process calculus"* [KSS00]

  While this type systems aims mainly at avoiding deadlocks, it is also interesting in that is considers resource usage analysis. For example, omitting the annotations, an assignment of the form $x\colon [\tilde{\tau}]/I.O.0$ tells us that name x is first used for input and then for output. It might be conceivable to compute monoid elements representing capabilities from these usage expressions.

  However, this does not work right away. For example the process $x.\overline{y}.0$ is assigned a type environment of the form $\Gamma = x\colon []^{t_x}/I.0, y\colon []^{t_y}/O.0$, from which it is not clear that y is not currently active. The tag ordering (in this case (t_x, t_y)), which gives some information about nesting of channels is an over-approximation, and can not be used immediately to compute, for example, lower bounds.

- *Pierce, Sangiorgi: "Typing and subtyping for mobile processes"* [PS96]

  In the work of Pierce and Sangiorgi, input/output behaviour of π-calculus processes is checked in a very refined way, using co- and contravariant types. It is not entirely clear how to incorporate the concept of co- and contravariance into our framework in a generic way, although it might lead to sharper bounds in some cases.

- *Kobayashi, Nakade, Yonezawa: "Static Analysis of Communication for Asynchronous Concurrent Programming Languages"* [KNY95]

  This paper presents an analysis technique for determining upper bounds for the number of enqueued messages and processes in HACL (Higher-order ACL). This corresponds to the analysis presented in Section 11.6.2. There are several similarities to our approach, especially since both approaches use subtraction.

  The technique presented in [KNY95] is an effect system rather than a type system and also because of the nature of HACL, which contains function abstraction and application, this effect system presented has a different flavour than our type system.

- *Honda: "Composing processes"* [Hon96]

  Our work has a similar aim as that of Honda [Hon96], in that it attempts to describe a general framework for process analysis using type systems. We concentrate on a more specialised but still generic type system, which enables us to prove the subject reduction property for the general case.

We have shown that, despite its generality, the type system can be instantiated in order to yield type systems related to existing ones. We have also shown how to parameterise type systems and what kind of parameters are feasible (in our case an ℓ-monoid).

- *Igarashi, Kobayashi: "A generic type system for the pi-calculus"* [IK01]

  This paper shows how to approximate π-calculus processes by processes of a simpler process calculus, thereby allowing the analysis of deadlock-freedom and race-freedom. This certainly gives a very powerful type system, even if it is often not entirely obvious how to extract information from a process type.

  Our aim was to give a less complex type system. It is clear that such a type system as ours might not be sufficient whenever a very detailed analysis of a process is required, but can certainly be very useful for fast debugging and for obtaining a first approximation of the capabilities of a process.

We did not introduce a type inference algorithm for our type system, but it should be possible to design a type inference algorithm along the lines of [IK00].

Our type system was partly inspired by a type system for a graph-based process calculus with graphs as types, which make it rather easy to add additional behaviour information (via morphisms and categorical functors) [Kön99a]. Graph-based type systems with lattices instead of monoids were presented in [Kön99b, 1]. For lattices or non-negative cones of ℓ-monoids, generic type systems are often easier to define. The main complication arises from non-positive elements and residuation.

There is also some similarity to dataflow analysis for which exists the concept of monotone frameworks which are parameterised over lattices [NNH99].

In order to conduct process analysis concerning more complex properties (as was done e.g. in [Kob98, BDNN98]) it is necessary to use type systems assigning behaviour information (i.e. monoid elements in our case) not only to single channels, but rather to tuples of channels or other more complex structures. This normally results in a semi-additive type system, in the terminology of Honda [Hon96], while our present type system is strictly additive. In order to extend this type system, a first solution would be to allow monoid labels for n-ary tuples of names. Another idea is to integrate it into the categorical framework presented in [Kön99b], which would allow us to specify very general behaviour descriptions.

We believe that generic type systems can be developed into tools suitable for fast debugging and the analysis of concurrent programs.

Acknowledgements: I would like to thank the anonymous referees for their detailed and extremely helpful comments.

Chapter 12

A Static Analysis Technique for Graph Transformation Systems

(Joint work with Paolo Baldan and Andrea Corradini)

Abstract

In this paper we introduce a static analysis technique for graph transformation systems. We present an algorithm which, given a graph transformation system and a start graph, produces a finite structure consisting of a hypergraph decorated with transitions (Petri graph) which can be seen as an approximation of the Winskel style unfolding of the graph transformation system. The fact that any reachable graph has an homomorphic image in the Petri graph and the additional causal information provided by transitions allow us to prove several interesting properties of the original system. As an application of the proposed technique we show how it can be used to verify the absence of deadlocks in an infinite-state Dining Philosophers system.

12.1 Introduction

Graphs are very useful to describe complex structures in a direct and intuitive way. Graph Transformation Systems (GTSs) [Roz97] add to the static description given by graphs a further dimension which models graph evolution via the application of rules, usually having local effects only. GTSs have been recognized to have fruitful applications in various fields of Computer Science [EEKR99], and specifically in the modelling and specification of concurrent and distributed systems [EKMR99].

As a high-level specification formalism for concurrent systems, GTSs are known to be more expressive than (Place/Transition) Petri nets, which can be seen, indeed, as GTSs acting on discrete graphs only (i.e., multisets of tokens) [Cor96]. However, even if the theory of GTSs is nowadays well developed and a number of tools for the support of specifications based on this formalism have been developed, GTSs are not yet used as widely as Petri nets. One reason for this could be the lack of analysis techniques, which have been proven to be extremely effective for Petri nets. Verification and validation techniques play an important role during the design of the specification of a complex system, as they offer the designer the possibility to raise confidence in the quality of the specification, for example by allowing the early detection of logical errors.

While several static analysis techniques have been proposed for Petri nets, ranging from the calculus of invariants [Rei85] to model checking based on finite complete prefixes [McM93],[1] the rich literature on GTSs does not contain many contributions to the static analysis of such systems (see [Koc00, 1]).

In this paper we present an original analysis technique for a class of (hyper)graph transformation systems, which, given a system and a start hypergraph, extracts from them an *approximated unfolding*, which is a finite structure (called *Petri graph*) consisting of a hypergraph and of a P/T net over it. Both the graphical and Petri net components of the approximated unfolding can be used to analyze the original system. For example, we will show that every hypergraph reachable from the start graph can be mapped homomorphically to the (graphical component of the) approximated unfolding. Therefore, if a property over graphs is reflected by graph morphisms, then if it holds on the approximated unfolding it also holds on all reachable graphs. Among these properties we mention the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). Furthermore, the transitions of the Petri net component of the approximated unfolding can be seen as (approximated) occurrences of rules of the original graph transformation system, and indeed every reachable graph of the GTS corresponds (in a sense formalized later) to a reachable marking of the net. This allows one to prove other properties directly on the Petri net component, including upper and lower bounds on the number of times an edge with a certain label is present in a reachable graph and certain causal dependencies among rule applications. Notice that in general the net component of the approximated unfolding is neither safe nor acyclic; roughly one can say that, at least for certain properties, the analysis of a graph transformation system can be reduced to the analysis of a Petri net, which is a computationally less powerful model and for which the existing analysis techniques can be used.

The construction of the approximated unfolding of a graph transformation system is similar in spirit to the construction of the finite complete prefix [McM93] of a net, but more complex. Both are based on the unfolding construction, which in the case of nets [NPW81] unwinds a Petri net into a branching occurrence net (a particularly simple Petri net satisfying suitable acyclicity and conflict freeness requirements), behaviourally equivalent to the original net. The unfolding cannot be used "directly" for verification purposes, since it is usually infinite. In the case of bounded nets, McMillan has observed

[1] We use the term "static analysis" in a quite wide meaning, as for example it is used in the community of the Static Analysis Symposia.

in [McM93] that it is possible to truncate the unfolding in such a way that the resulting finite structure, the *finite complete prefix*, contains as much information as the unfolding itself, and can therefore be used for checking efficiently behavioural properties ([Esp94, ERV66, VSY98]).

The unfolding construction has been generalized to graph transformation systems [Rib96, BCM99, Bal00], and the technique we propose makes use of unfolding steps for generating the (finite) approximated unfolding, but the analogy with the finite prefix construction of nets ends here. In fact the GTSs we consider are not finite-state in general, hence, we must abandon the idea of finding a complete *finite* part of the unfolding, where every state reachable in the considered GTS has an *isomorphic* image. Even if we relax the last requirement, by asking only that every reachable state has an *homomorphic* image in the constructed unfolding, since the states of the systems we consider are more structured (graphs *versus* multisets), it is not possible to rudely truncate the unfolding construction: at certain stages we have to merge parts of the unfolding already constructed. Because of this merging, the resulting structure is not acyclic (unlike the finite complete prefixes), and part of the information on the causality and concurrency of the system is lost. For what concerns state reachability, every state reachable in the original system is also reachable in the approximated unfolding, but we loose the converse implication (which instead holds for the finite complete prefix).

Technically, the algorithm that computes the approximated unfolding of a GTS is defined through two basic transformations, called *unfolding* and *folding* operations, which are applied as long as possible to the (Petri graph representing the) start graph of the system. Since both folding and unfolding are applied only if certain conditions are satisfied, the algorithm can be shown to terminate, a fact which guarantees that the resulting Petri graph is finite. Furthermore, although the proposed algorithm is non-deterministic, a local confluence property of the unfolding and folding transformations ensures that the approximated unfolding of a GTS is uniquely determined.

The paper is organized as follows. In Section 12.2 we introduce the class of GTSs on which our static analysis technique will be defined, as well as Petri graphs and some basic operations on them. The algorithm computing the approximated unfolding of a GTS is presented in Section 12.3, while Section 12.4 collects the main results about the algorithm, namely its termination, its confluence, and the fact that every reachable graph can be mapped to a reachable subgraph of the approximated unfolding. Section 12.5 illustrates the proposed method by applying it to the classical dining philosophers, both in a finite- and in an infinite-state variant. Section 12.6 concludes and hints at possible developments of the ideas presented in the paper.

12.2 Hypergraph rewriting, Petri nets and Petri graphs

In this section we first introduce the class of (hyper)graph transformation systems considered in the paper. Then, after recalling some basic notions for Petri nets, we will define Petri graphs, the structure combining hypergraphs and Petri nets, which will be used to approximate the behaviour of GTSs.

12.2.1 Graph transformation systems

In the following, given a set A we denote by A^* the set of finite strings of elements of A. Furthermore, if $f : A \to B$ is a function then we denote by $f^* : A^* \to B^*$ its extension to strings. Throughout the paper Λ denotes a fixed set of *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$.

Definition 12.2.1 (hypergraph) *A* (Λ-)hypergraph *G is a tuple (V_G, E_G, c_G, l_G), where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G : E_G \to {V_G}^*$ is a connection function and $l_G : E_G \to \Lambda$ is the labelling function for edges satisfying $ar(l_G(e)) = |c_G(e)|$ for every $e \in E_G$. Nodes are not labelled.*

A node $v \in V_G$ is called isolated *if it is not connected to any edge, i.e. if there are no edges $e \in E_G$ and $u, w \in {V_G}^*$ such that $c_G(e) = uvw$.*

Let G, G' be (Λ-)hypergraphs. A hypergraph morphism *$\varphi : G \to G'$ consists of a pair of total functions $\langle \varphi_V : V_G \to V_{G'},\ \varphi_E : E_G \to E_{G'} \rangle$ such that for every $e \in E_G$ it holds that $l_G(e) = l_{G'}(\varphi_E(e))$ and ${\varphi_V}^*(c_G(e)) = c_{G'}(\varphi_E(e))$.*

In the sequel, when dealing with hypergraph morphisms we will often omit the subscripts V and E when referring to the components of a morphism φ.

Definition 12.2.2 (rewriting rule) *A* rewriting rule *r is a triple (L, R, α), where L and R are hypergraphs, called* left-hand side *and* right-hand side*, respectively, and $\alpha : V_L \to V_R$ is an injective mapping.*

A rule $r = (L, R, \alpha)$ is called basic *if l_L is injective, i.e., different edges in the left-hand side L have different labels, no node in L is isolated and no node in $V_R - \alpha(V_L)$ is isolated in R.*

In the following we will consider *only* basic rules. This restriction is not strictly needed, but makes the presentation simpler. For example, a morphism of a left-hand side into a hypergraph is completely determined by the image of its edges. Furthermore, to simplify the notation we will assume, without loss of generality, that $V_L \subseteq V_R$, $E_L \cap E_R = \emptyset$ and that the mapping α is the identity.

Intuitively, a rule $r = (L, R, \alpha)$ specifies that an occurrence of the left-hand side L can be "replaced" by R, according to the following definition.

Definition 12.2.3 (hypergraph rewriting) *Let $r = (L, R, \alpha)$ be a rewriting rule. A* match *of r in a hypergraph G is any morphism $\varphi : L \to G$. In this case we write $G \Rightarrow_{r,\varphi} H$ or simply $G \Rightarrow_r H$, where H is defined as follows: $V_H = V_G \uplus (V_R - V_L)$, $E_H = (E_G - \varphi(E_L)) \uplus E_R$, and if $\overline{\varphi} : V_R \to V_H$ is the obvious extension of φ then*

$$c_H(e) = \begin{cases} c_G(e) & \text{if } e \in E_G - \varphi(E_L) \\ \overline{\varphi}^*(c_R(e)) & \text{if } e \in E_R \end{cases},$$
$$l_H(e) = \begin{cases} l_G(e) & \text{if } e \in E_G - \varphi(E_L) \\ l_R(e) & \text{if } e \in E_R \end{cases}$$

Given a graph transformation system (GTS), *i.e., a finite set of rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$. Furthermore we will denote the transitive closure of $\Rightarrow_{\mathcal{R}}$ by $\Rightarrow^*_{\mathcal{R}}$. A GTS with a start graph $(\mathcal{R}, G_{\mathcal{R}})$ is called a* graph grammar.

The application of the rule r to G at the match φ first removes from G the image of the edges of L. Then the graph G is extended by adding the new nodes in R (i.e., the nodes in $V_R - V_L$) and the edges of R. Observe that the (images of) the nodes in L are "preserved", i.e., not affected by the rewriting step.

The reader which is familiar with the double-pushout (DPO) approach [Ehr79] to graph rewriting would have recognized that our rules (L, R, α) can be seen as DPO rules ($L \hookleftarrow V_L \overset{\alpha}{\hookrightarrow} R$) and that our notion of rewriting is equivalent to a DPO construction. Hence compared to general DPO rules $L \overset{\varphi_L}{\leftarrow} K \overset{\varphi_R}{\rightarrow} R$ we have that (i) K is discrete, i.e., it contains no edges, (ii) no two edges in the left-hand side L have the same label, (iii) the morphism φ_L is surjective on nodes, (iv) V_L and $V_R - \varphi_R(V_K)$ do not contain isolated nodes.

12.2.2 Petri nets

In this subsection we fix some basic notation for Petri nets [Rei85, MM90]. Given a set A we will denote by $A^{\oplus}$ the free commutative monoid over A, whose elements will be called *multisets* over A. Given a function $f : A \to B$, by $f^{\oplus} : A^{\oplus} \to B^{\oplus}$ we denote its monoidal extension. On multisets $m, m' \in A^{\oplus}$, we use some common relations and operations, like *inclusion*, defined by $m \leq m'$ when there exists $m'' \in A^{\oplus}$ such that $m \oplus m'' = m'$ and *difference*, which, in the same situation, is defined by $m' - m = m''$. Furthermore, for $m \in A^{\oplus}$ and $a \in A$ we write $a \in m$ for $a \leq m$. The set underlying a multiset $m \in A^{\oplus}$ is defined by $[\![m]\!] = \{a \in A \mid a \in m\}$. Often we will confuse a subset $X \subseteq A$ with the multiset $\bigoplus_{x \in X} x$.

Definition 12.2.4 (Petri net) *Let A be a finite set of action labels. An A-*labelled Petri net *is a tuple $N = (S, T, {}^{\bullet}(), ()^{\bullet}, p)$ where S is a set of places, T is a set of transitions, ${}^{\bullet}(), ()^{\bullet} : T \to S^{\oplus}$ assign to each transition its pre-set and post-set and $p : T \to A$ assigns an action label to each transition.*

The Petri net is called irredundant *if there are no distinct transitions with the same label and pre-set, i.e., if for any $t, t' \in T$*

$$p(t) = p(t') \ \wedge \ {}^{\bullet}t = {}^{\bullet}t' \quad \Rightarrow \quad t = t'. \tag{12.1}$$

A marked Petri net *is pair (N, m_N), where N is a Petri net and $m_N \in S^{\oplus}$ is the initial marking.*

The irredundancy condition (12.1) requires that two distinct transitions differ for the label or for the pre-set. This condition, in the case of branching processes, allows one to interpret each transition as an occurrence of firing of a transition in the original net, uniquely determined by its causal history (see [Eng91]). Similarly, here it aims at avoiding the presence of multiple events which are indistinguishable for what regards the behaviour of the system. Hereafter *all* the considered Petri nets will be implicitly assumed irredundant, unless stated otherwise.

Definition 12.2.5 (causality relation) *Let N be a (marked) Petri net. The* causality relation $<_N$ *over N is the least transitive relation such that, for any $t \in T$, $s \in S$, we have (i) $s <_N t$ if $s \in {}^\bullet t$ and (ii) $t <_N s$ if $s \in t^\bullet$. For any $x \in S \cup T$ we define its sets of* causes $\lfloor x \rfloor = \{y \in S \cup T \mid y <_N x\}$ *and* consequences $\lceil x \rceil = \{y \in S \cup T \mid x <_N y\}$. *The definitions are extended in the obvious way to subsets of $S \cup T$.*

Observe that, since we want to use Petri nets to represent the causality structure of a system only in an approximated way, no assumptions are made concerning the acyclicity of the net.

12.2.3 Petri graphs

We next introduce the structure that we intend to use to approximate graph transformation systems, the so-called Petri graphs, which consist of an hypergraph and of a Petri net whose places are the edges of the graph.

Definition 12.2.6 (Petri graph) *Let $\mathcal{R}$ be a GTS. A Petri graph (over $\mathcal{R}$) is a tuple $P = (G, N, \mu)$ where G is a hypergraph, $N = (E_G, T_N, {}^\bullet(), ()^\bullet, p_N)$ is an $\mathcal{R}$-labelled Petri net where the places are the edges of G, and μ associates to each transition $t \in T_N$, with $p_N(t) = (L, R, \alpha)$, a hypergraph morphism $\mu(t) : L \cup R \to G$ such that*

$$ {}^\bullet t = \mu(t)^{\oplus}(E_L) \ \wedge \ t^\bullet = \mu(t)^{\oplus}(E_R) \tag{12.2} $$

A Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$ is a pair (P, ι) where $P = (G, N, \mu)$ is a Petri graph for $\mathcal{R}$ and $\iota : G_{\mathcal{R}} \to G$ is a graph morphism. The multiset $\iota^{\oplus}(E_{G_{\mathcal{R}}})$ is called the initial marking *of the Petri graph. A marking $m \in {E_G}^{\oplus}$ will be called* reachable (coverable) *in (P, ι) if it is reachable (coverable, i.e., there exists a reachable m' such that $m \le m'$) in the underlying Petri net.*

Condition (12.2) requires that each transition in the net can be viewed as an "occurrence" of a rule in $\mathcal{R}$. More precisely, if $p_N(t) = (L, R, \alpha)$ and $\mu(t) : L \cup R \to G$ is the morphism associated to the transition, then $\mu(t)_{|L} : L \to G$ must be a match of the rule in G such that the image of the edges of L in G coincides with the pre-set of t. Observe that, due to the assumption on the rules (no multiple labels and no isolated node in the left-hand side) the morphism $\mu(t)_{|L}$ (if it exists) is completely determined by ${}^\bullet t$. Then, the result of applying the rule to the considered match must be already in graph G, and the corresponding edges must coincide with the post-set of t. This is formalized by the condition over the image through $\mu(t)$ of the edges of R (note that the set E_R is seen as a multiset and $\mu(t)$ as a multiset function to take care of multiplicities).

Every hypergraph G can be considered as a Petri graph $[G] = (G, N, \mu)$ for $\mathcal{R}$, by taking N as the net with $S_N = E_G$ and no transitions. Similarly, $G_\mathcal{R}$ can be seen as Petri graph for $(\mathcal{R}, G_\mathcal{R})$ by taking as $\iota : G_\mathcal{R} \to G_\mathcal{R}$ the identity.

We now introduce a merging operation on Petri graphs which constructs the quotient of a Petri graph through an equivalence induced by a suitable relation.

Definition 12.2.7 (consistent and closed relation on a Petri graph) *Let $P = (G, N, \mu)$ be a Petri graph and let $\frown$ be a relation on $V_G \cup E_G \cup T_N$ (assume the sets V_G, E_G, T_N to be disjoint). We say that $\frown$ is* consistent *when (i) if $x \frown x'$ then $x, x' \in X$ for some $X \in \{V_G, E_G, T_N\}$, (ii) for all $e, e' \in E_G$ if $e \frown e'$ then $l_G(e) = l_G(e')$ and (iii) for all $t, t' \in T_N$, if $t \frown t'$ then $p_N(t) = p_N(t')$.*

A consistent relation $\frown$ over P is called closed *if for all $t, t' \in T_N$, $e, e' \in E_G$*

$$p_N(t) = p_N(t') = (L, R, \alpha) \ \wedge \ (\forall e \in E_L : \mu(t)(e) \frown \mu(t')(e)) \\ \Rightarrow \ t \frown t' \tag{12.3}$$

$$t \frown t' \Rightarrow \ \forall e \in E_L \cup E_R : \mu(t)(e) \frown \mu(t')(e) \tag{12.4}$$

$$e \frown e' \ \wedge \ c_G(e) = v_1 \dots v_m \ \wedge \ c_G(e') = v'_1 \dots v'_m \\ \Rightarrow \ \forall 1 \leq i \leq m : v_i \frown v'_i \tag{12.5}$$

To ensure that the quotient of a Petri graph with respect to a relation is well-defined and irredundant, the relation must be closed. Hence the simple observation below is essential for defining the merging operation.

Fact. Given any consistent relation $\frown$ over a Petri graph P there exists a least equivalence relation $\approx$ including $\frown$ and closed.

Definition 12.2.8 (Petri graph merging) *Let $P = (G, N, \mu)$ be a Petri graph and let $\frown$ be a consistent relation over P. Then the merging of P w.r.t. $\frown$, denoted by $P /\!\!/_\frown$, is the Petri graph (G', N', μ') defined as follows. Let $\approx$ be the least equivalence relation extending $\frown$ and closed in the sense of Definition 12.2.7. Then*

$$G' = (V_G/_\approx, E_G/_\approx, c_{G'}, l_{G'}),$$

where $c_{G'}([e]_\approx) = [v_1]_\approx \dots [v_n]_\approx$ and $l_{G'}([e]_\approx) = l_G(e)$ whenever $e \in E_G$ and $c_G(e) = v_1 \dots v_n$. Furthermore $N' = (E_{G'}, T_N/_\approx, {}^\bullet(), ()^\bullet, p_{N'})$, where ${}^\bullet[t]_\approx = \bigoplus_{e \in {}^\bullet t} [e]_\approx$, $[t]_\approx{}^\bullet = \bigoplus_{e \in t^\bullet} [e]_\approx$ and $p_{N'}([t]_\approx) = p_N(t)$ whenever $t \in T_N$. For each $t \in T_N$ the morphism $\mu'([t]_\approx)$ is defined by $\mu([t]_\approx)(x) = [\mu(t)(x)]_\approx$ for any graph item x in the rule $p_N(t)$.

Given a graph morphism $h : H \to G$ we will denote by $h /\!\!/_\frown : H \to G'$ the corresponding morphism, defined by $h /\!\!/_\frown(x) = [h(x)]_\approx$ for any $x \in V_H \cup E_H$.

The merging operation can be extended to sets of Petri graphs. Let $P_i = (G_i, N_i, \mu_i)$, with $i \in \{1, \dots, n\}$, be Petri graphs and assume that the sets V_{G_i}, E_{G_i}, T_{N_i} are pairwise disjoint. Then the componentwise union $P = P_1 \cup \dots \cup P_n$ is a Petri graph. A relation $\frown$ over $P_1, \dots, P_n$ is called consistent (closed) if it is a consistent (closed) relation over P. Given a consistent relation over $P_1, \dots, P_n$, we define the merging $\{P_1, \dots, P_n\} /\!\!/_\frown = P /\!\!/_\frown$.

12.2.4 The category of Petri graphs

Although the constructive definition of Petri graph merging given above is more intuitive and more suited for the presentation of an algorithm, in our proofs it is more convenient to work in the category of Petri graphs defined below, and to exploit the fact that Petri graph merging can be expressed by a colimit construction.

Definition 12.2.9 (category of Petri graphs) *A* Petri graph morphism *is a pair* $\psi = (\varphi, \tau) : (G, N, \mu) \to (G', N', \mu')$ *where*

- $\varphi : G \to G'$ *is a hypergraph morphism;*
- $\tau : T_N \to T_{N'}$ *is a mapping such that for every* $t \in T_N$*,* ${}^\bullet\tau(t) = \varphi^\oplus({}^\bullet t)$ *and* $\tau(t)^\bullet = \varphi^\oplus(t^\bullet)$*, and* $p_{N'} \circ \tau = p_N$ *(in other words,* $(\varphi_E, \tau) : N \to N'$ *is a Petri net morphism).*
- *for every* $t \in T_N$*,* $\mu'(\tau(t)) = \varphi \circ \mu(t)$*.*

The category of Petri graphs and Petri graph morphisms is denoted by PG*.*

In the context of this paper we are only interested in coequalizers and pushouts, but it is also possible to express other forms of colimits in terms of the merging operation.

Proposition 12.2.10 (coequalizers and pushouts in PG**)** *Let* (φ_1, τ_1), $(\varphi_2, \tau_2) : P \to P'$ *be two morphisms in* PG*, with* $P = (G, N, \mu)$*. The coequalizer of* $(\varphi_1, \tau_1), (\varphi_2, \tau_2)$ *is isomorphic to* $P/\!\!/_\frown$ *where* $\frown$ *is defined as*

$$\varphi_1(v) \frown \varphi_2(v) \quad \varphi_1(e) \frown \varphi_2(e) \quad \tau_1(t) \frown \tau_2(t)$$

for $v \in V_G$*,* $e \in E_G$ *and* $t \in T_N$*.*

Let $(\varphi_i, \tau_i) : P \to P_i$*,* $i \in \{1, 2\}$ *be two morphisms in* PG*, with* $P = (G, N, \mu)$*. The pushout* $(\varphi_1, \tau_1), (\varphi_2, \tau_2)$ *is isomorphic to* $\{P_1, P_2\}/\!\!/_\frown$*, where* $\frown$ *is defined as*

$$\varphi_1(v) \frown \varphi_2(v) \quad \varphi_1(e) \frown \varphi_2(e) \quad \tau_1(t) \frown \tau_2(t)$$

for $v \in V_G$*,* $e \in E_G$ *and* $t \in T_N$*.*

12.3 Algorithm computing the approximated unfolding

In this section we describe an algorithm which computes the approximated unfolding of a graph grammar. Given a graph grammar, the algorithm produces a *finite* Petri graph such that every graph reachable in the grammar corresponds, in a sense formalized later, to a marking which is reachable in the Petri graph.

Let $(\mathcal{R}, G_\mathcal{R})$ be a graph grammar. Its ordinary unfolding [Rib96, BCM99] is constructed inductively beginning from the start graph and then applying at each step in all possible ways the rules, without deleting the left-hand side, and recording each occurrence of a rule and each new graph item generated in the rewriting process. As a result one obtains an acyclic branching graph grammar describing the behaviour of $(\mathcal{R}, G_{\mathcal{R}})$. In particular every reachable graph embeds in (a concurrent subgraph of) the graph produced by the unfolding construction.

The unfolding is usually infinite, also in the case of finite-state systems. Here, to ensure that the our algorithm produces a finite structure, we consider—besides the *unfolding rule*, which extends the graph by simulating the application of a rule without deleting its left-hand side—a *folding rule*, which allows us to "merge" two occurrences of the left-hand side of a rule whenever they are, in a sense made precise later, one causally dependent on the other.

Definition 12.3.1 (folding operation) *Let $P = (G, N, \mu)$ be a Petri graph for a GTS $\mathcal{R}$. Let $r = (L, R, \alpha) \in \mathcal{R}$ be a rule and let $\varphi', \varphi : L \to G$ be matches of r in G. Let $\frown$ be the relation over P defined as follows: for every $e \in E_L$*

$$\varphi'(e) \frown \varphi(e).$$

The folding *of P at the matches φ', φ is the Petri graph* $\mathsf{fold}(P, r, \varphi', \varphi) = P/\!\!/_{\frown}$. *If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define* $\mathsf{fold}((P, \iota), r, \varphi', \varphi) = (P/\!\!/_{\frown}, \iota/\!\!/_{\frown})$.

To introduce the unfolding operation, we first need to fix some notation. If t is a transition and $r = (L, R, \alpha)$ is a rule we will write $P(t, r)$ to denote the Petri graph $(L \cup R, N, \mu)$ where $N = (E_{L\cup R}, \{t\}, {}^\bullet t = E_L, t^\bullet = E_R, p_N(t) = r)$ and $\mu(t) = id_{L\cup R}$. Whenever we can find a match of rule r in a given Petri graph, the unfolding operation extends the Petri graph by merging $P(t, r)$ at the match.

Definition 12.3.2 (unfolding operation) *Let $P = (G, N, \mu)$ be a Petri graph for a GTS $\mathcal{R}$. Let $r = (L, R, \alpha) \in \mathcal{R}$ be a rule and let $\varphi : L \to G$ be a match of r in G. Let $\frown$ be the relation over $\{P, P(t, r)\}$ defined as follows: for every $e \in E_L$*

$$\varphi(e) \frown e.$$

The unfolding *of P with rule r at match φ is the Petri graph* $\mathsf{unf}(P, r, \varphi) = \{P, P(t, r)\}/\!\!/_{\frown}$. *If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define* $\mathsf{unf}((P, \iota), r, \varphi) = (\{P, P(t, r)\}/\!\!/_{\frown}, \iota/\!\!/_{\frown})$.

We can now describe the algorithm which produces the approximated unfolding of a given graph grammar. The algorithm generates a sequence of Petri graphs, beginning from the start graph and applying, non-deterministically, at each step, a folding or unfolding operation, until none of such steps is admitted.

Definition 12.3.3 (approximated unfolding) *Let $(\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar. The algorithm generates a sequence $(P_i, \iota_i)_{i\in\mathbb{N}}$ of Petri graphs as follows.*

(Step 0) *Initialize $(P_0, \iota_0) = ([G_{\mathcal{R}}], id_{G_{\mathcal{R}}})$.*

(Step $i+1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i. Choose non-deterministically one of the following actions*

⋆ *Folding: Find a rule $r = (L, R, \alpha)$ in $\mathcal{R}$ and two matches $\varphi', \varphi : L \to G_i$ of r such that*

- *$\varphi^{\oplus}(E_L)$ is a coverable marking in P_i;*
- *there exists a transition $t \in T_{N_i}$ such that*

$$p_{N_i}(t) = r \ \wedge \ {}^{\bullet}t = \varphi'^{\oplus}(E_L) \ \wedge \ \forall e \in \varphi^{\oplus}(E_L) : (e \in {}^{\bullet}t \ \vee \ t <_{N_i} e). \quad (12.6)$$

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{fold}((P_i, \iota_i), r, \varphi', \varphi)$.

⋆ *Unfolding: Find a rule $r = (L, R, \alpha)$ in $\mathcal{R}$ and a match $\varphi : L \to G_i$ such that*

- *$\varphi^{\oplus}(E_L)$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^{\bullet}t = \varphi^{\oplus}(E_L)$ and $p_{N_i}(t) = r$;*
- *there is no other match $\varphi' : L \to G_i$ satisfying condition (12.6).*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

If no folding or unfolding step can be performed, the algorithm terminates. The resulting Petri graph (P_i, ι_i) is called the approximated unfolding *of $(\mathcal{R}, G_{\mathcal{R}})$ and denoted by $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$.*

Condition (12.6) basically states that we can fold two matches of a rule r whenever the first one has been already unfolded producing a transition t, and the second match depends on the first one, in the sense that any edge in the second match is already in the first one or causally depends on t. Roughly, the idea is that we should not unfold a left-hand side again, if we have already done the same unfolding step in its past, since this might lead to infinitely many steps. There are some similarities, to be further investigated, with the work in [Gen98] where the sets of descendants and of normal forms of term rewriting systems are approximated by constructing an approximation automaton. Condition $e \in {}^{\bullet}t$ in the disjunction is really needed: omitting such condition it would be possible to construct an input grammar on which the algorithm does not terminate.

The coverability of a marking can be decided by computing the coverability tree of the net, as described in [Rei85]. If this gets too costly, the condition of coverability can be relaxed or checked in an approximated way, a choice which does not compromise the result of correctness (see Proposition 12.4.2), but only reduces the "precision" of the algorithm: it will generate a worse approximation, where less properties of the given GTS can be proved.

12.4 Correctness, termination and confluence of the algorithm

We show that the algorithm described in the previous section is correct, namely that every reachable graph of a grammar is represented in the approximated unfolding produced by the algorithm. Furthermore the algorithm is terminating and confluent. Hence, by a classical result, its result is uniquely determined.

Correctness.

We first show that the computed Petri graph is an appropriate approximation of the given graph grammar, in the sense that for any graph reachable in the graph grammar, there is a morphism into the approximated unfolding such that the image of its edge set corresponds to a reachable marking.

For this purpose we need the following lemma which will also be relevant for the proof of confluence. It states that a folding operation, as introduced in Definition 12.3.2, corresponds to a coequalizer and that an unfolding operation, as introduced in Definition 12.3.1, corresponds to a pushout in the category PG.

Lemma 12.4.1 *Let $P = (G, N, \mu)$ be a Petri graph, let $r = (L, R, \alpha)$ be a rewriting rule and let $\varphi', \varphi : L \to G$ be graph morphisms.*

Then $P' = \mathsf{fold}(P, r, \varphi', \varphi)$ is the coequalizer of $(\varphi', \emptyset), (\varphi, \emptyset) : [L] \to P$ in the category PG where $\emptyset$ is a function having the empty set of transitions as a domain. Now let $\iota : G_{\mathcal{R}} \to G$ such that (P, ι) is a Petri graph for a grammar and let $(\varphi_0, \tau_0) : P \to P'$ be the arrow generated by the coequalizer. Then it holds that $(P', \varphi_0 \circ \iota) = \mathsf{fold}((P, \iota), r, \varphi', \varphi)$.

Moreover $\mathsf{unf}(P, r, \varphi)$ is the pushout of $(\varphi, \emptyset) : [L] \to P, (id_L, \emptyset) : [L] \to P(t, r)$ in the category PG. Again let (P, ι) be a Petri graph for a grammar and let $(\varphi_0, \tau_0) : P \to P'$ be one of the arrows generated by the pushout. Then it holds that $(P', \varphi_0 \circ \iota) = \mathsf{unf}((P, \iota), r, \varphi)$.

Proposition 12.4.2 *Let $(\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and assume that the algorithm computing the approximated unfolding terminates producing the Petri graph $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}}) = ((U, N, \mu), \iota)$.*

Then for every graph G with $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ there exists a morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$. Furthermore, if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$.

Proof: By induction on the length of the derivation $\Rightarrow^*_{\mathcal{R}}$:

Length 0: It holds that $G = G_{\mathcal{R}}$. In this case we simply set $\varphi = \iota$ and $m = \varphi^{\oplus}(E_{G_{\mathcal{R}}})$ and the result holds since m is the initial marking of the Petri graph which is trivially reachable.

Length $k+1$: Assume that $\underbrace{G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} H}_{k \text{ steps}} \Rightarrow_{\mathcal{R}} G$.

From the induction hypothesis it follows that there exists a morphism $\varphi_H : H \to U$, such that the marking $m_H = {\varphi_H}^{\oplus}(E_H)$ is reachable in $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$.

Let $r = (L, R, \alpha)$ be the rule applied in the last rewrite step, transforming H into G, and let $\eta_L : L \to H$ be the corresponding match of r in H. From the point of view of the double-pushout approach we have the following situation, where both squares below are pushouts:

$$\begin{tikzcd} L \arrow[d, "\eta_L"'] & V_L \arrow[l, hook', "\iota_L = id"'] \arrow[r, hook, "\iota_R = id"] \arrow[d, "\eta"] & R \arrow[d, "\eta_R"] \\ H & D \arrow[l, "\eta_1"'] \arrow[r, "\eta_2"] & G \end{tikzcd}$$

It holds that $\varphi_H \circ \eta_L : L \to U$ and $(\varphi_H \circ \eta_L)^{\oplus}(E_L) \overset{(\eta_L)_E \text{ inj.}}{=} {\varphi_H}^{\oplus}(\eta_L(E_L)) \leq {\varphi_H}^{\oplus}(E_H) = m_H$, which means that the marking corresponding to the image of the left-hand side is coverable.

Since the algorithm terminates it holds that no further folding or unfolding operations were applicable. Thus there must be a transition $t \in T_N$ such that $p_N(t) = r$ and the morphism $\mu(t) : L \cup R \to U$ satisfies $^{\bullet}t = (\varphi_H \circ \eta_L)^{\oplus}(E_L) = \mu(t)^{\oplus}(E_L)$, $\mu(t)_{|L} = \varphi_H \circ \eta_L$.

If we set $\eta'_R = \mu(t)_{|R}$, then $t^{\bullet} = {\eta'_R}^{\oplus}(E_R)$ and we obtain the following diagram

$$\begin{tikzcd} L \arrow[d, "\eta_L"'] & V_L \arrow[l, hook', "\iota_L"'] \arrow[r, hook, "\iota_R"] \arrow[d, "\eta"] & R \arrow[d, "\eta_R"] \arrow[ddr, "\eta'_R"] & \\ H \arrow[drrr, "\varphi_H"'] & D \arrow[l, "\eta_1"'] \arrow[r, "\eta_2"] & G \arrow[dr, dashed, "\varphi"] & \\ & & & U \end{tikzcd}$$

We have to check that it commutes, that is that $(\varphi_H \circ \eta_1) \circ \eta = \varphi_H \circ \eta_L \circ \iota_L$ and $\eta'_R \circ \iota_R$ are equal: let $v \in V_L$ which implies that $(\varphi_H \circ \eta_L \circ \iota_L)(v) = (\varphi_H \circ \eta_L)(v) = \mu(t)(v) = \eta'_R(v) = (\eta'_R \circ \iota_R)(v)$.

This implies the existence of a morphism $\varphi : G \to U$ such that $\varphi \circ \eta_2 = \varphi_H \circ \eta_1$ and $\varphi \circ \eta_R = \eta'_R$.

By firing transition t we can immediately reach the marking

$$\begin{aligned} & m_H - {^{\bullet}t} \oplus t^{\bullet} = {\varphi_H}^{\oplus}(E_H) \\ & \qquad -(\varphi_H \circ \eta_L)^{\oplus}(E_L) \oplus {\eta'_R}^{\oplus}(E_R) \\ \overset{(\eta_L)_E \text{ inj.}}{=} \quad & {\varphi_H}^{\oplus}(E_H - \eta_L(E_L)) \oplus {\eta'_R}^{\oplus}(E_R) \\ = \quad & {\varphi_H}^{\oplus}(\eta_1(E_D)) \oplus {\eta'_R}^{\oplus}(E_R) \\ \overset{(\eta_1)_E \text{ inj.}}{=} \quad & (\varphi_H \circ \eta_1)^{\oplus}(E_D) \oplus (\varphi \circ \eta_R)^{\oplus}(E_R) \\ = \quad & (\varphi \circ \eta_2)^{\oplus}(E_D) \oplus (\varphi \circ \eta_R)^{\oplus}(E_R) \\ \overset{(\eta_R)_E, (\eta_2)_E \text{ inj.}}{=} \quad & \varphi^{\oplus}(\eta_2(E_D) \oplus \eta_R(E_R)) = \varphi^{\oplus}(E_G) \end{aligned}$$

□

Termination.

The basic result towards the proof of termination shows that it is not possible to perform infinitely many unfolding steps, without having the folding condition satisfied at some stage. This property is independent of the graph structure and can be proved by considering only the causality structure of a Petri graph, as expressed by the underlying Petri net. More formally, we show that in any infinite Petri net, satisfying suitable acyclicity and well-foundedness requirement, there exists a pair of transitions t, t' (called a folding pair) such that the pre-set of t' is dependent on t in the sense of Condition (12.6) in Definition 12.3.3. Let us start formalizing the notion of folding pair.

Definition 12.4.3 *Let $N = (S, T, {}^\bullet(), ()^\bullet, p)$ be a Petri net. A* folding pair *in N is a pair of transitions $t, t' \in T$ such that $t \neq t'$, $p(t) = p(t')$ and for all $s \in {}^\bullet t'$ either $s \in {}^\bullet t$ or $t <_N s$.*

We will also need an operation which removes from a given Petri net a subnet and all its consequences, as expressed by the following definition.

Definition 12.4.4 *Let $N = (S, T, {}^\bullet(), ()^\bullet, p)$ be a Petri net and let $Q \subseteq T$. We define*

$$\begin{aligned} N - Q \quad = \quad & (S' = S - \{s \mid \exists t \in Q \colon t <_N s\}, \\ & T' = T - \{t' \mid \exists t \in Q \colon (t <_N t' \vee t = t')\},\ {}^\bullet()_{|T'}, ()^\bullet{}_{|T'}, p_{|T'}), \end{aligned}$$

i.e., all elements of Q and their consequences are removed from N.

The next key lemma ensures that in any infinite net obtained by applying only unfolding steps there exists a folding pair.

Lemma 12.4.5 *Let $N = (S, T, {}^\bullet(), ()^\bullet, p)$ be an infinite irredundant Petri net, labelled over a finite set A, and satisfying the following conditions:*

- *for any $x \in S \cup T$ the set $\lfloor x \rfloor$ (the causes of x) is finite;*
- *the set $Min(N) = \{s \mid \lfloor s \rfloor = \emptyset\}$ is finite, i.e., only finitely many places have an empty set of causes;*
- *the relation $<_N$ is acyclic;*
- *the pre-set ${}^\bullet t$ of each transition is a set (rather than a proper multiset);*
- *for $t, t' \in T$ with $p(t) = p(t')$ it holds that $|{}^\bullet t| = |{}^\bullet t'|$.*

Then net N contains a folding pair.

Proof: If Q is empty, the lemma is trivially true, otherwise let m be the cardinality of the pre-sets of all elements of Q.

The proof is conducted by (downward) induction on $n = |\bigcap_{t \in Q} [\![{}^\bullet t]\!]|$.

- Let $n = m$ where m is the upper bound for n. In this case ${}^\bullet t = {}^\bullet t'$ for all $t, t' \in Q$, since furthermore $p(t) = p(t')$ it follows from the fact that N is irredundant that Q contains at most one element, it is therefore finite and we can set $Q' = \emptyset$.

- We assume that the lemma holds for all sets Q' which satisfy $|\bigcap_{t\in Q'}[\![{}^\bullet t]\!]| > n$ and we assume that $|\bigcap_{t\in Q}[\![{}^\bullet t]\!]| = n$.

  Now let M be the set of minimal elements of Q with respect to $<_N$. We distinguish the following cases:

  - Q contains a folding pair. Then we are done.
  - M is infinite. In this case we consider the set $S' = \bigcup_{t\in M}[\![{}^\bullet t]\!]$. By contradiction we can show that S' is infinite: if S' is finite, then we can derive from the irredundancy and from the fact that we have finitely many rewrite rules that the elements of S' can be the direct causes of only finitely many transitions. But this is a contradiction since M is infinite.

    Since the elements of M are minimal the set S' is still contained in the places of $N - M = N - Q$ and since the presence of infinitely many places implies the presence of infinitely many transitions (following from the fact that originally we have only finitely many places), we can infer that $N - Q$ is still infinite and in this case we can set $Q' = Q$.
  - M is finite and Q does *not* contain a folding pair. We show by induction on $|M|$ that there is a $Q' \subseteq Q$ such that $Q - Q'$ is finite and $N - Q'$ is infinite.
    * Let $|M| = 0$. In this case Q itself is empty and it suffices to set $Q' = \emptyset$.
    * We assume that the statement holds for all M with $|M| \leq k$ and we now have $|M| = k + 1$.

      We choose one $t \in M$ and since there is no folding pair, then for every $t' \in Q - \{t\}$ there must be a place $s_{t'} \in {}^\bullet t'$ such that $t \not< s_{t'}$ and $s_{t'} \notin {}^\bullet t$. Otherwise we would have found a folding pair.

      Since for every $t' \in Q - \{t\}$ it holds that $s_{t'} \notin {}^\bullet t$, specifically we have that $s_{t'} \notin \bigcap_{t\in Q}[\![{}^\bullet t]\!]$.

      We now consider the following two cases:
      - the set $\{s_{t'} \mid t' \in Q - \{t\}\}$ is finite, i.e. it has the form $\{s_1, \ldots, s_l\}$. We can now define $Q_i = \{t \in Q \mid s_i \in {}^\bullet t\}$ and it holds that $Q = \{t\} \cup Q_1 \cup \ldots \cup Q_l$.

        For every one of the Q_i it holds that $|\bigcap_{t\in Q_i}[\![{}^\bullet t]\!]| \geq n + 1$ and we can apply the (outer) induction hypothesis which implies that we can—one after the other—remove almost all of the elements of the sets $Q_1, \ldots, Q_l$ from N and obtain a still infinite Petri net. That is there is a set $Q' \subseteq Q - \{t\}$ such that $Q - Q'$ is finite and $N - Q'$ is infinite. Note that by removing Q_i, we might also remove some elements of Q_j with $j > i$.

        More formally this can be shown by induction on l.
      - the set $\{s_{t'} \mid t' \in Q - \{t\}\}$ is infinite, but since it is not contained in the consequences of t, the Petri net $N' = N - \{t\}$

still contains infinitely many places and therefore infinitely many transitions.

Furthermore the set of minimal elements of $P = Q \cap T_{N'}$ is $M - \{t\}$ which has cardinality k and thus the (inner) induction hypothesis is applicable. It implies that there is a set $P' \subseteq P$ such that $P - P'$ is finite and $N' - P'$ is infinite. We now set $Q' = P' \cup \{t\} \cup \{t' \in Q \mid t <_N t'\}$ and it follows that $Q' \subseteq Q$, furthermore $Q - Q' = (Q \cap S_{N'}) - P' = P - P'$ and is therefore finite. And it holds that $N - Q' = N' - P'$ which is infinite.

□

By using Lemma 12.4.5 we can show that there cannot be an infinite net without a folding pair.

Lemma 12.4.6 *If $N = (S, T, {}^\bullet(), ()^\bullet, p)$ is a Petri net satisfying the conditions of Lemma 12.4.5, then it contains a folding pair.*

Proof: Let $A' = \{a \in A \mid \exists^\omega t \colon p(t) = a\}$, i.e. the set of all action labels that occur infinitely often in the net. Since A is finite, it follows immediately that A' is also finite. We proceed by induction on $|A'|$.

- If $|A'| = 0$, then N is finite and the lemma is trivially true.
- We assume that the lemma holds for the case where k rewrite rules occur infinitely often and we assume that $|A'| = k + 1$. Choose one $a \in A'$ and regard the set $Q = \{t \in T \mid p(t) = a\}$.

  Then according to Lemma 12.4.5 it either holds that Q contains a folding pair and we are done or we can remove almost all the elements of Q and retain an infinite net N'. Since in N' only k rewrite rules occur infinitely often, but N' is still infinite, it follows from the induction hypothesis that N' contains a folding pair, which is also a folding pair of N.

□

The above lemma ensures that in our algorithm a folding step will be eventually performed. We have yet to show termination of the algorithm.

Proposition 12.4.7 *The algorithm computing the approximated unfolding (see Definition 12.3.3) terminates for every graph grammar $(\mathcal{R}, G_{\mathcal{R}})$.*

Proof (Sketch): In parallel to the computation of the approximated unfolding we construct a second *acyclic* Petri net N' as follows. For every unfolding step we add to N' a new transition, corresponding to the transition added to the approximated unfolding. The net N' is left unchanged in a folding step. The construction is done in order to ensure the existence of a surjective net morphism from N' to its "folded" counterpart, i.e., the net N_i underlying the Petri graph constructed by the algorithm.

Suppose by contradiction that the algorithm does not terminate. Hence the net N' gets infinitely large and therefore, by Lemma 12.4.6, it contains a folding pair $\bar{u}$, $\bar{t}$. The image of such a folding pair through the net morphism from N' to the net underlying the approximated unfolding $\mathcal{U}(\mathcal{G}, G_{\mathcal{G}})$, provides a folding pair u, t also in $\mathcal{U}(\mathcal{G}, G_{\mathcal{G}})$. But then we can show that the second transition t in the pair could never have been added to the Petri graph since this would have been a violation of the third condition of the unfolding step in Definition 12.3.3.

Assume we are given a start graph $G_0 = G_{\mathcal{R}}$ and a set of rewrite rules $\mathcal{R}$ as the input to the algorithm. This results in a sequence $P_0 = (G_0, N_0, \mu_0), P_1 = (G_1, N_1, \mu_1), \ldots$ of Petri graphs. Our aim is to show that this sequence will eventually terminate.

- In parallel to the unfolding/folding of the graph we construct a sequence of tuples $(N'_0, \beta_0), (N'_1, \beta_1), \ldots$ where the N'_i are irredundant Petri nets satisfying the conditions of Lemma 12.4.5 and the $\beta_i : N'_i \to N_i$ are net morphisms. This sequence is constructed in the following way:

  start tuple: we set $N'_0 = N_0$ and $\beta_0 : N'_0 \to N_0$ is the identity.

  unfolding step: we assume that P_{i+1} was obtained by an unfolding step, i.e., a transition t' was added to N_i, with $p_{N_{i+1}}(t') = (L, R, \alpha)$ and there is a PG morphism $(\varphi_{i+1}, \tau_{i+1}) : P_i \to P_{i+1}$, the existence of which is ensured by Lemma 12.4.1. We set $\varphi_L = \mu_{i+1}(t')_{|L}$ and $\varphi_R = \mu_{i+1}(t')_{|R}$.
  We assume that $E_L = \{e_1, \ldots, e_k\}$ and $E_R = \{e'_1, \ldots, e'_l\}$. For every edge e_j of the left-hand side we choose a $s_j \in \beta_i^{-1}(\varphi_L(e_j))$. (Since we will later show that all the β_i are surjective, such an s_i always exists.)
  Furthermore let $\bar{t}$ be a new transition and let $s'_1, \ldots, s'_l$ be new places. We construct N'_{i+1} as follows:

  $$\begin{aligned} N'_{i+1} \quad = \quad & (S_{N'_i} \cup \{s'_1, \ldots, s'_l\}, T_{N'_i} \cup \{\bar{t}\}, \\ & {}^\bullet() \cup \{\bar{t} \mapsto \{s_1, \ldots, s_l\}\}, ()^\bullet \cup \{\bar{t} \mapsto \{s'_1, \ldots, s'_k\}\}, \\ & p_{N'_i} \cup \{\bar{t} \mapsto r\}). \end{aligned}$$

  And β_{i+1} is set to

  $$\beta_{i+1} = (((\varphi_{i+1})_E, \tau_{i+1}) \circ \beta_i) \cup \{s'_j \mapsto \varphi_R(e'_j) \mid 1 \leq j \leq l\} \cup \{\bar{t} \mapsto t'\}$$

  where t' is the new transition in N_{i+1}.

  folding step: we assume that P_{i+1} was obtained by a folding step and there is a PG morphism $(\varphi_{i+1}, \tau_{i+1}) : P_i \to P_{i+1}$, the existence of which is again implied by Lemma 12.4.1.
  In this case we set $N'_{i+1} = N'_i$, $\beta_{i+1} = ((\varphi_{i+1})_E, \tau_{i+1}) \circ \beta_i$.

  Note that the described procedure is non-deterministic since we have several possibilities to choose the s_j in the unfolding step. Furthermore the construction is defined in such a way that the places and transitions of every net N'_i are contained in the places and transitions of N'_{i+1}.

- By induction on i we can show that the following invariants hold:
  - every occurrence net N_i satisfies the conditions of Lemma 12.4.5.
  - the mappings β_i are surjective.
  - the β_i are net morphisms, i.e. for every transition $t \in T_{N'_i}$ it holds that ${}^\bullet(\beta_i(t)) = \beta_i{}^\oplus({}^\bullet t)$ and $(\beta_i(t))^\bullet = \beta_i{}^\oplus(t^\bullet)$. And furthermore $p_{N'_i} = p_{N_i} \circ \beta_i$.
    (By definition of $<$ this implies that $x <_{N'_i} y$ for $x, y \in S_{N'_i} \cup T_{N'_i}$ implies $\beta_i(x) <_{N_i} \beta_i(y)$.)

  The first two conditions and the fact that the β_i preserve action labels are straightforward to check, we only prove that the β_i preserve pre-sets and post-sets: since N'_0 does not contain any transitions, it is obvious that the invariant holds for N'_0 and β_0. We have to show that it is also preserved by unfolding and folding steps:

  unfolding step: now let $t \in T_{N'_{i+1}}$. We distinguish the following two cases:
  - $t \in T_{N'_i}$, which means that $\beta_i(t) \in T_{N_i}$. Therefore the induction hypothesis implies that ${}^\bullet(\beta_i(t)) = \beta_i{}^\oplus({}^\bullet t)$ and $(\beta_i(t))^\bullet = \beta_i{}^\oplus(t^\bullet)$.
    It holds that $\beta_{i+1}(t) = \tau_{i+1}(\beta_i(t))$, which implies that ${}^\bullet(\beta_{i+1}(t)) = {}^\bullet(\tau_{i+1}(\beta_i(t))) = \varphi_{i+1}{}^\oplus({}^\bullet\beta_i(t)) = \varphi_{i+1}{}^\oplus(\beta_i{}^\oplus({}^\bullet t)) = \beta_{i+1}{}^\oplus(t)$.
    In the same way we can show that $(\beta_{i+1}(t))^\bullet = \beta_{i+1}{}^\oplus(t^\bullet)$.
  - $t = \bar{t}$ and the transition was added by the last construction step. In this case ${}^\bullet(\beta_{i+1}(\bar{t})) = {}^\bullet t' = \varphi_L{}^\oplus(\{e_1, \dots, e_k\}) = \beta_{i+1}{}^\oplus(\{s_1, \dots, s_k\}) = \beta_{i+1}{}^\oplus({}^\bullet\bar{t})$.
    We can also show that $(\beta_{i+1}(\bar{t}))^\bullet = t'^\bullet = \varphi_R{}^\oplus(\{e'_1, \dots, e'_l\}) = \beta_{i+1}{}^\oplus(\{s'_1, \dots, s'_l\}) = \beta_{i+1}{}^\oplus(\bar{t}^\bullet)$.

  folding step: the mapping β_{i+1} is obviously a net morphism since it is the composition of two net morphisms $((\varphi_{i+1})_E, \tau_{i+1})$ and β_i.

  From the fact that the β_i are net morphisms, we can also show, by contradiction, that every N'_i is irredundant: we assume that we add (in an unfolding step) a new transition $\bar{t}$ to N'_i with a pre-set ${}^\bullet\bar{t} = \{s_1, \dots, s_k\}$, but there is already a transition $\bar{u} \in T_{N'_i}$ such that ${}^\bullet\bar{u} = \{s_1, \dots, s_k\}$. We set $u' = \beta_i(\bar{u})$ and it holds that ${}^\bullet u' = \beta_i({}^\bullet\bar{u}) = \varphi_L{}^\oplus(E_L)$ and $p_{N_i}(u') = p_{N'_i}(\bar{u}) = r$. But this implies that the third condition for the unfolding step was violated, i.e. there is a contradiction.

- We now assume that the algorithm does not terminate, which implies that it makes infinitely many unfolding steps (folding steps decrease the size of the graph G_i). But since unfolding steps increase the size of N'_i and folding steps do not alter its size, it follows that the infinite union

$N' = \bigcup_{i=1}^{\infty} N_i'$ (defined in the obvious way) is infinite. We can check that also the infinite net N' satisfies all the conditions of Lemma 12.4.5. The finiteness conditions holds since adding a new transition $\bar{t}$ and new places $s_1', \dots, s_k'$ in the unfolding step does not alter the causes of already existing transitions and places. Furthermore $\lfloor \bar{t} \rfloor = \bigcup_{i=1}^{l} \lfloor s_i \rfloor$ which is finite and $\lfloor s_j' \rfloor = \lfloor \bar{t} \rfloor \cup \{\bar{t}\}$ which is also finite. And finally we never introduce places which have no causes, and therefore the size of $Min(N_i')$ is constant.

Therefore we can apply Lemma 12.4.6 and obtain the existence of a folding pair $\bar{u}, \bar{t} \in T_{N'}$ where $\bar{u} \neq \bar{t}$, $p(\bar{u}) = p(\bar{t}) = r$ and $\forall s' \in {}^\bullet\bar{t}$: $(s' \in {}^\bullet\bar{u} \vee \bar{u} <_{N'} s')$.

- We assume that $\bar{t}$ was added when N_{i+1}' was constructed from N_i', which must have consequently been an unfolding step, adding the transition $\beta_{i+1}(\bar{t}) = t'$ to N_i. It is our aim to show that this unfolding operation could never have been applied and thus obtain a contradiction.

  Since the causes of an already existing transition are never altered during the construction of the N_{i+1}', the folding pair $\bar{u}, \bar{t}$ is already present in N_{i+1}'.

  We set $u' = \beta_{i+1}(\bar{u})$ and since $\bar{u}$ is already present in N_i', it holds that $u' = \tau_{i+1}(\beta_i(\bar{u})) \in T_{N_i}$, which implies that $u' \neq t'$ (an unfolding step does not merge any transitions). Since the mapping p of the Petri nets is preserved by β_{i+1}, it also holds that $p_{N_{i+1}}(u') = p_{N_{i+1}}(t') = r = (L, R, \alpha)$.

  Now let $e' \in {}^\bullet t'$ and from the construction of N_{i+1}' it follows that there is an $s' \in \beta_{i+1}^{-1}(e')$ such that $s' \in {}^\bullet\bar{t}$. It follows from the folding condition that $s' \in {}^\bullet\bar{u}$ or $\bar{u} <_{N_{i+1}'} s'$.

  In the former case it follows that $e' = \beta_{i+1}(s') \in {}^\bullet(\beta_{i+1}(\bar{u})) = {}^\bullet u'$ and in the latter case it follows that $u' = \beta_{i+1}(\bar{u}) <_{N_{i+1}} \beta_{i+1}(s') = e'$. (In both cases we use that fact that β_i is a net morphism and therefore preserves the causality relation.)

- We set $\varphi_u = \mu_{i+1}(u')_{|L}$ and $\varphi_t = \mu_{i+1}(t')_{|L}$ and it holds that that ${\varphi_u}^{\oplus}(E_L) = {}^\bullet u'$ and ${\varphi_t}^{\oplus}(E_L) = {}^\bullet t'$. Furthermore for every $e' \in {\varphi_t}^{\oplus}(E_L) = {}^\bullet t'$, either $e' \in {}^\bullet u'$ or $u' <_{N_{i+1}} e'$.

  Since the causes of u' and $\varphi_t(E_L)$ are not changed by the unfolding step, this means that the condition for the application of the folding step is satisfied, which forbids the application of the unfolding step and leads to a contradiction.

$\square$

Confluence.

In order to prove that the algorithm produces a uniquely determined result, independently of the order in which folding and unfolding steps are applied, we show that the rewriting relation on Petri graphs induced by folding and unfolding steps is locally confluent. We first need two preliminary lemmata. The first one observes that the coverability property of markings is preserved under Petri graph morphism.

Lemma 12.4.8 *Let $(N, m_N), (N', m_{N'})$ be two Petri nets with initial markings and let $\beta : N \to N'$ be a net morphism such that $m_{N'} = \beta^{\oplus}(m_N)$. If a marking m is coverable in (N, m_N), then $\beta^{\oplus}(m)$ is coverable in $(N', m_{N'})$.*

The second lemma shows that, under specific conditions, folding and unfolding steps have no effect on the Petri graph. However, notice that in both these cases the respective application conditions would not be satisfied.

Lemma 12.4.9 *Let (P, ι) with $P = (G, N, \mu)$ be a Petri graph for a graph grammar, let $r = (L, R, \alpha)$ be a rewriting rule and let $\psi : L \to G$ be an occurrence of the left-hand side in G. Then $\mathsf{fold}((P, \iota), r, \psi, \psi) \cong (P, \iota)$.*

If furthermore P contains a transition $t \in T_N$ such that $p_N(t) = r$ and ${}^{\bullet}t = \psi^{\oplus}(E_L)$, then $\mathsf{unf}((P, \iota), r, \psi) \cong (P, \iota)$.

We introduce the following notation: we write $(P, \iota) \rightsquigarrow^{unf}_{r,\psi} (P', \iota')$ whenever $(P', \iota') \cong \mathsf{unf}((P, \iota), r, \psi)$. We write $(P, \iota) \dashrightarrow^{unf}_{r,\psi} (P', \iota')$ whenever $(P, \iota) \rightsquigarrow^{unf}_{r,\psi} (P', \iota')$ and the application conditions of the unfolding step are satisfied.

In the same way $(P, \iota) \rightsquigarrow^{fold}_{r,\psi,\eta} (P', \iota')$ whenever $(P', \iota') \cong \mathsf{fold}((P, \iota), r, \psi, \eta)$. Again we write $(P, \iota) \dashrightarrow^{fold}_{r,\psi,\eta} (P', \iota')$ whenever $(P, \iota) \rightsquigarrow^{fold}_{r,\psi,\eta} (P', \iota')$ and the application conditions of the folding step are satisfied. The following proposition only holds if we consider irredundant Petri nets.

Proposition 12.4.10 *We write $(P, \iota) \dashrightarrow (P', \iota')$ whenever there is a rule r and morphisms ψ, η such that $(P, \iota) \dashrightarrow^{unf}_{r,\psi} (P', \iota')$ or $(P, \iota) \dashrightarrow^{fold}_{r,\psi,\eta} (P', \iota')$.*

Let $(P, \iota) \dashrightarrow (P_i, \iota_i)$ for $i \in \{1, 2\}$. Then there is a Petri graph (P', ι') such that $(P_i, \iota_i) \dashrightarrow^{*} (P', \iota')$.

Proof (Sketch): The proof mainly relies on the fact that both folding and unfolding operations can be described as special colimits in a suitable category of Petri graphs. Then a general categorical result that ensures the commutativity of colimits can be exploited. Finally, things must be accommodated to take into account also the applicability conditions of folding and unfolding steps as described in the algorithm (see Definition 12.3.3).

---

confluence without application conditions: if $(P, \iota) \dashrightarrow (P_i, \iota_i)$ then there are PG morphisms $(\varphi_i, \tau_i) : P \to P_i$ such that $\iota_i = \varphi_i \circ \iota$. Since both folding and unfolding steps can be described in terms of colimits (see Lemma 12.4.1) and colimits commute, it holds that

- $(P, \iota) \dashrightarrow^{unf}_{r_i,\psi_i} (P_i, \iota_i)$, $i \in \{1, 2\}$ implies $(P_1, \iota_1) \rightsquigarrow^{unf}_{r_2,\varphi_1 \circ \psi_2} (P', \varphi'_1 \circ \iota_1)$ and $(P_2, \iota_2) \rightsquigarrow^{unf}_{r_1,\varphi_2 \circ \psi_1} (P', \varphi'_2 \circ \iota_2)$ for a Petri graph P' which is the colimit of φ_1 and φ_2.

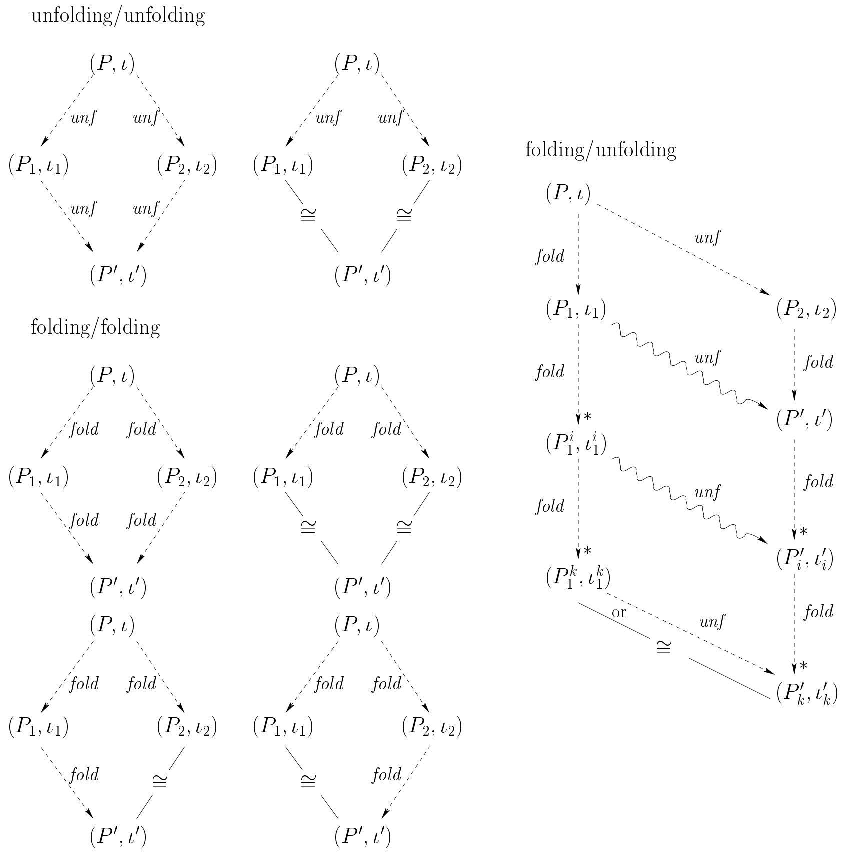


Figure 12.1: Confluence of the rewriting system

By $(\varphi_i', \tau_i') : P_i \to P'$ we denote the arrows generated by this colimit. It holds that $\varphi_1' \circ \iota_1 = \varphi_1' \circ \varphi_1 \circ \iota = \varphi_2' \circ \varphi_2 \circ \iota = \varphi_2' \circ \iota_2$.

- $(P, \iota) \dashrightarrow^{\mathit{fold}}_{r_i, \psi_i, \eta_i} (P_i, \iota_i)$, $i \in \{1, 2\}$ implies $(P_1, \iota_1) \rightsquigarrow^{\mathit{fold}}_{r_2, \varphi_1 \circ \psi_2, \varphi_1 \circ \eta_2} (P', \iota')$ and $(P_2, \iota_2) \rightsquigarrow^{\mathit{fold}}_{r_1, \varphi_2 \circ \psi_1, \varphi_2 \circ \eta_1} (P', \iota')$ for some Petri graph (P', ι'). (The morphism ι' can be computed as above.)
- $(P, \iota) \dashrightarrow^{\mathit{fold}}_{r_1, \psi_1, \eta_1} (P_1, \iota_1)$ and $(P, \iota) \dashrightarrow^{\mathit{unf}}_{r_2, \psi_2} (P_2, \iota_2)$ implies $(P_1, \iota_1) \rightsquigarrow^{\mathit{unf}}_{r_2, \varphi_1 \circ \psi_2} (P', \iota')$ and $(P_2, \iota_2) \rightsquigarrow^{\mathit{fold}}_{r_1, \varphi_2 \circ \psi_1, \varphi_2 \circ \eta_1} (P', \iota')$ for some Petri graph (P', ι'). (The morphisms ι' can be computed as above.)

In this way we have shown confluence without regarding the application conditions. In the next step we show that the rewriting system is still confluent if we consider the conditions (also see Fig. 12.1).

unfolding/unfolding: Let $(P, \iota) \dashrightarrow^{\mathit{unf}}_{r_i, \psi_i} (P_i, \iota_i)$, $i \in \{1, 2\}$ where $P = (G, N, \mu)$, $P_i = (G_i, N_i, \mu_i)$, $r_i = (L_i, R_i, \alpha_i)$ and $\psi_i : L_i \to G$. We show that either the application conditions for the unfolding of rule r_2 at the occurrence $\varphi_1 \circ \psi_2 : L_2 \to G_1$ are satisfied or that (P_1, ι_1) and (P_2, ι_2) are isomorphic. (The same is true for the unfolding of rule r_1 in P_2.)

- we first have to show that $(\varphi_1 \circ \psi_2)^{\oplus}(E_{L_2})$ is coverable. This follows from the fact that ${\psi_2}^{\oplus}(E_{L_2})$ is coverable and from Lemma 12.4.8.
- if there is a transition $t \in T_{N_1}$ such that $p_{N_1}(t) = r_2$, ${}^{\bullet}t = (\varphi_1 \circ \psi_2)^{\oplus}(E_{L_2})$ and $\mu(t)_{|L_2} = \varphi_1 \circ \psi_2$, then t was introduced by the last unfolding step, since otherwise the unfolding of r_2 would not have been possible in P.

  This implies that $p_{N_1}(t) = r_1 = r_2$, $\mu(t)_{|L_1} = \varphi_1 \circ \psi_1 = \varphi_1 \circ \psi_2$. Since φ_1 is injective it follows that $\psi_1 = \psi_2$, therefore both steps unfold the same occurrence of a left-hand side and the results are therefore isomorphic.
- since the unfolding step does not change the causes of $(\varphi_1 \circ \psi_2)^{\oplus}(E_{L_2})$, the folding condition can not be satisfied, i.e. there is no other occurrence of L_2 such that the folding condition holds for the pair of the two occurrences.

folding/folding: Let $(P, \iota) \dashrightarrow^{fold}_{r_i,\psi_i,\eta_i} (P_i, \iota_i)$, $i \in \{1, 2\}$ where $P = (G, N, \mu)$, $P_i = (G_i, N_i, \mu_i)$, $r_i = (L_i, R_i, \alpha_i)$ and $\psi_i, \eta_i : L_i \to G$. We show that either the application conditions for the folding of the two occurrences $\varphi_1 \circ \psi_2, \varphi_1 \circ \eta_2 : L_2 \to G_1$ are satisfied or that (P_1, ι_1) and (P', ι') are isomorphic. (The same is true for the corresponding folding in P_2.)

- we first have to show that $(\varphi_1 \circ \eta_2)^{\oplus}(E_{L_2})$ is coverable. This follows from the fact that ${\eta_2}^{\oplus}(E_{L_2})$ is coverable and from Lemma 12.4.8.
- if the two occurrences $\varphi_1 \circ \psi_2, \varphi_1 \circ \eta_2 : L_2 \to G_1$ are equal, and we know that $(P_1, \iota_1) \rightsquigarrow_{r_2,\varphi_1\circ\psi_2,\varphi_1\circ\eta_2} (P', \iota')$, it follows from Lemma 12.4.9 that (P_1, ι_1) and (P', ι') are isomorphic.
- the folding condition for the occurrences $\psi_2, \eta_2 : L_2 \to G$ is satisfied, i.e. there is a transition t with $p_N(t) = r_2$ and ${}^{\bullet}t = {\psi_2}^{\oplus}(E_{L_2})$ and every $e \in {\eta_2}^{\oplus}(E_{L_2})$ is either a member of ${\psi_2}^{\oplus}(E_{L_2})$ or $t <_N e$. Since (φ_1, τ_1) is a PG morphism, it follows that there is a transition $t' = \tau_1(t)$ such that every $e' \in (\varphi_1 \circ \eta_2)^{\oplus}(E_{L_2})$ is either a member of $(\varphi_1 \circ \psi_2)^{\oplus}(E_{L_2})$ of $t' <_{N_1} e'$.

  Therefore the folding condition is again satisfied.

folding/unfolding: Let $(P, \iota) \dashrightarrow^{fold}_{r_1,\psi_1,\eta_1} (P_1, \iota_1)$ and $(P, \iota) \dashrightarrow^{unf}_{r_2,\psi_2} (P_2, \iota_2) = (G_2, N_2, \mu_2)$ where $P = (G, N, \mu)$ and $P_i = (G_i, N_i, \mu_i)$. This is the most difficult case, since the application of the folding step might invalidate the application condition of the unfolding step.

- With the same argumentation as above (see folding/folding) we can show that either (P_2, ι_2) and (P', ι') are isomorphic or the conditions for the folding step are satisfied and $(P_2, \iota_2) \dashrightarrow^{fold}_{r_1,\varphi_2\circ\psi_1,\varphi_2\circ\eta_1} (P', \iota')$. Furthermore $(P_1, \iota_1) \rightsquigarrow^{unf}_{r_2,\varphi_2\circ\psi_2} (P', \iota')$ and there is a PG morphism $(\varphi', \tau') : (P_1, \iota_1) \to (P', \iota')$ (see Fig. 12.1).
- But in P_1 $\varphi_1 \circ \psi_2 : L_2 \to G_1$ might be part of a folding pair $\zeta_1, \varphi_1 \circ \psi_2 : L_2 \to G_1$, which forbids unfolding, and after folding this pair, the

image of L_2 might again be part of a folding pair, etc. So there is a (possibly empty) sequence of transitions of the form

$$\begin{aligned}(P_1, \iota_1) &= (P_1^0, \iota_1^0) \dashrightarrow^{fold}_{r_2,\zeta_1,\varphi_1\circ\psi_2} (P_1^1, \iota_1^1) \dashrightarrow^{fold}_{r_2,\zeta_2,\varphi^1\circ\varphi_1\circ\psi_2} \\ &\quad \ldots \dashrightarrow^{fold}_{r_2,\zeta_k,\varphi^k\circ\ldots\circ\varphi^1\circ\varphi_1\circ\psi_2} (P_1^k, \iota_1^k)\end{aligned}$$

where the $(\varphi^{i+1}, \tau^{i+1}) : P_1^i \to P_1^{i+1}$ are PG morphisms. We assume that in the Petri graph P_1^k no further folding step of this form is applicable. The sequence must terminate since a folding step decreases the size of a Petri graph.

- Now we can construct another sequence of Petri graphs starting from (P', ι') of the form $(P', \iota') = (P'_0, \iota'_0), \ldots, (P'_k, \iota'_k)$ such that (P'_i, ι'_i) and (P'_{i+1}, ι'_{i+1}) are either isomorphic, or there is a folding step from one to the other. And furthermore it holds that

$$(P_1^i, \iota_1^i) \rightsquigarrow^{unf}_{r_2,\varphi^i\circ\ldots\circ\varphi^1\circ\varphi_1\circ\psi_2} (P'_i, \iota'_i) \tag{12.7}$$

  We show this by induction on i, it clearly holds for $i = 0$.

  We now assume that (12.7) holds and let $(\varphi'_i, \tau'_i) : P_1^i \to P'_i$ be the corresponding PG morphism. As previously (see folding/folding) we can argue that $\varphi'_i \circ \zeta_i, \varphi'_i \circ \varphi^i \circ \ldots \circ \varphi^1 \circ \varphi_1 \circ \psi_2$ either are equal or satisfy the conditions for the application of a folding rule. In the former case we set $(P'_{i+1}, \iota'_{i+1}) = (P'_i, \iota'_i)$ and in the latter case we define $(P'_{i+1}, \iota'_{i+1}) = \mathsf{fold}((P'_i, \iota'_i), r_2, \varphi'_i \circ \zeta_i, \varphi'_i \circ \varphi^i \circ \ldots \circ \varphi^1 \circ \varphi_1 \circ \psi_2)$.
  Since colimits commute it also holds that

$$(P_1^{i+1}, \iota_1^{i+1}) \rightsquigarrow^{unf}_{r_2,\varphi^{i+1}\circ\ldots\circ\varphi^1\circ\varphi_1\circ\psi_2} (P'_{i+1}, \iota'_{i+1})$$

- So finally we reach two Petri graphs (P_1^k, ι_1^k) with $P_1^k = (G_1^k, N_1^k, \mu_1^k)$ respectively (P'_k, ι'_k) with $P'_k = (G'_k, N'_k, \mu'_k)$ from (P_1, ι_1) respectively (P_2, ι_2) by folding steps, such that condition (12.7) holds for $i = k$.

  Now the application condition for folding is not satisfied any more and the occurrence of L_2 in G_1^k is still coverable since coverability is preserved by application of morphisms (see Lemma 12.4.8). The only condition that might forbid the application of the unfolding step is the existence of a transition $t \in T_{N_1^k}$ such that ${}^\bullet t = (\varphi^k \circ \ldots \circ \varphi^1 \circ \varphi_1 \circ \psi_2)^{\oplus}(E_{L_2})$. But then we can derive from Lemma 12.4.9 that (P_1^k, ι_1^k) and (P'_k, ι'_k) are isomorphic.

  If there is no such transition we directly get $(P_1^k, \iota_1^k) \rightsquigarrow^{unf}_{r_2,\varphi^k\circ\ldots\circ\varphi^1\circ\varphi_1\circ\psi_2} (P'_k, \iota'_k)$.

□

Since for a rewriting system local confluence and termination imply confluence we conclude the following result.

Proposition 12.4.11 *For any input $(\mathcal{R}, G_{\mathcal{R}})$ the algorithm computing the approximated unfolding terminates with a result $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$ unique up to isomorphism.*

Proof: Recall that, by Proposition 12.4.10, the algorithm computing the approximated unfolding is confluent and, by Proposition 12.4.7, it is also terminating. Then uniqueness follows from the Diamond Lemma [DJ90], which states that local confluence and termination imply global confluence. □

12.5 The approximated unfolding at work: checking absence of deadlocks for dining philosophers

In order to illustrate our method, in this section we show how it can be applied to a well-known example, the dining philosophers system, which is presented in two versions, finite- and infinite-state.

Let us start with the classical finite-state version of the problem. Assume that sitting at the table are a left-handed philosopher and a right-handed philosopher with two forks between them. Our method is also applicable to instances of the problem with a greater number of philosophers. The restriction to two philosophers only avoids that the involved graphs become very large and hard to draw.

A philosopher, modelled by a binary edge, cycles through states H_X (hungry), W_X (waiting for the second fork), E_X (eating) where $X \in \{L, R\}$ depending on whether the philosopher is left- or right-handed. The thinking state is omitted. A fork is also represented by a binary edge labelled F. The system is described by the set of rewriting rules and by the start graph depicted in Fig. 12.2. A rule (L, R, α) is drawn in the form $L \Rightarrow R$, where edges are depicted by square boxes which are connected to a source node (the first node) and a target node (the second node). The mapping α is indicated by dashed arrows.

The algorithm in Definition 12.3.3 produces the Petri graph (a) in Fig. 12.3. Transitions are depicted by small rectangles and the connection to their pre-sets and post-sets is indicated by dashed arrows.

The algorithm terminates after six unfolding steps and four folding steps. Two unfolding steps which apply rules $(Wait_L)$ and (Eat_L), respectively, to edge H_L with the corresponding forks, give rise to edge E_L. Then a further unfolding step using rule $(Hungry_L)$ unfolds this edge into a graph consisting of two edges labelled F and one edge labelled H_L. But this graph consists of two left-hand sides of previously applied rules and the edges are causally dependent on the corresponding transitions. Hence two folding steps can be applied, that merge the three edges (F, H_L and F) of the newly unfolded graph with the original edges from which they were derived. A symmetric reasoning applies for edge H_R.

We would like to prove that no deadlocks can occur in the system. First observe that any reachable graph is a cycle and, since an eating philosopher can always be reduced, a deadlocked state is necessarily a cycle including only

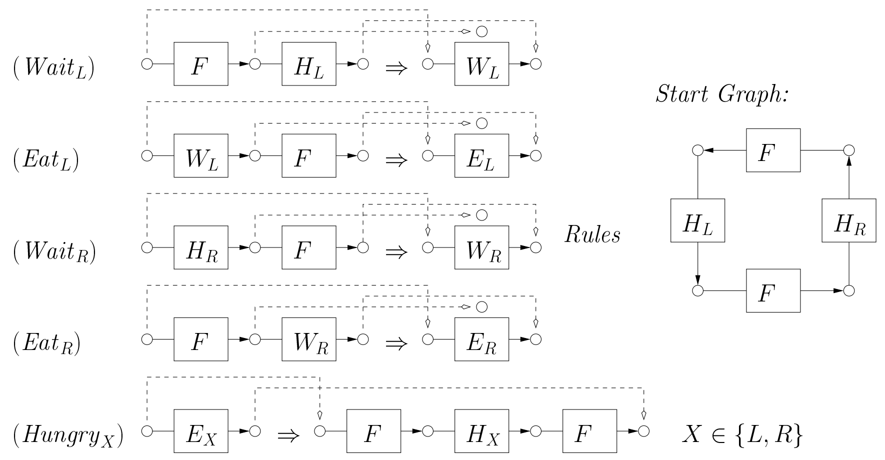
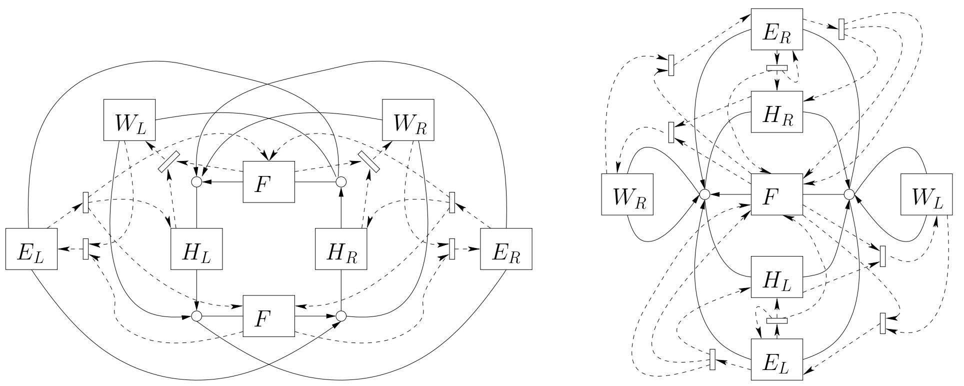


Figure 12.2: A graph grammar modelling the dining philosophers (finite-state version).



(a)

(b)

Figure 12.3: Approximated unfoldings as Petri graphs: (a) dining philosophers, finite-state version; (b) dining philosophers, infinite-state version.

hungry and waiting philosophers, where no forks are to the left of a left-handed hungry or a right-handed waiting philosopher and no forks are to the right of a right-handed hungry and a left-handed waiting philosopher. The absence of cycles is a property reflected by graph morphisms. Thus we can try to verify the absence of deadlocked states by analyzing cycles in the hypergraph associated to the approximated unfolding. To this aim we consider such graph as a finite-state automaton over the alphabet $\Sigma = \{F, H_X, W_X, E_X \mid X \in \{L, R\}\}$—with nodes as states and edges as transitions—and declare one of the four nodes as the initial and final state, thereby obtaining the languages L_{nw} (northwest node), L_{ne} (northeast node), L_{sw} (southwest node), L_{se} (southeast node). In this way we obtain all possible cycles of forks and philosophers as a regular language. By declaring, e.g., the northeast node as initial and final node we obtain the following language:

$$L_{ne} = (((FH_L + W_L)(E_R H_L)^* F + E_L)(W_R H_L (E_R H_L)^* F)^* H_R)^*.$$

An additional analysis of the Petri net would of course reveal that only a small finite subset of L_{ne} will ever occur, but here this is not needed for the analysis.

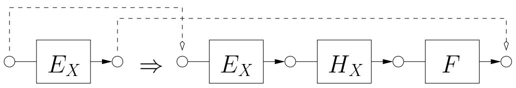
The language of all cycles allowing for the application of a rewrite rule is

$$\begin{aligned} L_{lhs} \quad = \quad & \Sigma^* E_L \Sigma^* + \Sigma^* E_R \Sigma^* + \Sigma^* F H_L \Sigma^* + H_L \Sigma^* F + \Sigma^* H_R F \Sigma^* + \\ & F \Sigma^* H_R + \Sigma^* W_L F \Sigma^* + F \Sigma^* W_L + \Sigma^* F W_R \Sigma^* + W_R \Sigma^* F. \end{aligned}$$

The language of all cycles which may occur but which do not allow the application of any rewriting rule can be now computed as $(L_{nw} \cup L_{ne} \cup L_{sw} \cup L_{se}) - L_{lhs} = \lambda$, i.e., the empty word. It is immediately clear that the circle of philosophers will never disappear entirely and thus we can conclude that no deadlocks will ever occur.

It is worth observing that if we forget about the graphical structure of the Petri graph, considering only the underlying Petri net, then we obtain a classical Petri net model of the dining philosophers. Therefore, in this case, the absence of deadlocks can be proved also by analyzing the Petri net underlying the approximated unfolding with classical Petri net techniques.

Now, in order to make things more interesting, we extend the example to an infinite-state system by adding a rule (Rep_X) which allows an eating philosopher to reproduce, creating another hungry philosopher with an adjacent fork.

(Rep_X) 

$X \in \{L, R\}$

Observe that we can reuse the unfolding of the finite-state case and continue by unfolding the edges E_R and E_L using the two new rules. A sequence of further unfolding and folding steps causes the causes the two pairs of opposite nodes in the square to collapse, ending up with Petri graph (b) in Fig. 12.3.

Again we would like to prove that no deadlocks can occur. By declaring the left-hand node as initial and final state, we obtain the following language:

$$L_{left} \quad = \quad (W_R^*((H_L + H_R) W_L^* (F + E_L + E_R))^*)^*$$

while using the right-hand node in the same role, we obtain the language:

$$L_{right} \quad = \quad (W_L^*((F + E_L + E_R) W_R^* (H_L + H_R))^*)^*.$$

The language of all cycles which may occur but which do not allow the application of a rewriting rule can be now computed as $(L_{left} \cup L_{right}) - L_{lhs} = W_L^* + W_R^*$. Then, an analysis of the Petri net underlying the approximated unfolding reveals that actually no marking which consists of tokens exclusively in W_L or of tokens exclusively in W_R is reachable from the initial marking which consists of two tokens on F and one token on H_L and H_R each. Hence the system will never reach a deadlock.

Observe that in this case the analysis of the underlying Petri net by itself is not sufficient. In fact the Petri net can deadlock: we start from the initial marking and after the firing of two transitions we obtain a marking with one token on W_R and one token on W_L, where no further firing is possible.

12.6 Conclusion

We have presented a static analysis technique for graph transformation systems which produces a finite structure, called Petri graph, combining hypergraphs and Petri nets, which approximates the graphs which are reachable in the original grammar. Such a structure can be used to check safety properties, like the absence of deadlocks, in the original system.

An interesting question which has only been brushed in the paper, concerns the techniques which should be used to extract information from a computed Petri graph. It is certainly possible to reuse most of the well-established analysis techniques developed for Petri nets in the literature, such as coverability trees. However, as observed in the example, also the graphical structure underlying a Petri graph might play an essential role when establishing a property of the system. Since every graph reachable in the original grammar can be mapped to the approximated unfolding through a graph morphism, all properties which are reflected by graph morphisms can be checked on the approximated unfolding. We are currently investigating a syntactical characterization of such a class of properties. Other interesting issues are the use of methods from formal language theory (as hinted at in the example) and of model checking techniques.

Another question is the following: what can we do when we fail to prove a property? Obviously, it might still be the case that the considered property holds of the system, but this fact cannot be derived from the approximated unfolding where we have lost too much information by over-approximating. A partial solution could be to refine the description of the system, by computing a better approximation of the "complete" unfolding. This can be done by delaying folding steps and unfolding the Petri graph a bit further, "freezing" some parts of the approximated unfolding in order to avoid that a folding step leads to confuse them with other parts. A sequence of subsequently better approximations should converge to the whole, usually infinite, unfolding. In connection to this it would be interesting to determine which kind of systems can be "approximated" in an exact way—maybe by variations of the folding condition—one candidate being certainly Petri nets.

It is our aim to extend the proposed analysis technique to more general forms of graph rewriting, e.g., to the general double-pushout approach. In this case, since also edges might be preserved by a rewriting rule, the Petri net underlying a Petri graph cannot be simply an ordinary net, but it will be necessary to resort to contextual nets as in [Bal00].

Acknowledgements:

We are grateful to Javier Esparza for his insightful suggestions and to Alin Stefanescu and Stefan Schwoon who helped us with the finite-automaton tool used to compute the languages in Section 12.5. We are also grateful to anonymous referees for their valuable comments.

Chapter 13

Approximating the Behaviour of Graph Transformation Systems

(Joint work with Paolo Baldan)

Abstract

We propose a technique for the analysis of graph transformation systems based on the construction of finite structures approximating the behaviour of such systems with arbitrary accuracy. Following a classical approach, one can construct a chain of finite under-approximations (k-*truncations*) of the Winskel's style unfolding of a graph grammar. More interestingly, also a chain of finite over-approximations (k-*coverings*) of the unfolding can be constructed and both chains converge (in a categorical sense) to the full unfolding. The finite over- and under-approximations can be used to check properties of a graph transformation system, like safety and liveness properties, expressed in (meaningful fragments of) the modal μ-calculus. This is done by embedding our approach in the general framework of abstract interpretation.

13.1 Introduction

Graph transformation systems (GTSs) [Roz97] are a powerful specification formalism for concurrent and distributed systems [EKMR99], generalising Petri nets. Along the years their concurrent behaviour has been deeply studied and a consolidated theory of concurrency is now available [Roz97, EKMR99]. Although several semantics of Petri nets, like process and unfolding semantics, have been extended to GTSs (see, e.g., [CMR96, Rib96, BCM98, BCM99]), concerning automated verification, the literature does not contain many contributions to the static analysis of GTSs (see [Koc00, 1]).

Most of the mentioned semantics for GTSs define an operational model of computation, which gives a concrete description of the behaviour of the system in terms of non-effective (e.g., infinite, non-decidable) structures. In this paper, generalising the work in [1], we provide a technique for constructing finite approximations of the behaviour for a class of (hyper)graph transformation systems. We show how one can construct under- and over-approximations of the behaviour of the system. The "accuracy" of such approximations can be fixed and arbitrarily increased in a way that the corresponding chain of (both under- and over-) approximations converges to the exact behaviour.

We concentrate on the unfolding semantics of GTSs, one reason for referring to a concurrent semantics being the fact that it allows to avoid to check all the interleavings of concurrent events. The unfolding construction for GTSs produces a static structure which fully describes the concurrent behaviour of the system, including all possible rewriting steps and their mutual dependencies, as well as all reachable states [Rib96, BCM99]. However, as already mentioned, the unfolding, being infinite for any non-trivial system, cannot be used directly for verification purposes. Given a *graph grammar*, i.e., a GTS with a start hypergraph, we show how to construct finite structures which can be seen as approximations of the full unfolding of the grammar, at a chosen level k of accuracy.

Under-approximations (k-truncations). The unfolding of a graph grammar $\mathcal{G}$ can be defined categorically as the colimit of its prefixes of finite causal depth. Hence "under-approximations" of the behaviour of $\mathcal{G}$ can be trivially produced by stopping the construction of the unfolding at a finite causal depth k, thus obtaining the so-called *k-truncation* $\mathcal{T}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$. In the case of Petri nets this is at the basis of the *finite prefix approach*: if the system is finite-state and if the stop condition is suitably chosen, the prefix turns out to be *complete*, i.e., it contains the same information as the full unfolding [McM93, Esp94]. In general, for infinite-state systems, any truncation of the unfolding will be just an under-approximation of the behaviour of the system, in the sense that any computation in the truncation can be really performed in the original system, but not vice versa. Nevertheless, finite truncations can still be used to check interesting properties of the grammar, e.g., some liveness properties of the form "eventually A" for a predicate A (see Section 13.5).

Over-approximations (k-coverings). A more challenging issue is to provide (sensible) over-approximations of the behaviour of a grammar $\mathcal{G}$, i.e., finite approximations of the unfolding which "represent" all computations of the original system (but possibly more). To this aim, generalising [1], we propose an algorithm which, given a graph grammar $\mathcal{G}$, produces a finite structure, called *Petri graph*, consisting of a hypergraph and of a P/T net (possibly not safe or cyclic) over it, which can be seen as an over-approximation of the unfolding. Differently from [1], one can require the approximation to be exact up to a certain causal depth k, thus obtaining the so-called *k-covering* $\mathcal{C}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$.

The covering $\mathcal{C}^k(\mathcal{G})$ over-approximates the behaviour of $\mathcal{G}$ in the sense that every computation in $\mathcal{G}$ is mapped to a valid computation in $\mathcal{C}^k(\mathcal{G})$. Moreover every hypergraph reachable from the start graph can be mapped homomorphically to (the graphical component of) $\mathcal{C}^k(\mathcal{G})$ and its image is reachable in the

Petri graph. Therefore, given a property over graphs reflected by graph morphisms, if it holds for all graphs reachable in the covering $\mathcal{C}^k(\mathcal{G})$ then it also holds for all reachable graphs in $\mathcal{G}$. Important properties of this kind are the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). Temporal properties, such as several safety properties of the form "always A", can be proven directly on the Petri net component of the coverings (see Section 13.5).

The fact that the unfolding can be approximated with arbitrary high accuracy is formalised by proving that both under- and over-approximations of the unfolding, converge to the full (exact) unfolding. In categorical terms, the unfolding $\mathcal{U}(\mathcal{G})$ of a graph grammar $\mathcal{G}$ can be expressed both as the colimit of the chain of k-truncations $\mathcal{T}^k(\mathcal{G})$ and as the limit of the chain of k-coverings $\mathcal{C}^k(\mathcal{G})$:

$$
\begin{tikzcd}
\mathcal{T}^0(\mathcal{G}) \arrow[r] \arrow[drr] & \mathcal{T}^1(\mathcal{G}) \arrow[r, dotted, no head] \arrow[dr] & \mathcal{T}^k(\mathcal{G}) \arrow[r] \arrow[d] & \mathcal{T}^{k+1}(\mathcal{G}) \arrow[dl] & \cdots \\
 & & \mathcal{U}(\mathcal{G}) \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & & \\
\mathcal{C}^0(\mathcal{G}) & \mathcal{C}^1(\mathcal{G}) \arrow[l] \arrow[r, dotted, no head] & \mathcal{C}^k(\mathcal{G}) & \mathcal{C}^{k+1}(\mathcal{G}) \arrow[l] & \cdots
\end{tikzcd}
$$

The idea that finite under- and over-approximations can be used for checking properties of a graph grammar $\mathcal{G}$ is enforced by identifying significant fragments of the μ-calculus for which the validity of a formula in some approximation implies the validity of the same formula in the original grammar. Nicely, this is done by viewing our approach as a special case of the general paradigm of abstract interpretation.

13.2 Hypergraph rewriting, Petri nets and Petri graphs

In this section we first introduce the class of (hyper)graph transformation systems considered in the paper. Then, after recalling some basic notions for Petri nets, we will define Petri graphs, the structure combining hypergraphs and Petri nets, which will be used to represent and approximate the behaviour of GTSs.

13.2.1 Graph transformation systems

Given a set A we denote by A^* the set of finite strings of elements of A. For $u \in A^*$ we write $|u|$ for the length of u. Moreover, if $f : A \to B$ is a function then $f^* : A^* \to B^*$ denotes its extension to strings. Throughout the paper Λ denotes a fixed set of *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$.

Definition 13.2.1 (hypergraph) *A* (Λ-)hypergraph G *is a tuple* (V_G, E_G, c_G, l_G), *where* V_G *is a set of nodes,* E_G *is a set of edges,* $c_G : E_G \to {V_G}^*$ *is a connection function and* $l_G : E_G \to \Lambda$ *is the labelling function for edges* *satisfying* $ar(l_G(e)) = |c_G(e)|$ *for every* $e \in E_G$*. Nodes are not labelled. A node* $v \in V_G$ *is called* isolated *if it is not connected to any edge.*

We use rules as in the double-pushout approach [Ehr79], with some restrictions.

Definition 13.2.2 (rewriting rule) *A graph rewriting rule is a span of injective hypergraph morphisms* $r = (L \overset{\varphi_L}{\hookleftarrow} K \overset{\varphi_R}{\hookrightarrow} R)$*, where* L*,* K*,* R *are finite hypergraphs. The rule is called* simple *if (i)* K *is discrete, i.e. it contains no edges, (ii) no two edges in the left-hand side* L *have the same label, (iii) the morphism* φ_L *is bijective on nodes, (iv)* V_L *does not contain isolated nodes.*

Hereafter we will restrict to simple rules. A simple rule can delete and produce but not preserve edges, while nodes cannot be deleted (conditions (i) and (iii)). Moreover, it cannot consume two edges with the same label and its left-hand side must be connected (conditions (ii) and (iv)). These restrictions are mainly aimed at simplifying the presentation. Only (iii), which allows to apply a rule without checking the dangling condition, could require serious technical complications to be removed (but observe that deletion of nodes can be simulated considering graphs up to isolated nodes and leaving a node isolated instead of deleting it).

To simplify the notation, in the following we will assume that for any rule $r = (L \overset{\varphi_L}{\hookleftarrow} K \overset{\varphi_R}{\hookrightarrow} R)$, the morphisms φ_L and φ_R are (set-theoretical) inclusions and that $K = L \cap R$ (componentwise). Furthermore the components of a rule r will be denoted by L_r, K_r and R_r.

Definition 13.2.3 (hypergraph rewriting) *Let* r *be a rewriting rule. A* match *of* r *in a hypergraph* G *is any morphism* $\varphi : L_r \to G$*. In this case we write* $G \Rightarrow_{r,\varphi} H$ *or simply* $G \Rightarrow_r H$*, if there exists a double-pushout diagram*

$$\begin{tikzcd} L_r \arrow[d, "\varphi"'] & K_r \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_r \arrow[d] \\ G & D \arrow[l, hook'] \arrow[r, hook] & H \end{tikzcd}$$

Given a graph transformation system (GTS)*, i.e., a finite set of rules* $\mathcal{R}$*, we write* $G \Rightarrow_{\mathcal{R}} H$ *if* $G \Rightarrow_r H$ *for some* $r \in \mathcal{R}$*. Moreover* $\Rightarrow_{\mathcal{R}}^*$ *denotes the transitive closure of* $\Rightarrow_{\mathcal{R}}$*. A* GTS *with a (finite) start graph* $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ *is called a* graph grammar*.*

13.2.2 Petri nets

We fix some basic notation for Petri nets [Rei85, MM90]. Given a set A we will denote by $A^{\oplus}$ the free commutative monoid over A (*multisets* over A). Given a function $f : A \to B$, by $f^{\oplus} : A^{\oplus} \to B^{\oplus}$ we denote its monoidal extension.

Definition 13.2.4 (Petri net) *Let* A *be a finite set of action labels. An* A*-labelled* Petri net *is a tuple* $N = (S, T, {}^{\bullet}(), ()^{\bullet}, p)$ *where* S *is a set of places,* T *is a set of transitions,* ${}^{\bullet}(), ()^{\bullet} : T \to S^{\oplus}$ *assign to each transition its pre-set and post-set and* $p : T \to A$ *assigns an action label to each transition.*

The Petri net is called irredundant *if there are no distinct transitions with the same label and pre-set, i.e., if for any* $t, t' \in T$

$$p(t) = p(t') \ \wedge \ {}^\bullet t = {}^\bullet t' \quad \Rightarrow \quad t = t'. \tag{13.1}$$

A marked Petri net *is a pair* (N, m_N)*, where* N *is a Petri net and* $m_N \in S^\oplus$ *is the initial marking.*

The irredundancy condition (13.1) aims at avoiding the presence of multiple events, indistinguishable for what regards the behaviour of the system. Hereafter *all* the considered Petri nets will be assumed irredundant, unless stated otherwise.

Definition 13.2.5 (causality relation) *Let* N *be a (marked) Petri net. The* causality relation $<_N$ *over* N *is the least transitive relation such that, for any* $t \in T$, $s \in S$*, we have (i)* $s <_N t$ *if* $s \in {}^\bullet t$ *and (ii)* $t <_N s$ *if* $s \in t^\bullet$.

13.2.3 Petri graphs

Petri graphs, as introduced in [1], are structures consisting of a hypergraph and of a Petri net whose places are the edges of the graph.

Definition 13.2.6 (Petri graph) *Let* $\mathcal{R}$ *be a* GTS*. A Petri graph (over* $\mathcal{R}$*) is a tuple* $P = (G, N, \mu)$ *where* G *is a hypergraph,* $N = (E_G, T_N, {}^\bullet(), ()^\bullet, p_N)$ *is an* $\mathcal{R}$*-labelled Petri net with edges of* G *as places, and* μ *associates to each transition* $t \in T_N$*, with* $p_N(t) = r$*, a hypergraph morphism* $\mu(t) : L_r \cup R_r \to G$ *such that*

$${}^\bullet t = \mu(t)^\oplus(E_{L_r}) \ \wedge \ t^\bullet = \mu(t)^\oplus(E_{R_r}) \tag{13.2}$$

A Petri graph for a grammar $(\mathcal{R}, G_\mathcal{R})$ *is a pair* (P, ι) *where* $P = (G, N, \mu)$ *is a Petri graph for* $\mathcal{R}$ *and* $\iota : G_\mathcal{R} \to G$ *is a graph morphism. The multiset* $\iota^\oplus(E_{G_\mathcal{R}})$ *is called* initial marking *of the Petri graph. A marking* $m \in {E_G}^\oplus$ *is called* reachable (coverable) *in* (P, ι) *if it is reachable (coverable) in the underlying Petri net.*

Condition (13.2) allow to interpret transitions in the net as "occurrences" of rules in $\mathcal{R}$. More precisely, if $p_N(t) = r$ and $\mu(t) : L_r \cup R_r \to G$ is the morphism associated to the transition, then $\mu(t)_{|L} : L_r \to G$ must be a match of r in G such that the image of the edges of L_r in G coincides with the pre-set of t. Then, the graph items resulting from the application of r must be already in G, and the corresponding edges must coincide with the post-set of t. This is formalised by the condition over the image through $\mu(t)$ of the edges of R_r. For an example see Section 13.5, where Fig. 13.2 presents two Petri graphs for the GTS in Fig. 13.1.

A safe marking m of a Petri graph $P = (G, N, \mu)$ is intended to represent the subgraph of G consisting of the edges in m and of the nodes attached to these edges. For a general non-safe marking edges with k tokens will result in k "parallel" edges. This is formalised in the next definition.

Definition 13.2.7 *Let $P = (G, N, \mu)$ be a Petri graph. Given a hypergraph morphism $\varphi : G' \to G$ injective on nodes, we say that the marking $\varphi^{\oplus}(E_{G'})$* generates *the graph G'.*

In the following we will often confuse a marking of a Petri graph with its generated graph, and say, e.g., that a given graph is reachable in a Petri graph.

Every hypergraph G can be considered as a Petri graph $[G] = (G, N, \mu)$ for $\mathcal{R}$, by taking N as the net with $S_N = E_G$ and no transitions. Similarly, $G_{\mathcal{R}}$ can be seen as a Petri graph for $(\mathcal{R}, G_{\mathcal{R}})$ by taking as $\iota : G_{\mathcal{R}} \to G_{\mathcal{R}}$ the identity.

Definition 13.2.8 (category of Petri graphs) *A* Petri graph morphism *is a pair $\psi = (\varphi, \tau) : (G, N, \mu) \to (G', N', \mu')$ where*

- *$\varphi : G \to G'$ is a hypergraph morphism;*
- *$\tau : T_N \to T_{N'}$ is a mapping such that for every $t \in T_N$, ${}^\bullet\tau(t) = \varphi^{\oplus}({}^\bullet t)$ and $\tau(t)^\bullet = \varphi^{\oplus}(t^\bullet)$, and $p_{N'} \circ \tau = p_N$.*
- *for every $t \in T_N$, $\mu'(\tau(t)) = \varphi \circ \mu(t)$.*

The category of Petri graphs and Petri graph morphisms is denoted by PG*.*

It is possible to show that the category PG is finitely cocomplete, i.e. it contains all finite colimits. In particular we will later make use of pushouts and coequalizers to define unfolding and folding operations.

13.3 Unfolding and under-approximations

In this section we define the unfolding of a graph grammar. Following a common approach in the literature (see, e.g., [Rib96, Sas94]) the unfolding is defined as the limit (actually, the categorical colimit) of the chain of its finite prefixes, each of which can be seen as an *under-approximation* of the behaviour of the system.

The finite prefixes of the unfolding are constructed inductively beginning from the start graph of the grammar and performing, at each stage, all the possible basic unfolding steps, until the given causal depth is reached. A basic step roughly consists of the "partial" application of a rule to a match, which does not delete the left-hand side, but only records the new graph item generated in the rewriting process and the rule occurrence.

To formally define a basic step we need to fix some notation. Given a transition t and a rule r we will denote by $P(t, r)$ the Petri graph $(L_r \cup R_r, N, \mu)$ where $N = (E_{L_r \cup R_r}, \{t\}, {}^\bullet t = E_{L_r}, t^\bullet = E_{R_r}, p_N(t) = r)$ and $\mu(t) = id_{L_r \cup R_r}$. By $\emptyset$ we denote a function with an empty set as domain.

Definition 13.3.1 (unfolding operation) *Let $P = (G, N, \mu)$ be a Petri graph for a* GTS *$\mathcal{R}$. Let $r \in \mathcal{R}$ be a rule and let $\varphi : L_r \to G$ be a match of r in G. The* unfolding *of P with rule r at match φ, denoted by $\mathsf{unf}(P, r, \varphi)$, is the Petri graph obtained as pushout of $(\varphi, \emptyset) : [L_r] \to P$ and $(id_{L_r}, \emptyset) : [L_r] \to P(t, r)$.*

If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define $\mathsf{unf}((P, \iota), r, \varphi) = (P', \psi \circ \iota)$ *where* $(\psi, \tau) : P \to P'$ *is the* PG *morphism generated by the pushout.*

We need to define the depth of an item in a Petri graph. We start with a definition of depth over Petri nets. To deal with the presence of causal cycles it is convenient to define several depth functions, each one measuring the depth of an item up to a fixed level k. Consider the monoid $\mathbb{M}_k = (\{0, \dots, k\}, +)$, where for $m, n \in \{0, \dots, k\}$, $m + n$ is ordinary addition if $m + n \leq k$ and $m + n = k$ otherwise.

Definition 13.3.2 (depth in a Petri net) *Let N be a Petri net. We define a function $D : (S_N \cup T_N \to \mathbb{M}_k) \to (S_N \cup T_N \to \mathbb{M}_k)$ as follows:*

$$D(d)(x) = \max\{d(s) \mid s \in S_N \ \wedge \ s < x\} + 1.$$

Then the function $depth_k : S_N \cup T_N \to \mathbb{M}_k$, assigning depth information to every Petri net item is the least fixed point of D.

The function $depth_k$ assigns to each item x of a Petri net its causal depth, i.e., the length h of the maximal chain of causally related items leading from the initial marking to x, when $h \leq k$ and k otherwise. Note that an item x located in a causality cycle has always maximal depth, i.e., $depth_k(x) = k$ for any k.

The definition generalises to Petri graphs in a straightforward way: places become edges and the depth of a node v is defined as the maximal depth of rules r where v appears in $R_r \setminus L_r$ (intuitively, of rules which can "generate" node v).

Definition 13.3.3 (depth of items in a Petri graph) *Let (P, ι) be a Petri graph with $P = (G, N, \mu)$. For any k the function $depth_k : E_G \cup T_N \to \mathbb{M}_k$ is defined as in Definition 13.3.2. This function is extended to nodes by defining, for $v \in V_G$*

$$depth_k(v) = \max\{depth_k(t) \mid p_N(t) = r \ \wedge \ v \in \mu(t)(V_{R_r} \backslash V_{L_r})\}$$

The prefixes of the unfolding of a graph grammar up to a given causal depth k are defined by the following algorithm.

Definition 13.3.4 (k-truncation) *Let $k \in \mathbb{N}$ and let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar. The algorithm generates a sequence $(P_i, \iota_i)_{i \in \mathbb{N}}$ of Petri graphs.*

(Step 0) *Initialise $(P_0, \iota_0) = ([G_{\mathcal{R}}], id_{G_{\mathcal{R}}})$.*

(Step $i + 1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i.*

$\star$ *Unfolding: Find a rule r in $\mathcal{R}$ and a match $\varphi : L_r \to G_i$ such that*

- *$\varphi^{\oplus}(E_{L_r})$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^\bullet t = \varphi^{\oplus}(E_{L_r})$ and $p_{N_i}(t) = r$;*

- *for all $x \in \varphi(L_r)$ it holds that $depth_k(x) \neq k$.*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

If no unfolding step can be performed, the algorithm stops. The resulting Petri graph (P_i, ι_i) is called k-truncation *of the unfolding of $\mathcal{G}$ and denoted by $\mathcal{T}^k(\mathcal{G})$.*

It can be easily proven that the unfolding procedure described above is terminating and confluent, and thus that $\mathcal{T}^k(\mathcal{G})$ is well-defined. Furthermore, $\mathcal{T}^{k+1}(\mathcal{G})$ can be obtained from $\mathcal{T}^k(\mathcal{G})$ by performing only the unfolding steps which involve items of depth k. This gives a uniquely determined embedding $\lambda_k : \mathcal{T}^k(\mathcal{G}) \to \mathcal{T}^{k+1}(\mathcal{G})$ for any $k \in \mathbb{N}$. The diagram $\mathcal{T}^0(\mathcal{G}) \xrightarrow{\lambda_0} \ldots \mathcal{T}^k(\mathcal{G}) \xrightarrow{\lambda_k} \mathcal{T}^{k+1}(\mathcal{G}) \xrightarrow{\lambda_{k+1}} \ldots$ is called the *truncation tower.*

The next definition introduces the full unfolding of a graph grammar as colimit of its finite truncations (which can be shown to exist).

Definition 13.3.5 (unfolding as colimit of the k-truncations) *The* (full) unfolding $\mathcal{U}(\mathcal{G})$ *of a graph grammar $\mathcal{G}$ is the colimit of the truncation tower.*

The proposition below clarifies in which sense the unfolding represents the behaviour of the original grammar: any graph reachable in a graph grammar can be mapped *isomorphically* to a reachable subgraph of its unfolding, and, vice versa, any reachable subgraph of the unfolding is the isomorphic image of a reachable graph in the original grammar. Furthermore steps in the original grammar correspond to steps in the unfolding [Rib96, BCM99].

Proposition 13.3.6 *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and let $\mathcal{U}(\mathcal{G}) = (U, N, \mu)$ be its unfolding. Then for every graph G we have $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ iff there exists an injective morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{U}(\mathcal{G})$. Furthermore, in the situation above if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{U}(\mathcal{G})$. And if ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} m$ for some marking m, then there exists a graph G' such that $G \Rightarrow_{\mathcal{R}} G'$ and $m = {\varphi_{G'}}^{\oplus}(E_{G'})$.*

Clearly, k-truncations provide, in general, only under-approximations of the behaviour of the original grammar $\mathcal{G}$, i.e., only one implication of Proposition 13.3.6 holds: any graph reachable in $\mathcal{T}^k(\mathcal{G})$ is mapped isomorphically to a graph reachable in $\mathcal{G}$ and any valid computation in $\mathcal{T}^k(\mathcal{G})$ corresponds to a valid derivation sequence in $\mathcal{G}$, but, in general, not vice versa. Still, as we will see in Section 13.5, k-truncations can be useful for proving properties of the original grammar.

13.4 Folding and over-approximations

In this section we define an algorithm which, given a graph grammar $\mathcal{G}$ and a level of accuracy k, produces a finite Petri graph $\mathcal{C}^k(\mathcal{G})$, called *$k$-covering*, which can be seen as an over-approximation of the behaviour of the grammar $\mathcal{G}$.

We have already mentioned that the full unfolding is usually infinite, also for finite-state systems. To obtain a finite over-approximation we modify the unfolding procedure by considering, besides the *unfolding rule*, also a *folding rule* which allows us to "merge" two occurrences of the left-hand side of a rule whenever they are, in a sense made precise later, one causally dependent on the other. Intuitively, the presence of such two occurrences of a left-hand side reveals a cyclic behaviour and applying the folding rule one avoids to unfold the corresponding infinite path. While guaranteeing finiteness, the folding operation causes a loss of information in a way that the resulting structure over-approximates the behaviour of the original system: every graph reachable in the original grammar $\mathcal{G}$ corresponds to a marking which is reachable in the covering and every valid derivation in $\mathcal{G}$ corresponds to a valid firing sequence in the covering (but not vice versa).

In order to compute better over-approximations of the behaviour the idea is to delay folding steps, constraining the algorithm to apply only unfolding steps until a given causal depth is reached. Roughly, this is obtained by "freezing" an initial part of the approximated unfolding, up to a given causal depth k, and by allowing only unfolding and no folding steps to affect that part. The resulting over-approximation $\mathcal{C}^k(\mathcal{G})$ is "exact" up to causal depth k, in the sense that any graph reachable in $\mathcal{G}$ in less than k steps will have a reachable *isomorphic* image in $\mathcal{C}^k(\mathcal{G})$. Instead, graphs which are reachable in a larger number of steps, in general, will be mapped homomorphically in $\mathcal{C}^k(\mathcal{G})$ (still to a reachable graph).

In this way one can obtain arbitrarily accurate approximations, a fact which is enforced by proving that the chain of k-coverings of a grammar $\mathcal{G}$ converges to the full (possibly infinite) unfolding $\mathcal{U}(\mathcal{G})$. In categorical terms, $\mathcal{U}(\mathcal{G})$ turns out to be the limit of the chain of coverings in a suitable subcategory of Petri graphs.

13.4.1 Computing k-coverings

A basic definition needed to introduce k-coverings is that of a folding operation. Intuitively, it allows to merge two matches of the same rule in a Petri graph.

Definition 13.4.1 (folding operation) *Let $P = (G, N, \mu)$ be a Petri graph for a* GTS *$\mathcal{R}$. Let $r \in \mathcal{R}$ be a rule and let $\varphi', \varphi : L_r \to G$ be matches of r in G. The* folding *of P at the matches φ', φ, denoted $\mathsf{fold}(P, r, \varphi', \varphi) = P'$, is the Petri graph P' obtained as the coequalizer of $(\varphi, \emptyset), (\varphi', \emptyset) : [L_r] \to P$ in category* PG.

If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define $\mathsf{fold}((P, \iota), r, \varphi', \varphi) = (P', \psi \circ \iota)$ where $(\psi, \tau) : P \to P'$ is the PG *morphism generated by the coequalizer.*

The algorithm which produces the k-covering $\mathcal{C}^k(\mathcal{G})$ generates a sequence of Petri graphs, beginning from the start graph of $\mathcal{G}$ and applying, non-deterministically, at each step, a folding or unfolding operation, until none of such steps is admitted. Folding steps will be applied only at depth k or greater. Note that as soon as folding steps are applied, the Petri graph will contain cycles.

Definition 13.4.2 (k-covering) *Let $\mathcal{G} = (\mathcal{R}, G_\mathcal{R})$ be a graph grammar and let $k \in \mathbb{N}$. The algorithm generates a sequence $(P_i, \iota_i)_{i \in \mathbb{N}}$ of Petri graphs, as follows.*

(Step 0) *Initialise $(P_0, \iota_0) = ([G_\mathcal{R}], id_{G_\mathcal{R}})$.*

(Step $i+1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i. Choose non-deterministically one of the following actions*

$\star$ Folding: *Find a rule r in $\mathcal{R}$ and two different matches $\varphi', \varphi : L_r \to G_i$ of r such that*

- *$\varphi^\oplus(E_{L_r})$ is a coverable marking in P_i;*
- *there exists a transition $t \in T_{N_i}$ such that*

$$p_{N_i}(t) = r \;\wedge\; {}^\bullet t = \varphi'^\oplus(E_{L_r}) \;\wedge\; \forall e \in \varphi^\oplus(E_{L_r}) : (e \in {}^\bullet t \;\vee\; t <_{N_i} e) \quad (13.3)$$

- *for every edge or node $x \in E_{L_r} \cup V_{L_r}$ it holds that*

$$\varphi(x) = \varphi'(x) \;\;\vee\;\; depth_k(\varphi(x)) = depth_k(\varphi'(x)) = k. \quad (13.4)$$

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{fold}((P_i, \iota_i), r, \varphi', \varphi)$.

$\star$ Unfolding: *Find a rule r in $\mathcal{R}$ and a match $\varphi : L_r \to G_i$ such that*

- *$\varphi^\oplus(E_{L_r})$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^\bullet t = \varphi^\oplus(E_{L_r})$ and $p_{N_i}(t) = r$;*
- *there is no other match $\varphi' : L_r \to G_i$ satisfying the folding condition.*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

*If no folding or unfolding step can be performed, the algorithm terminates. The resulting Petri graph (P_i, ι_i) is called k-*covering *of the unfolding of $\mathcal{G}$ and denoted by $\mathcal{C}^k(\mathcal{G})$.*

Condition (13.3) basically states that we can fold two matches of a rule r whenever the first one has been already unfolded producing a transition t, and the second match depends on the first one, in the sense that any edge in the second match is already in the first one or causally depends on t. Roughly, the idea is that we should not unfold a left-hand side again, if we have already done the same unfolding step in its past, since this might lead to infinitely many steps. There are some similarities, to be further investigated, with the work in [Gen98] where the sets of descendants and of normal forms of term rewriting systems are approximated by constructing an approximation automaton. Additionally, by Condition (13.4) only items of depth k can be merged, in a way that the prefix up to depth k of the unfolding is not involved in any folding operations. Actually some items of depth less than k can be part of a folding operation, but they must be left unchanged by the step.

13.4.2 Correctness, termination and confluence

We first show that the computed Petri graph $\mathcal{C}^k(\mathcal{G})$ gives an over-approximation of the behaviour of the given graph grammar, exact up to causal depth k. More precisely we prove that for any graph reachable in $\mathcal{G}$, there is a morphism into the covering $\mathcal{C}^k(\mathcal{G})$ such that the image of its edge set corresponds to a reachable marking. Furthermore, if a graph is reachable in $\mathcal{G}$ in less than k steps, then it will be mapped isomorphically to to (the graphical component) of $\mathcal{C}^k(\mathcal{G})$.

Proposition 13.4.3 (correctness) *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and assume that the algorithm computing the k-covering terminates producing the Petri graph $\mathcal{C}^k(\mathcal{G}) = ((U, N, \mu), \iota)$. Then for every graph G*

i) if $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ there exists a morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{C}^k(\mathcal{G})$. Furthermore, if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{C}^k(\mathcal{G})$.

ii) If $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ with a (possibly parallel) derivation of length less than k then there exists an injective morphism $\varphi_G : G \to U$ such that the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{C}^k(\mathcal{G})$ and $\max\{depth_k(x)) \mid x \in G\} < k$, and vice versa. Furthermore if ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} m$ for some transition t, then there exists a graph G' such that $G \Rightarrow_{\mathcal{R}} G'$ and $m = {\varphi_{G'}}^{\oplus}(E_{G'})$.

It is not obvious at first glance that the algorithm computing the k-covering always terminates. To prove termination we rely on the corresponding result in [1] where we show that it is not possible to perform infinitely many unfolding steps, without having the folding condition satisfied at some stage.

Proposition 13.4.4 (termination) *The algorithm computing the k-covering (see Definition 13.4.2) terminates for every graph grammar $\mathcal{G}$ and every $k \in \mathbb{N}$.*

In order to prove that the algorithm produces a uniquely determined result, independently of the order in which folding and unfolding steps are applied, we can show that the rewriting relation on Petri graphs induced by folding and unfolding steps is locally confluent. By the Diamond Lemma [DJ90], for a rewriting system local confluence and termination imply confluence.

Proposition 13.4.5 (confluence) *For any input grammar $\mathcal{G}$ and $k \in \mathbb{N}$ the algorithm computing the k-covering terminates with a result $\mathcal{C}^k(\mathcal{G})$ unique up to isomorphism.*

13.4.3 Full unfolding as limit of the coverings

The fact that folding and unfolding operations are given in terms of colimits allows us to define, for any k, a (uniquely determined) Petri graph morphism $\upsilon_k : \mathcal{C}^{k+1}(\mathcal{G}) \to \mathcal{C}^k(\mathcal{G})$.

The diagram $\mathcal{C}^0(\mathcal{G}) \stackrel{\upsilon_0}{\leftarrow} \ldots \mathcal{C}^k(\mathcal{G}) \stackrel{\upsilon_k}{\leftarrow} \mathcal{C}^{k+1}(\mathcal{G}) \stackrel{\upsilon_{k+1}}{\leftarrow} \ldots$ is called the *covering tower*.

The next proposition presents a central result of this paper. For technical reasons we consider the full subcategory PG^* of PG having as objects Petri graphs in which every edge is coverable and every transition can be fired.

Proposition 13.4.6 (unfolding as limit of the coverings) *The limit in the category* PG^* *of the covering tower* $\mathcal{C}^0(\mathcal{G}) \overset{v_0}{\leftarrow} \ldots \mathcal{C}^k(\mathcal{G}) \overset{v_k}{\leftarrow} \mathcal{C}^{k+1}(\mathcal{G}) \overset{v_{k+1}}{\leftarrow} \ldots$ *is the full unfolding* $\mathcal{U}(\mathcal{G})$ *of the graph grammar.*

13.5 Checking Temporal Properties

In this section we illustrate how our technique can be seen as a specific instance of abstract interpretation [Cou96, JN95]. Embedding our work into this context we can resort to some results from [LGS+95], thus identifying classes of temporal properties (μ-calculus formulae) which, being preserved/reflected by abstractions, can be studied over suitable approximations of a GTS.

We recall some concepts from [LGS+95], the more basic one being the formalisation of abstraction given in terms of Galois connections (over powerset lattices).

Definition 13.5.1 (Galois connection) *Let* Q_1 *and* Q_2 *be two sets of states. A* Galois connection *from* $\mathcal{P}(Q_1)$ *to* $\mathcal{P}(Q_2)$ *is a pair of monotonic functions* (α, γ)*, with* $\alpha : \mathcal{P}(Q_1) \to \mathcal{P}(Q_2)$ (abstraction) *and* $\gamma : \mathcal{P}(Q_2) \to \mathcal{P}(Q_1)$ (concretization)*, such that* $id_{Q_1} \subseteq \gamma \circ \alpha$ *and* $\alpha \circ \gamma \subseteq id_{Q_2}$.

Next we introduce $\langle \alpha, \gamma \rangle$-simulations which turn out to coincide with simulations in the sense of Milner (see [LGS+95] for details).

Definition 13.5.2 ($\langle \alpha, \gamma \rangle$-simulation) *Let* $T_i = (Q_i, \to_i)$ *with* $i \in \{1, 2\}$ *be transition systems, where* Q_i *is a set of* states *and* $\to_i \subseteq Q_i \times Q_i$ *is the* transition relation*. Let furthermore* (α, γ) *be a Galois connection from* $\mathcal{P}(Q_1)$ *to* $\mathcal{P}(Q_2)$*.*

We say that T_2 $\langle \alpha, \gamma \rangle$-simulates T_2*, written* $T_1 \sqsubseteq_{\langle \alpha, \gamma \rangle} T_2$*, if* $\alpha \circ pre[\to_1] \circ \gamma \subseteq pre[\to_2]$*, where the function* $pre[\to_i] : \mathcal{P}(Q_i) \to \mathcal{P}(Q_i)$ *is defined by* $pre[\to_i](Q) = \{q \in Q_i \mid \exists\, q' \in Q : q \to_i q'\}$*.*

Let T_1, T_2 be transition systems and let $\varphi : T_1 \to T_2$ be a *transition system morphism*, i.e., a function $\varphi : Q_1 \to Q_2$ such that such that for any $q, q' \in Q_1$ if $q \to_1 q'$ then $\varphi(q) \to_2 \varphi(q')$ (in other words, φ is a special kind of simulation). Then, it can be easily seen that the pair (φ, φ^{-1}) is a Galois connection and furthermore $T_1 \sqsubseteq_{\langle \varphi, \varphi^{-1} \rangle} T_2$.

We next discuss how our under- and over-approximations of the behaviour of a graph grammar can be interpreted in this context. First observe that, a Petri graph (P, ι), with $P = (G, N, \mu)$, can be associated with a transition system $\mathbb{M}_{(P,\iota)}$, having reachable markings (multi-sets of edges) as states and the firing relation of the underlying Petri net N as transition relation. Alternatively we can consider the transition system, $\mathbb{G}_{(P,\iota)}$, where states are graphs (generated by the reachable markings, in the sense of Definition 13.2.7) and the transition relation is again induced by the firing relation of N.

Let $\mathcal{G}$ be a graph grammar and consider the full unfolding $\mathcal{U}(\mathcal{G})$, the k-truncations $\mathcal{T}^k(\mathcal{G})$ and the k-coverings $\mathcal{C}^k(\mathcal{G})$. Since by Definition 13.3.5 and Proposition 13.4.6 the full unfolding is the colimit of the truncations and the limit of the coverings, we have (unique) morphisms $\eta_k : \mathcal{T}^k(\mathcal{G}) \to \mathcal{U}(\mathcal{G})$ and

$\theta_k : \mathcal{U}(\mathcal{G}) \to \mathcal{C}^k(\mathcal{G})$, which can be regarded as functions from sets of markings to sets of markings and furthermore they are morphisms between the transition systems of the underlying Petri nets. Hence we have the following result.

Proposition 13.5.3 *Let $\mathcal{G}$ be a graph grammar. Then (η_k, η_k^{-1}) and $(\theta_k, \theta_k^{-1})$ are Galois connections and $\mathbb{M}_{\mathcal{T}^k(\mathcal{G})} \sqsubseteq_{\langle \eta_k, \eta_k^{-1} \rangle} \mathbb{M}_{\mathcal{U}(\mathcal{G})} \sqsubseteq_{\langle \theta_k, \theta_k^{-1} \rangle} \mathbb{M}_{\mathcal{C}^k(\mathcal{G})}$.*

Modal μ-calculus. One of the central results of [LGS+95] is the preservation and reflection of modal μ-calculus formulae on transitions systems. Recall that the modal μ-calculus is a temporal logic enriched with fixed-point operators. The syntax of μ-calculus formulae is the following:

$$f ::= \ A \mid X \mid \Diamond f \mid \Box f \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \mu X.f \mid \nu X.f$$

where $A \in \mathcal{A}$ are *atomic propositions* and $X \in \mathcal{X}$ are *proposition variables.* The formulae are evaluated over a transition system $T = (Q, \to)$, with respect to an *interpretation* $I : \mathcal{A} \to \mathcal{P}(Q)$, associating to any atomic proposition $A \in \mathcal{A}$ the set of states $I(A)$ where it holds. A formula $\Diamond f$ / $\Box f$ holds in a state q if some / any single step leads to a state where f holds. The connectives $\neg$,$\vee$, $\wedge$ are interpreted in the usual way. The formulae $\mu X.f$ and $\nu X.f$ represent the *least* and *greatest fixed point*, respectively. We write $q \models^I f$ to mean that the (closed) formula f holds in the state q, under the interpretation I. We say that a transition system T *satisfies a (closed) formula f under an interpretation I*, written $T \models^I f$, if $q_0 \models^I f$ where q_0 is the initial state of T.

The fragment of the modal μ-calculus without negation and box operator is denoted by $\Diamond L_\mu$. By dropping negation and the diamond operator we obtain the fragment $\Box L_\mu$. Some typical *liveness* properties of the form "eventually A" (i.e., $\mu X.(A \vee \Diamond X)$) can be expressed in the fragment $\Diamond L_\mu$, whereas some typical *safety* properties of the form "always A" (i.e., $\nu X.(A \wedge \Box X)$) can be expressed in the fragment $\Box L_\mu$. However, while for linear time there exists a syntactic characterization of liveness and safety properties [Pra94], in the case of branching time there is not yet any established definition of liveness and safety [MT01].

Let us come back to graph transformation systems, where atomic propositions stand for graph properties, i.e., for sets of graphs. Let (P, ι) be a Petri graph and let f be a μ-calculus formula over a set of atomic propositions $\mathcal{A}$. Assume that I_m and I_g are interpretations of $\mathcal{A}$ over $\mathbb{M}_{(P,\iota)}$ and $\mathbb{G}_{(P,\iota)}$, respectively, such that, for any $A \in \mathcal{A}$, any marking m and graph $G(m)$ generated by m

$$m \in I_m(A) \quad \textit{iff} \quad G(m) \in I_g(A). \tag{13.5}$$

Then it is immediate to see that $\mathbb{M}_{(P,\iota)} \models^{I_m} f$ if and only if $\mathbb{G}_{(P,\iota)} \models^{I_g} f$. Furthermore, given a graph grammar $\mathcal{G}$, seen as a transition system in the obvious way, by Proposition 13.3.6 it follows that $\mathcal{G} \models^{I_g} f$ if and only if $\mathbb{G}_{\mathcal{U}(\mathcal{G})} \models^{I_m} f$.

Using the above observations and exploiting the preservation and reflection properties in [LGS+95] we can obtain the following result. We say that a set Gr of hypergraphs is *preserved* by graph morphisms whenever the existence of a morphism $\varphi : G \to G'$ with $G \in Gr$ implies $G' \in Gr$. Symmetrically, Gr is *reflected* by graph morphisms whenever the existence of a morphism $\varphi : G \to G'$ with $G' \in Gr$ implies $G \in Gr$.

Corollary 13.5.4 *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$, let f be a μ-calculus formula over a set of atomic propositions $\mathcal{A}$. Let I_m and I_g be interpretations satisfying (13.5). Then*

- *If $f \in \Diamond L_\mu$, $\mathbb{M}_{\mathcal{T}^k(\mathcal{G})} \models^{I_m} f$ and every set $I_g(A)$ is preserved by hypergraph morphisms, then $\mathcal{G} \models^{I_g} f$.*
- *If $f \in \Box L_\mu$, $\mathbb{M}_{\mathcal{C}^k(\mathcal{G})} \models^{I_m} f$ and every set $I_g(A)$ is reflected by hypergraph morphisms, then $\mathcal{G} \models^{I_g} f$.*

We have shown how to reduce the analysis of the full transition system of a graph grammar to the analysis of simpler transition systems, generated by Petri nets (underlying Petri graphs). These transition systems might still have infinitely many states, but there are several decidability results for the modal μ-calculus and other forms of temporal logics [Esp97, HRY91, Jan90].

Example. Let us consider the simple graph grammar $\mathcal{S}$ in Fig. 13.1, where edge labels have the following meaning: C (connections), S_{pub} (public servers), S_{prv} (private servers), P_{int} (internal processes) and P_{ext} (external processes). Internal processes can wander around the network and public servers can extend the network by creating new connections. Our aim is to show that the external process is never connected to a private server and thus has access to classified data. That is, we want to show that the following logical formula is satisfied by the graph transformation system: $f = \nu X.(A \ \wedge \ \Box X)$ where the atomic proposition A holds for all the graphs in $Gr_A = \{G \mid \forall e_1, e_2 \in E_G.(l_G(e_1) = S_{prv} \ \wedge \ l_G(e_2) = P_{ext} \ \Rightarrow \ c_G(e_1) \neq c_G(e_2))\}$ (it always holds that whenever a private server and an external process appear in a graph, then they are not connected to the same node). One can easily show that Gr_A is reflected by hypergraph morphisms.

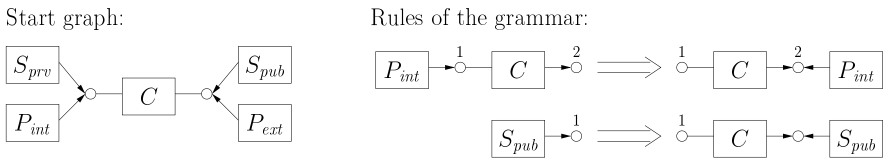


Figure 13.1: The example graph grammar $\mathcal{S}$.

Applying the algorithm in Definition 13.4.2 to the graph grammar $\mathcal{S}$ to compute the 0-covering $\mathcal{C}^0(\mathcal{S})$, we obtain the left-hand Petri graph in Fig. 13.2. Observe that the formula f is not satisfied by this covering, since A is invalid already for the initial marking. Hence this gives us no indication whether or not the formula holds for $\mathcal{S}$. Therefore we try and compute the 1-covering and get the Petri graph on the right-hand side of Fig. 13.2. Now we can establish that f holds just by looking at the graph structure of the 1-covering $\mathcal{C}^1(\mathcal{S})$: edges of the form S_{prv} and of the form P_{ext} do not share a common node.

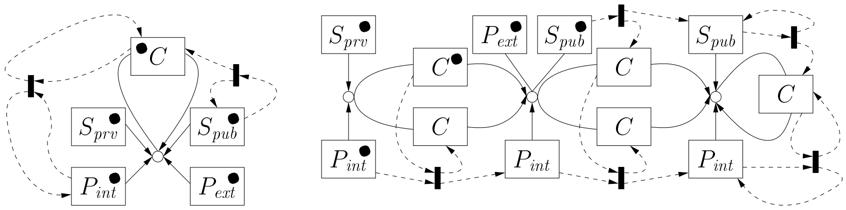


Figure 13.2: The 0-covering $\mathcal{C}^0(\mathcal{S})$ and the 1-covering $\mathcal{C}^1(\mathcal{S})$ of grammar $\mathcal{S}$ in Fig. 13.1.

It would also be possible to extend the example by adding rules that allow movement of external processes and verify the same property. However, in this case the 1-covering would get larger and harder to draw. In [BCK02] we have shown how to analyse a more complex GTS.

13.6 Conclusion

We have presented a technique for computing under- and over-approximations of the behaviour of graph transformation systems and we have identified suitable classes of properties of a GTS which can be inferred by analysing its approximations. We envision a scenario where a property of a given GTS can be checked by computing better and better approximations and verifying the property for each of them. Because of undecidability issues, this process might never terminate and it could also be costly from a complexity point of view, but with appropriate heuristics and fine-tuning of the technique, it is conceivable that several interesting properties for non-trivial GTSs can be verified in such a way.

In order to test the applicability of our theory we plan to implement the presented algorithm and to apply it to practical examples.

On the theoretical side, there are still several open problems. First, it would be interesting to classify logical formulae on graphs which are preserved and reflected by graph morphisms, via a kind of type system. This would enable us to extend the results of Section 13.5 to a logic in which one is able to reason specifically about graph transition systems (see also [Cou97]). Additionally it would be necessary to detail how the verification of these formulae on Petri graphs can be reduced to the existing model-checking techniques for Petri nets.

Another relevant issue is the extension of the developed theory to GTSs having more general forms of rules. Particularly promising, in order to decrease the size of the approximations, is the case of GTSs where rules might have a non-discrete left-hand side. This extension would require to resort to contextual nets in order to represent the Petri net structure underlying a Petri graph.

An open and, as it seems, highly non-trivial question is the treatment of finite-state GTSs. It would be quite interesting to understand if for a given GTS with only finitely many reachable graphs (up to isomorphism), there is a way to construct—using folding and unfolding steps—a finite Petri graph which gives an exact representation of the original GTS, without any proper approximation. This would allow to reduce the analysis of finite-state GTSs to that of Petri nets.

Acknowledgements:

The authors are grateful to Andrea Corradini for his insightful suggestions and to the anonymous referees for their comments on the submitted version of this paper.

Chapter 14

A Logic for Analyzing Abstractions of Graph Transformation Systems

(Joint work with Paolo Baldan and Bernhard König)

Abstract

A technique for approximating the behaviour of graph transformation systems (GTSs) by means of Petri net-like structures has been recently defined in the literature. In this paper we introduce a monadic second-order logic over graphs expressive enough to characterise typical graph properties, and we show how its formulae can be effectively verified. More specifically, we provide an encoding of such graph formulae into quantifier-free formulae over Petri net markings and we characterise, via a type assignment system, a subclass of formulae F such that the validity of F over a GTS $\mathcal{G}$ is implied by the validity of the encoding of F over the Petri net approximation of $\mathcal{G}$. This allows us to reuse existing verification techniques, originally developed for Petri nets, to model-check the logic, suitably enriched with temporal operators.

14.1 Introduction

Distributed and mobile systems can often be specified by graph transformation systems (GTSs) in a very natural way. However, work on static analysis and verification of GTSs is scarce. The fact that GTSs can be seen as a proper extension of Petri nets suggests the possibility of relying on techniques already developed in the literature for this related formalism. However, unlike Petri nets, graph transformation systems are usually Turing-complete so that many problems decidable for general P/T-nets become undecidable for GTSs.

A technique proposed in [1, 2] is based on the approximation of GTSs by means of Petri net-like structures in the spirit of abstract interpretation of reactive systems [LGS$^+$95]. More precisely, an approximated unfolding construction maps any given GTS $\mathcal{G}$ to a finite structure $\mathcal{U}(\mathcal{G})$, called *covering* (or approximated unfolding) of $\mathcal{G}$. The covering $\mathcal{U}(\mathcal{G})$ is a so-called *Petri graph*, i.e. a structure consisting of a Petri net with a graphical structure over places. It provides an *over-approximation* of the behaviour of $\mathcal{G}$, in the sense that any graph reachable in $\mathcal{G}$ can be mapped homomorphically to the graph underlying $\mathcal{U}(\mathcal{G})$ and its image is a reachable marking of $\mathcal{U}(\mathcal{G})$. (Note that, since $\mathcal{G}$ is possibly infinite-state, while $\mathcal{U}(\mathcal{G})$ is finite, it would not be possible to have in $\mathcal{U}(\mathcal{G})$ isomorphic images of all graphs reachable in $\mathcal{G}$.) Therefore, given a property over graphs reflected by graph morphisms, if it holds for all states reachable in the abstraction $\mathcal{U}(\mathcal{G})$ then it also holds for all reachable graphs in $\mathcal{G}$. In other words, if T is a temporal logic formula containing only universal quantifiers (e.g. a formula in ACTL* or in a suitable fragment of the modal μ-calculus) and where state predicates are reflected by graph morphisms, then the validity of T over the covering $\mathcal{U}(\mathcal{G})$ allows us to infer the validity of T for the original system [CGL99].

However, several relevant questions remain to be answered. First of all, which logic should we use to specify state predicates (i.e., graph properties)? How can we identify a subclass of such predicates which is reflected by graph morphisms and which can thus be safely checked over the approximation? And finally, given the approximation $\mathcal{U}(\mathcal{G})$, is there a way of encoding formulae expressing graph properties into "equivalent" formulae over Petri net markings?

As for the first point, we propose to describe state predicates, i.e., the graph properties of interest, by means of a monadic second-order logic $\mathcal{L}2$ on graphs, where quantification is allowed over (sets of) edges. (Similar logics are considered in [Cou97].) Relevant graph properties can be expressed in $\mathcal{L}2$, e.g., the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (related to security properties) or cycles (related to deadlock-freedom).

Regarding the second question, we introduce a type inference system characterising a subclass of formulae in the logic $\mathcal{L}2$ which are reflected by graph morphisms. Hence, given any formula F in such a class, if F can be proved for any reachable state of the approximation $\mathcal{U}(\mathcal{G})$ then we can deduce that F holds for any reachable graph of the original GTS $\mathcal{G}$.

Finally, given the approximation $\mathcal{U}(\mathcal{G})$, we define a constructive translation of graph formulae in $\mathcal{L}2$ into formulae over markings of the Petri net underlying the abstraction $\mathcal{U}(\mathcal{G})$. More precisely, any graph formula F is mapped to a formula $\hat{F}$ over markings such that a marking satisfies $\hat{F}$ if and only if the graph it represents satisfies F. Since the graph underlying $\mathcal{U}(\mathcal{G})$ is finite and fixed after computing the abstraction, we can perform quantifier elimination on graph formulae and, surprisingly, encode even monadic second-order logic formulae into propositional formulae on markings, containing only predicates of the form $\#s \leq c$ (the number of tokens in place s is smaller than or equal to c). We remark that the encoding for the first-order fragment of $\mathcal{L}2$ is simpler and can be defined inductively.

Altogether these results allow us to verify behavioural properties of a GTS by reusing existing model-checking techniques for Petri nets. In fact, given a formula T of a suitable temporal logic (e.g. a formula of ACTL* or of a fragment of the modal μ-calculus without $\diamond$ and negation), where state predicates are reflected by graph morphisms, then, by the construction mentioned above and using general results from abstract interpretation [LGS+95], T can be translated into a formula which can be checked over the Petri net underlying $\mathcal{U}(\mathcal{G})$. We recall that general temporal state-based logics over Petri nets, i.e., logics where basic predicates have the form $\#s \leq c$, are not decidable in general, but important fragments of such logics are [HRY91, HR89, Jan90].

For the sake of simplicity, although the approximation method of [1, 2] was originally designed for hypergraphs, in this paper we concentrate on directed graphs. The extension to general hypergraphs requires some changes to the graph logic $\mathcal{L}2$. This rises some technical difficulties which are, while not being insurmountable, a hindrance to the clear and easy presentation of our results.

In the rest of the paper we will first summarise the approximation technique for GTSs in [1], shortly mentioning some results from [2]. Then, we will define the monadic second-order logic $\mathcal{L}2$ over graphs and we will introduce the type system characterising a subclass of formulae in $\mathcal{L}2$ which are reflected by graph morphisms, and which can thus be checked on the covering. Finally we will show how to encode these formulae into quantifier-free state-based formulae on the markings of Petri nets, starting from the simpler case of first-order formulae.

14.2 Approximated Unfolding Construction

In this section we sketch the algorithm, introduced in [1], for the construction of a finite approximation of the unfolding of a graph transformation system. We first define graphs and structure-preserving morphisms on graphs. We will assume that Λ denotes a fixed and finite set of labels. Note that multiple edges between nodes are allowed.

Definition 14.2.1 (Graph, graph morphism) *A* graph $G = (V_G, E_G, s_G, t_G, l_G)$ *consists of a set V_G of nodes, a set E_G of edges, a source and a target function $s_G, t_G \colon E_G \to V_G$ and a function $l_G \colon E_G \to \Lambda$ labelling the edges.*

A graph morphism $\varphi \colon G_1 \to G_2$ *is a pair of mappings $\varphi_V \colon V_{G_1} \to V_{G_2}$ and $\varphi_E \colon E_{G_1} \to E_{G_2}$ such that $\varphi_V \circ s_{G_1} = s_{G_2} \circ \varphi_E$, $\varphi_V \circ t_{G_1} = t_{G_2} \circ \varphi_E$ and $l_{G_1} = l_{G_2} \circ \varphi_E$ for each edge $e \in E_{G_1}$. A morphism φ will be called* edge-bijective *if φ_E is a bijection. The subscripts in φ_E and φ_V will be usually omitted.*

We next define the notion of a graph transformation system and the corresponding rewriting relation.

Definition 14.2.2 (Graph transformation system) *A* graph transformation system (GTS) $(G_0, \mathcal{R})$ *consists of an initial graph G_0 and a set $\mathcal{R}$ of rewriting rules of the form $r = (L, R, \alpha)$, where L, R are graphs, called* left-hand side *and* right-hand side, *respectively, and $\alpha \colon V_L \to V_R$ is an injective function.*

A match *of a rewriting rule r in a graph G is a morphism $\varphi \colon L \to G$ which is injective on edges. We can apply r to a match in G obtaining a new graph H, written $G \stackrel{r}{\Rightarrow} H$. The target graph H is defined as follows*

$$V_H = V_G \uplus (V_R - \alpha(V_L)) \qquad E_H = (E_G - \varphi(E_L)) \uplus E_R$$

and, defining $\overline{\varphi} : V_R \to V_H$ *by* $\overline{\varphi}(\alpha(v)) = \varphi(v)$ *if* $v \in V_L$ *and* $\overline{\varphi}(v) = v$ *otherwise, the source, target and labelling functions are given by*

$$\begin{array}{rcl} e \in E_G - \varphi(E_L) & \Rightarrow & s_H(e) = s_G(e), \quad t_H(e) = t_G(e), \quad l_H(e) = l_G(e) \\ e \in E_R & \Rightarrow & s_H(e) = \overline{\varphi}(s_R(e)), \quad t_H(e) = \overline{\varphi}(t_R(e)), \quad l_H(e) = l_R(e) \end{array}$$

Intuitively, the application of r to G at the match φ first removes from G the image of the edges of L. Then the graph G is extended by adding the new nodes in R (i.e., the nodes in $V_R - \alpha(V_L)$) and the edges of R. Observe that the (images of) the nodes in L are preserved, i.e., not affected by the rewriting step.

Example 14.2.3 Consider a system where processes compete for resources R_1 and R_2. A process needs both resources in order to perform some task. The system is represented as a GTS Sys as follows. We consider edges labelled by R_1, R_2, R_1^f, R_2^f standing for assigned and free resources, respectively, and P_1, P_2 and P_3 denoting a process waiting for resource R_1, a process waiting for resource R_2 and a process holding both resources, respectively. Furthermore, edges labelled by D_1 and D_2 connect the target node of a process and the source node of a resource when the process is asking for the resource. When the target node of a resource coincides with the source node of a process, this means that the resource is assigned to the process. The initial scenario for Sys is represented in Fig. 14.1, with a single process P_1 asking for both resources.

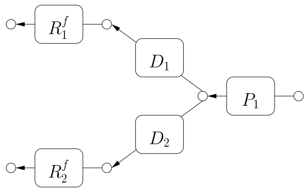


Figure 14.1: Start graph of Sys with a process and resources.

The rewriting rules of Sys are defined with the aim of avoiding deadlocks in the form of vicious cycles. There are three kind of rules, depicted in Fig. 14.2: (1) a process P_i can acquire a free resource R_j^f whenever $i = j$ and become P_{i+1}, (2) P_3 can release its resources and (3) processes of the form P_1 can fork creating more processes of the same kind with demand for the same resources. The natural numbers $1, 2, 3, \ldots$ which decorate nodes in the left-hand side and right-hand side of rules implicitly represent the mapping α.

Observe that an additional rule, analogous to rule 1, but with $i = 1$ and $j = 2$, would possibly lead to a vicious cycle with circular demand for resources, in two steps (see Fig. 14.3).

Some basic notation concerning multisets is needed to deal with Petri nets. Given a set A we will denote by $A^{\oplus}$ the free commutative monoid over A, whose elements will be called *multisets* over A. In the sequel we will sometime identify $A^{\oplus}$ with the set of functions $m \colon A \to \mathbb{N}$ such that the set $\{a \in A \mid m(a) \neq 0\}$ is

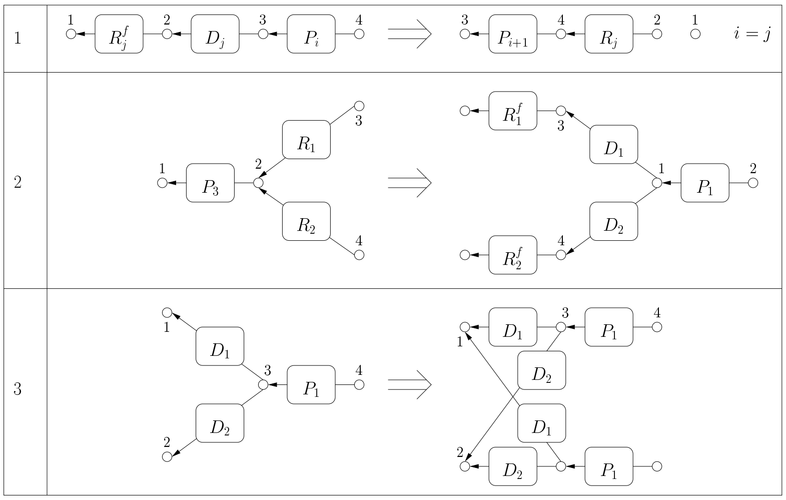


Figure 14.2: Rewriting rules of the GTS Sys.

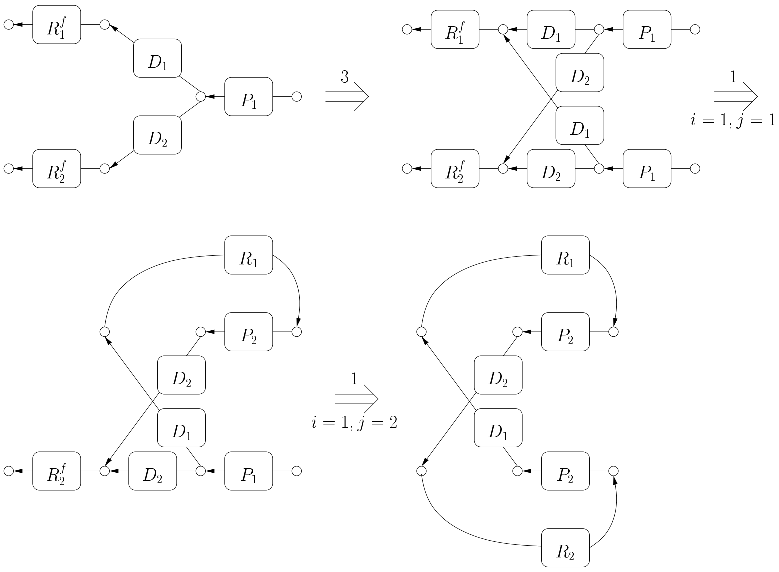


Figure 14.3: Vicious cycle representing a deadlock.

finite. E.g., in particular, $m(a)$ denotes the multiplicity of an element a in the multiset m. Sometimes a multiset will be also identified with the underlying set, writing, e.g., $a \in m$ for $m(a) \neq 0$. Given a function $f\colon A \to B$, by $f^{\oplus}\colon A^{\oplus} \to B^{\oplus}$ we denote its monoidal extension, i.e., $f^{\oplus}(m)(b) = \sum_{f(a)=b} m(a)$ for every $b \in B$.

In order to approximate graph transformation systems we use Petri graphs, introduced in [1], which are basically Petri nets, specifying the operational behaviour, with added graph structure.

Definition 14.2.4 (Petri graphs) *Let $\mathcal{G} = (G_0, \mathcal{R})$ be a GTS. A* Petri graph *P* (over $\mathcal{G}$) *is a tuple (G, N, m_0) where*

- *G is a graph;*
- *$N = (E_G, T_N, {}^\bullet(), ()^\bullet, p_N)$ is a Petri net, where the set of places E_G is the edge set, T_N is the set of transitions, ${}^\bullet(), ()^\bullet\colon T_N \to E_G^{\oplus}$ specify the post-set and pre-set of each transition and $p_N\colon T_N \to \mathcal{R}$ is the labelling function;*
- *$m_0 \in (E_G)^{\oplus}$ is the* initial marking *of the Petri graph, satisfying $m_0 = \iota^{\oplus}(E_{G_0})$ for a suitable graph morphism $\iota : G_0 \to G$ (i.e., m_0 must properly correspond to the initial state of the GTS $\mathcal{G}$).*

A marking $m \in E_G^{\oplus}$ will be called reachable (coverable) *in P if it is reachable (coverable) from the initial marking in the Petri net underlying P.*

Remark. The definition of Petri graph is slightly different from the original one in [1], in that we omit some graph morphisms associated to transitions (the μ-component) and to the initial marking, and the so-called irredundancy condition. Both are needed for the actual construction of the Petri graph from a GTS, but they play no role in the results of this paper.

A marking m of a Petri graph can be seen as an abstract representation of a graph in the following sense.

Definition 14.2.5 *Let (G, N, m_0) be a Petri graph and let $m \in E_G^{\oplus}$ be a marking of N. The graph* generated *by m, denoted by $graph(m)$, is the graph H defined as follows: $V_H = \{v \in V_G \mid \exists e \in m\colon (s_G(e) = v \vee t_G(e) = v)\}$, $E_H = \{(e, i) \mid e \in m \wedge 1 \leq i \leq m(e)\}$, $s_H((e, i)) = s_G(e)$, $t_H((e, i)) = t_G(e)$ and $l_H((e, i)) = l_G(e)$.*

Alternatively the graph $graph(m)$ can be defined as the unique graph H, up to isomorphism, such that there exists a morphism $\psi\colon H \to G$ injective on nodes with $\psi^{\oplus}(E_H) = m$. An example of a Petri net marking with the corresponding generated graph can be found in Fig. 14.4.

Given a GTS $(G_0, \mathcal{R})$, with some minor constraints on the format of rewriting rules (see [1, 2]), we can construct a Petri graph approximation of $(G_0, \mathcal{R})$, called *covering* and denoted by $\mathcal{U}(G_0, \mathcal{R})$. The covering is produced by the last step of the following (terminating) algorithm which generates a sequence $P_i = (G_i, N_i, m_i)$ of Petri graphs.

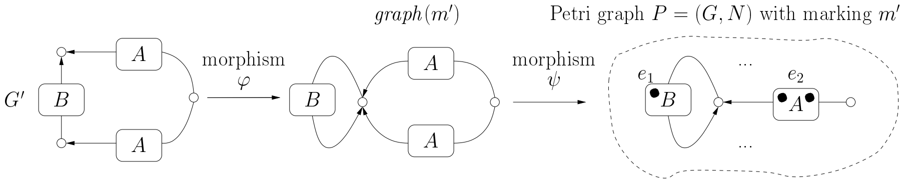


Figure 14.4: A pair (G', m') contained in a simulation.

1. $P_0 = (G_0, N_0, m_0)$, where the net N_0 contains no transitions and $m_0 = E_{G_0}$.
2. As long as one of the following steps is applicable, transform P_i into P_{i+1}, giving precedence to folding steps.

   Unfolding. Find a rule $r = (L, R, \alpha) \in \mathcal{R}$ and a match $\varphi \colon L \to G_i$ such that $\varphi(E_L^{\oplus})$ is coverable in P_i. Then extend P_i by "attaching" R to G_i according to α and add a transition t, labelled by r, describing the application of rule r.

   Folding. Find a rule $r = (L, R, \alpha) \in \mathcal{R}$ and two matches $\varphi, \varphi' \colon L \to G_i$ such that $\varphi^{\oplus}(E_L)$ and $\varphi'^{\oplus}(E_L)$ are coverable in N_i and the second match is causally dependent on the transition unfolding the first match. Then merge the two matches by setting $\varphi(e) \equiv \varphi'(e)$ for each $e \in E_L$ and factoring through the resulting equivalence relation $\equiv$.

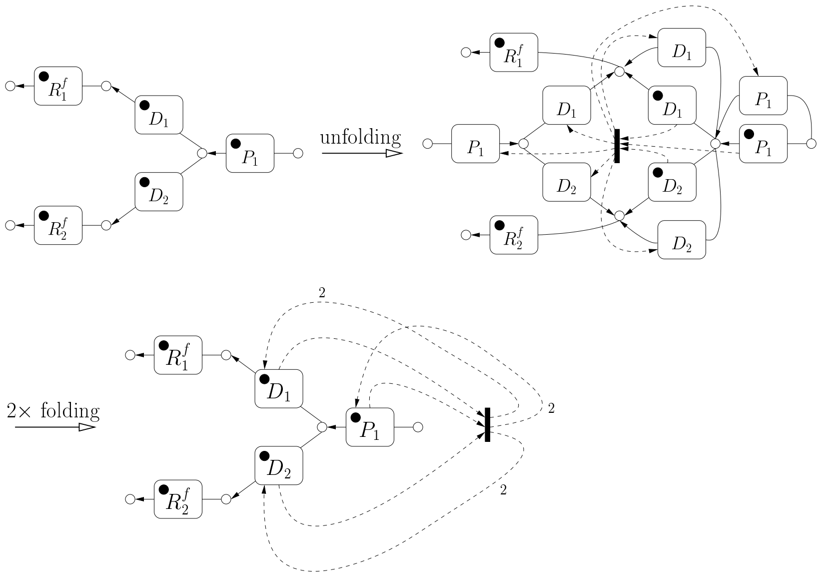


Figure 14.5: An unfolding and two folding steps.

For instance an unfolding step involving rule 3 is depicted in Fig. 14.5. Transitions are represented as black rectangles and the Petri net structure is rendered by connecting edges (places) to transitions with dashed lines. The label k for dashed lines represents the weight with which the target/source place occurs in the post-set (pre-set); when the weight is 1, the label is omitted. In the resulting Petri graph we can find three occurrences of the left-hand side of rule 3. The latter two are causally dependent on the first, which means that they can be merged in two folding steps. The algorithm, starting from the start graph in Fig. 14.1, terminates producing the Petri graph $\mathcal{U}(\mathsf{Sys})$ in Fig. 14.6, where the initial marking is represented by tokens.

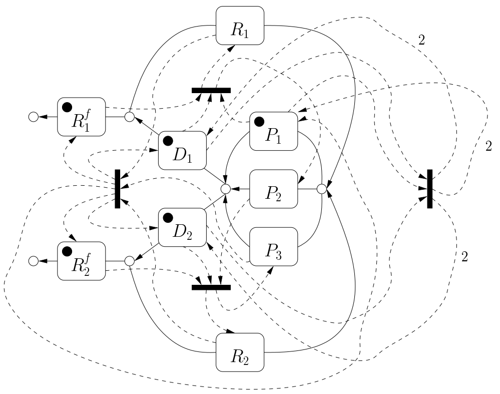


Figure 14.6: The Petri graph $\mathcal{U}(\mathsf{Sys})$ computed as covering of Sys.

The covering $\mathcal{U}(G_0, \mathcal{R})$ is an abstraction of the original GTS $(G_0, \mathcal{R})$ in the following sense.

Proposition 14.2.6 (Abstraction) *Let $\mathcal{G} = (G_0, \mathcal{R})$ be a graph transformation system and let $\mathcal{U}(\mathcal{G}) = (G, N, m_0)$ be its covering. Furthermore let $\mathbf{G}$ be the set of graphs reachable from G_0 in $\mathcal{G}$ and let $\mathbf{M}$ be the set of reachable markings in $\mathcal{U}(\mathcal{G})$. Then there exists a simulation $S \subseteq \mathbf{G} \times \mathbf{M}$ with the following properties:*

- $(G_0, m_0) \in S$;
- *whenever $(G', m') \in S$ and $G' \stackrel{r}{\Rightarrow} G''$, then there exists a marking m'' with $m' \stackrel{r}{\rightarrow} m''$ and $(G'', m'') \in S$;*
- *for every $(G', m') \in S$ there is an edge-bijective morphism $\varphi \colon G' \rightarrow graph(m')$.*

The above result will allow us to use existing results concerning abstractions of reactive systems [CGL99, LGS+95]. Consider the system Sys in our running example. We would like to verify that, according to the design intentions, Sys is deadlock-free. This is formalised by the requirement that all reachable graphs do not contain a vicious cycle, i.e., a cycle of edges where P_2-labelled edges (processes holding a resource and waiting for a second resource) occur twice. This graph property is reflected by graph morphisms, hence, by using Proposition 14.2.6, if we can prove it on the covering $\mathcal{U}(\mathsf{Sys})$, we could deduce that it holds for the original system Sys as well. Observe that actually, in this case, even the stronger property $\#e \leq 1$, where e is the edge labelled P_2, holds for all reachable markings as it can be easily verified by drawing the coverability graph of the Petri net. This is an ad hoc proof of the property, which instead, by the results in this paper, will follow as an instance of a general theory.

The idea that will be concretized by the results in the paper, is the following. Let $\mathcal{G}$ be a GTS and let $\mathcal{U}(\mathcal{G})$ be its covering. By Proposition 14.2.6, $\mathcal{U}(\mathcal{G}) = (G, N, m_0)$ "approximates" $\mathcal{G}$ via a simulation consisting of pairs (G', m') such that G' can be mapped to $\mathit{graph}(m')$ (see, e.g., Fig. 14.4) via an edge-bijective morphism. Given a formula on graphs F, expressing a state property in $\mathcal{G}$, a corresponding formula $M(F)$ on the markings of $\mathcal{U}(\mathcal{G})$ is constructed such that, for any pair in the simulation,

$$m' \models M(F) \;\Rightarrow\; G' \models F.$$

This will be obtained in two steps. First, we will identify formulae F which are reflected by edge-bijective morphisms, ensuring that $\mathit{graph}(m') \models F$ implies $G' \models F$. Then, we will encode F into a propositional formula $M(F)$ on multisets such that $m' \models M(F) \iff \mathit{graph}(m') \models F$.

Call $\mathcal{F}$ the above mentioned class of graph formulae. Now, one can consider a temporal logic over GTSs, where basic predicates are taken from $\mathcal{F}$. For suitable fragments of such logics, e.g., the modal μ-calculus without negation and the "possibility operator" $\diamond$, by Proposition 14.2.6 and exploiting general results in [LGS$^+$95], any temporal formula T over graphs can be translated to a formula $M(T)$ over markings (translating the basic predicates as above), such that, if $N \models M(T)$ then $\mathcal{G} \models T$, i.e., T is valid for the original GTS.

14.3 A Second-Order Monadic Logic for Graphs

We introduce the monadic second-order logic $\mathcal{L}2$ for specifying graph properties. Quantification is allowed over edges, but not over nodes (as, e.g., in [Cou97]).

Definition 14.3.1 (Graph formula) *Let $\mathcal{X}_1 = \{x, y, z, \ldots\}$ be a set of (first-order) edge variables and let $\mathcal{X}_2 = \{X, Y, Z, \ldots\}$ be a set of (second-order) variables representing edge sets. The set of* graph formulae *of the logic $\mathcal{L}2$ is defined as follows, where $\ell \in \Lambda$*

$$\begin{array}{llr} F \;::= & x = y \;\mid\; s(x) = s(y) \;\mid\; s(x) = t(y) \;\mid\; t(x) = t(y) \;\mid & \\ & \mathit{lab}(x) = \ell \;\mid\; x \in X & \textit{(Predicates)} \\ & F \vee F \;\mid\; F \wedge F \;\mid\; F \Rightarrow F \;\mid\; \neg F & \textit{(Connectives)} \\ & \forall x.F \;\mid\; \exists x.F \;\mid\; \forall X.F \;\mid\; \exists X.F & \textit{(Quantifiers)} \end{array}$$

We denote by $\mathit{free}(F)$ and $\mathit{Free}(F)$ the sets of first-order and second-order variables, respectively, occurring free in F, defined in the obvious way.

Note that, even if quantification over nodes is disallowed, formulae expressing properties of classes of nodes can be easily stated, e.g., the property "for all non-isolated nodes v it holds that $P(v)$" is formalised as "$\forall x.(P(s(x)) \wedge P(t(x)))$".

Definition 14.3.2 (Quantifier depth) *The first-order and second-order* quantifier depth *($\mathrm{qd}_1(F)$ and $\mathrm{qd}_2(F)$, respectively) of a graph formula F in $\mathcal{L}2$ is inductively defined as follows, where A is a predicate, $op \in \{\wedge, \vee, \Rightarrow\}$ and $i \in \{1, 2\}$.*

$$\begin{array}{ccc} \mathrm{qd}_i(A) = 0 & \mathrm{qd}_i(\neg F_1) = \mathrm{qd}_i(F_1) & \mathrm{qd}_i(F_1\, op\, F_2) = \max\{\mathrm{qd}_i(F_1), \mathrm{qd}_i(F_2)\} \\ \multicolumn{2}{c}{\mathrm{qd}_1(\forall x.F_1) = \mathrm{qd}_1(\exists x.F_1) = \mathrm{qd}_1(F_1) + 1} & \mathrm{qd}_2(\forall x.F_1) = \mathrm{qd}_2(\exists x.F_1) = \mathrm{qd}_2(F_1) \\ \multicolumn{2}{c}{\mathrm{qd}_1(\forall X.F_1) = \mathrm{qd}_1(\exists X.F_1) = \mathrm{qd}_1(F_1)} & \mathrm{qd}_2(\forall X.F_1) = \mathrm{qd}_2(\exists X.F_1) = \mathrm{qd}_2(F_1) + 1 \end{array}$$

The notion of satisfaction is defined in a straightforward way.

Definition 14.3.3 (Satisfaction) *Let G be a graph, let F be a graph formula in $\mathcal{L}2$, let $\sigma : \mathit{free}(F) \to E_G$ and $\Sigma : \mathit{Free}(F) \to \mathcal{P}(E_G)$ be valuations for the free first- and second-order variables of F, respectively. The* satisfaction relation *$G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance:*

$$\begin{array}{rcl} G \models_{\sigma,\Sigma} x = y & \iff & \sigma(x) = \sigma(y) \\ G \models_{\sigma,\Sigma} s(x) = s(y) & \iff & s_G(\sigma(x)) = s_G(\sigma(y)) \\ G \models_{\sigma,\Sigma} \mathit{lab}(x) = \ell & \iff & l_G(\sigma(x)) = \ell \\ G \models_{\sigma,\Sigma} x \in X & \iff & \sigma(x) \in \Sigma(X) \end{array}$$

Example 14.3.4 The formula NC_ℓ below states that a graph does not contain a cycle including two distinct edges labelled ℓ, a property that will be used to express the absence of vicious cycles in our system Sys. It is based on the formula $NP(x, y)$, which says that there is no path connecting the edges x and y, stating that a set that contains at least all successors of x does not always contain y.

$$\begin{array}{rcl} NP(x,y) & = & \neg\forall X.(\forall z.(t(x) = s(z) \vee \exists w.(w \in X \wedge t(w) = s(z))) \Rightarrow z \in X) \\ & & \qquad \Rightarrow y \in X) \\ NC_\ell & = & \forall x.\forall y.(\mathit{lab}(x) = \ell \wedge \mathit{lab}(y) = \ell \wedge \neg(x = y) \\ & & \qquad \Rightarrow NP(x,y) \vee NP(y,x)) \end{array}$$

The following standard argument shows that this property can not be stated in first-order logic, a fact which motivates our choice of considering a second-order logic: it is easy to find sentences ψ_n in first-order logic stating that 'there is no cycle of length $\leq n$ through two distinct edges labelled ℓ'. Every finite subset of the theory $T = \{\neg NC_\ell\} \cup \{\psi_n\}_{n\in\mathbb{N}}$ is satisfiable but T itself is not satisfiable. The compactness theorem rules this out for first-order theories, so NC_ℓ cannot be first-order.

14.4 Preservation and Reflection of Graph Formulae

In this section we introduce a type system over graph formulae in $\mathcal{L}2$ which allows us to single out subclasses of formulae preserved or reflected by edge-bijective morphisms. By Proposition 14.2.6, given a GTS $\mathcal{G}$ every graph reachable in $\mathcal{G}$ can be mapped homomorphically via an edge-bijective morphism to

Typing predicates:

$$s(x) = s(y),\ s(x) = t(y),\ t(x) = t(y)\colon\ \rightarrow \qquad x = y,\ lab(x) = \ell,\ x \in X\colon\ \leftrightarrow$$

Typing connectives and quantifiers:

$$\frac{F\colon d}{\neg F\colon d^{-1}} \qquad \frac{F_1, F_2\colon d}{F_1 \vee F_2, F_1 \wedge F_2\colon d} \qquad \frac{F_1\colon d^{-1}, \quad F_2\colon d}{F_1 \Rightarrow F_2\colon d}$$

$$\frac{F\colon d}{\forall x.F\colon d} \qquad \frac{F\colon d}{\exists x.F\colon d} \qquad \frac{F\colon d}{\forall X.F\colon d} \qquad \frac{F\colon d}{\exists X.F\colon d}$$

Figure 14.7: The type system for preservation and reflection.

the graph generated by a marking reachable in the covering $\mathcal{U}(\mathcal{G})$ of $\mathcal{G}$. Hence a formula reflected by all edge-bijective morphisms can be safely checked over the approximation $\mathcal{U}(\mathcal{G})$, in the sense that if it holds in $\mathcal{U}(\mathcal{G})$, then we can deduce that it holds also in $\mathcal{G}$.

To define the notions of reflection (and preservation) of general graph formulae, possibly with free variables, observe that valuations are naturally "transformed" under graph morphisms. Let F be formula, let $\varphi : G_1 \to G_2$ be a graph morphism, and let $\sigma_1 : \mathit{free}(F) \to E_{G_1}$ and $\Sigma_1 : \mathit{Free}(F) \to \mathcal{P}(E_{G_1})$ be valuations. A valuation for the first-order variables of F in G_2 is naturally given by $\varphi \circ \sigma_1$, while a valuation Σ_2 for second-order variables can be defined by $\Sigma_2(X) = \varphi(\Sigma_1(X))$ for any variable X. Abusing the notation, Σ_2 will be denoted by $\varphi \circ \Sigma_1$.

Definition 14.4.1 (Reflection and Preservation) *Let F be a formula in $\mathcal{L}2$ and let $\varphi\colon G_1 \to G_2$ be a graph morphism. We say that F is* preserved by *φ if for all valuations $\sigma_1\colon \mathit{free}(F) \to E_{G_1}$ and $\Sigma_1\colon \mathit{Free}(F) \to \mathcal{P}(E_{G_1})$*

$$G_1 \models_{\sigma_1,\Sigma_1} F \quad \Rightarrow \quad G_2 \models_{\varphi\circ\sigma,\varphi\circ\Sigma_1} F.$$

Symmetrically, F is reflected *by φ if the above holds where $\Rightarrow$ is replaced by $\Leftarrow$.*

Observe that, in particular, a closed formula F is preserved by a graph morphism $\varphi : G_1 \to G_2$ if $G_1 \models_{\emptyset,\emptyset} F$ implies $G_2 \models_{\emptyset,\emptyset} F$.

As mentioned above we are interested in syntactic criteria characterising classes of graph formulae reflected, respectively preserved, by all edge-bijective graph morphisms. For first-order predicate logic, criteria for arbitrary morphisms can be found in [Hod93]. Here we provide a technique which works for general second-order monadic formulae, based on a type system assigning to every formula F either $\rightarrow$, meaning that F is preserved, or $\leftarrow$, meaning that F is reflected by edge-bijective morphisms. The type rules are given in Fig. 14.7 where it is intended that $\rightarrow^{-1}=\leftarrow$ and $\leftarrow^{-1}=\rightarrow$. Moreover $F :\leftrightarrow$ is a shortcut for $F :\rightarrow$ and $F :\leftarrow$, while $F_1, F_2 : d$ stands for $F_1 : d$ and $F_2 : d$.

The type system can be shown to be correct.

Proposition 14.4.2 (Correctness) *Let F be a graph formula. If $F\colon\ \rightarrow$ is provable then F is preserved by all edge-bijective morphisms. Similarly, if $F\colon\ \leftarrow$ is provable then F is reflected by all edge-bijective graph morphisms.*

Proof: The proof goes by induction on the rules needed to prove $F\colon d$. We only prove two cases, the other cases can be shown similarly.

- We assume that F has the form $x \in X$. Hence F is typed by the axiom $x \in X\colon\ \leftarrow$. Now let $\varphi\colon G_1 \to G_2$ be an edge-bijective morphism and let σ_1 and Σ_1 be valuations. Assume that $G_2 \models_{\varphi\circ\sigma_1,\varphi\circ\Sigma_1} x \in X$. This implies that $\varphi(\sigma_1(x)) \in \varphi(\Sigma_1(X))$. Since φ is an edge-bijective morphism it follows that $\sigma(x) \in \Sigma_1(X)$ and we conclude that $G_1 \models_{\sigma_1,\Sigma_1} x \in X$.

- We assume that F has the form $\forall X.F'$. Then F is typed in the following way:

$$\frac{F'\colon\ \rightarrow}{\forall X.F'\colon\ \rightarrow}$$

  Let $\varphi\colon G_1 \to G_2$ be an edge-bijective morphism and let σ_1 and Σ_1 be valuations. Assume that $G_1 \models_{\sigma_1,\Sigma_1} \forall X.F'$. This implies that for all $E_1 \subseteq E_{G_1}$ it holds that $G_1 \models_{\sigma_1,\Sigma_1\cup\{X\mapsto E_1\}} F'$. For any $E_2 \subseteq E_{G_2}$ we can infer

$$G_1 \models_{\sigma,\Sigma_1\cup\{X\mapsto\varphi^{-1}(E_2)\}} F'.$$

  By induction hypothesis we have that $G_2 \models_{\varphi\circ\sigma_1,\varphi\circ\Sigma_1\cup\{X\mapsto E_2\}} F'$ for all $E_2 \subseteq E_{G_2}$ and we obtain $G_2 \models_{\varphi\circ\sigma_1,\varphi\circ\Sigma_1} F'$.

□

Example 14.4.3 It holds that $NP(x,y)\colon\ \leftarrow$ and $NC_\ell\colon\ \leftarrow$, i.e., absence of paths and of vicious cycles is reflected by edge-bijective morphisms.

Not all formulae that are preserved respectively reflected are recognised by the above type system. The following result shows that this incompleteness is a fundamental problem, due to the undecidability of reflection and preservation.

Proposition 14.4.4 (Undecidability of the Reflection (Preservation) Problem for formulae) *The following two sets are undecidable:*

$$\begin{aligned} \mathit{Refl}_{FO} &= \{F \mid F \text{ closed first-order formula, reflected by edge-bijective graph morphisms}\} \\ \mathit{Pres}_{FO} &= \{F \mid F \text{ closed first-order formula, preserved by edge-bijective graph morphisms}\} \end{aligned}$$

Proof: We show that Refl_{FO} is undecidable, the proof for Pres_{FO} is provided by the fact that a formula F is preserved if and only if $\neg F$ is reflected.

Let Λ' be a multi-set over elements of Λ. By $G(\Lambda')$ we denote the graph with exactly one node v and $|\Lambda'|$ edges, where each edge e satisfies $s_{G(\Lambda')}(e) = v = t_{G(\Lambda')}(e)$ and for every $\ell \in \Lambda$ there are exactly $\Lambda'(\ell)$ edges. Since every graph can be mapped edge-bijectively to one of the $G(\Lambda')$, it is fairly easy to see that a formula F is a tautology if and only if it is contained in Refl_{FO} and it holds for each of the $G(\Lambda')$. Because of Corollary 14.7.3 it is not necessary to check infinitely many graphs of the form $G(\Lambda')$, but it is possible to compute an upper bound for the number of edges from the quantifier depth of F. If Refl_{FO} were decidable, this would give us a procedure to decide the first-order theory on the class of graphs, a contradiction to a result of Trakhtenbrot [Tra50]. $\square$

14.5 A Propositional Logic on Multisets

In order to characterise markings of Petri nets we use the following logic on multisets. We consider a fixed universe A over which all multisets are formed.

Definition 14.5.1 (Multiset formula) *The set of* multiset formulae*, ranged over by M, is defined as follows, where $a \in A$ and $c \in \mathbb{N}$*

$$M \ ::= \ \#a \leq c \ \mid \ \neg M \ \mid \ M \vee M' \ \mid \ M \wedge M'.$$

Let m be a multiset with elements from A. The satisfaction relation $m \models M$ is defined, on basic predicates, as $m \models (\#a \leq c) \iff m(a) \leq c$. Logical connectives are dealt with as usual.

We will consider also derived predicates of the form $\#a \geq c$ and $\#a = c$ where

$$\begin{aligned} (\#a \geq c) &= \begin{cases} \neg(\#e \leq c-1) & \text{if } c > 0 \\ \mathit{true} & \text{otherwise} \end{cases} \\ (\#e = c) &= (\#e \leq c) \wedge (\#e \geq c). \end{aligned}$$

14.6 Encoding First-Order Graph Logic

In this section we show how first-order graph formulae can be encoded into "equivalent" multiset formulae. More precisely, given the fixed Petri graph $P = (G, N, m_0)$ the aim is to find an encoding M_1 of first-order graph formulae into multiset formulae such that $\mathit{graph}(m) \models F \iff m \models M_1(F)$ for every marking m of P and every closed first order graph formula F.

The encoding M_1 is based on the following observation: every graph $\mathit{graph}(m)$ for some marking m of P can be generated from the finite "template graph" G in the following way: some edges of G might be removed and some edges might be multiplied, generating several parallel copies of the same template edge. Whenever a formula has two free variables x, y and $\mathit{graph}(m)$ has n parallel copies $e_1, \dots, e_n$ of the same edge, it is not necessary to associate x and y with all edges, but it is sufficient to assign e_1 to x and e_2 to y (first alternative) or e_1 to both x and y (second alternative). Thus, whenever we encode a formula F, we have to keep track of the following information: a partition P on the free variables $\mathit{free}(F)$, telling us which variables are mapped to the same edge, and a mapping ρ from $\mathit{free}(F)$ to the edges of G, with $\rho(x) = e$ meaning that x will be instantiated with a copy of the template edge e. Since there might be several different copies of the same template edge, two variables

x and y in different sets of P can be mapped by ρ to the same edge of G. Whenever we encode an existential quantifier $\exists x$, we have to form a disjunction over all the possibilities we have in choosing such an x: either x is instantiated with the same edge as another free variable y, in this case x and y should be in the same set of the partition P. Or x is instantiated with a new copy of an edge in G. In this case, a new set $\{x\}$ is added to P and we have to make sure that enough edges are available by adding a suitable predicate.

We need the following notation. We will describe an equivalence relation on a set A by a partition $P \subseteq \mathcal{P}(A)$ of A, where every element of P represents an equivalence class. We will write $x\,P\,y$ whenever x, y are in the same equivalence class. Furthermore we assume that each equivalence P is associated with a function $rep : P \to A$ which assigns a representative to every equivalence class. The encoding given below is independent of any specific choice of representatives.

Given a function $f : A \to B$ such that $f(a) = f(a')$ for all $a, a' \in A$ with aPa' and a fixed $b \in B$ we define $n_{P,f}(b) = |\{k \in P \mid f(rep(k)) = b\}|$, i.e., $n_{P,f}(b)$ is the number of sets in the partition P that are mapped to b.

Definition 14.6.1 *Let G be a directed graph, let F be graph formula in the first-order fragment of $\mathcal{L}2$, let $\rho : free(F) \to E_G$ and let $P \subseteq \mathcal{P}(free(F))$ be an equivalence relation such that $x\,P\,y$ implies $\rho(x) = \rho(y)$ for all $x, y \in free(F)$. The* encoding *M_1 is defined as follows:*

$$
\begin{array}{rcl}
M_1[\neg F, \rho, P] & = & \neg M_1[F, \rho, P] \\
M_1[F_1 \vee F_2, \rho, P] & = & M_1[F_1, \rho, P] \vee M_1[F_1, \rho, P] \\
M_1[F_1 \wedge F_2, \rho, P] & = & M_1[F_1, \rho, P] \wedge M_1[F_1, \rho, P] \\
M_1[x = y, \rho, P] & = & \left\{ \begin{array}{ll} \mathit{true} & \mathit{if}\ x\,P\,y \\ \mathit{false} & \mathit{otherwise} \end{array} \right. \\
M_1[lab(x) = \ell, \rho, P] & = & \left\{ \begin{array}{ll} \mathit{true} & \mathit{if}\ l_G(\rho(x)) = \ell \\ \mathit{false} & \mathit{otherwise} \end{array} \right. \\
M_1[s(x) = s(y), \rho, P] & = & \left\{ \begin{array}{ll} \mathit{true} & \mathit{if}\ s_G(\rho(x)) = s_G(\rho(y)) \\ \mathit{false} & \mathit{otherwise} \end{array} \right. \\
 & & \textit{the formulae } t(x) = t(y) \textit{ and } s(x) = t(y) \\
 & & \textit{are treated analogously} \\
M_1[\exists x.F, \rho, P] & = & \bigvee_{k \in P} (M_1[F, \rho \cup \{x \mapsto \rho(rep(k))\}, P \backslash \{k\} \cup \{k \cup \{x\}\}]) \vee \\
 & & \bigvee_{e \in E_G} (M_1[F, \rho \cup \{x \mapsto e\}, P \cup \{\{x\}\}] \wedge (\#e \geq n_{P,\rho}(e) + 1)) \\
M_1[\forall x.F, \rho, P] & = & \bigwedge_{k \in P} (M_1[F, \rho \cup \{x \mapsto \rho(rep(k))\}, P \backslash \{k\} \cup \{k \cup \{x\}\}]) \wedge \\
 & & \bigwedge_{e \in E_G} ((\#e \geq n_{P,\rho}(e) + 1) \Rightarrow M_1[F, \rho \cup \{x \mapsto e\}, P \cup \{\{x\}\}])
\end{array}
$$

If F is a closed formula (i.e., without free variables), we define $M_1(F) = M_1[F, \emptyset, \emptyset]$.

It is worth remarking that such an approach is similar to the model-theoretic method of quantifier elimination, defined by Tarski in the 1950's to show decidability and completeness for theories like dense linear orderings or algebraically closed fields (see [Rob63]). We remark that here finiteness of graphs is essential.

We can now show that the encoding is correct in the sense explained above. We will omit the index Σ in $\models_{\sigma,\Sigma}$ when talking about first-order formulae only.

Proposition 14.6.2 *Let (G, N, m_0) be a Petri graph, F a first-order formula in $\mathcal{L}2$ and m a marking of N. Then it holds that*

$$\mathit{graph}(m) \models_\sigma F \iff m \models M_1[F, \rho, P],$$

when

- $\rho : \mathit{free}(F) \to E_G$;
- *P is an equivalence on $\mathit{free}(F)$ such that $x\,P\,y$ implies $\rho(x) = \rho(y)$ for any $x, y \in \mathit{free}(F)$;*
- *$\sigma : \mathit{free}(F) \to E_{\mathit{graph}(m)}$ satisfies $x\,P\,y \iff \sigma(x) = \sigma(y)$ and $\varphi \circ \sigma = \rho$, where $\varphi\colon \mathit{graph}(m) \to G$ denotes the projection of $\mathit{graph}(m)$ over G, i.e., a graph morphism such that $\varphi((e, i)) = e \in E_G$.*

Proof: We assume that we have a fixed marking m and a logical formula F on graphs. We first show the direction from left to right and afterwards the direction from right to left.

$\Rightarrow$: We proceed by induction on the structure of F.

$F = (x = y)$: since it holds that $\mathit{graph}(m) \models_\sigma x = y$ we can conclude that $\sigma(x) = \sigma(y)$ which implies $x\,P\,y$ and therefore $\rho(x) = \rho(y)$.
Furthermore we can conclude that $M_1[x = y, \rho, P] = \mathit{true}$ and therefore $m \models M_1[x = y, \rho, P]$.

$F = (\mathit{lab}(x) = \ell)$: since it holds that $\mathit{graph}(m) \models_\sigma \mathit{lab}(x) = \ell$, it follows that $l_G(\rho(x)) = l_G(\varphi(\sigma(x))) = l_{\mathit{graph}(m)}(\sigma(x)) = \ell$. Therefore we know that $M_1[\mathit{lab}(x) = \ell, \rho, P] = \mathit{true}$ and it holds that $m \models M_1[\mathit{lab}(x) = \ell, \rho, P]$.

$F = (s(x) = s(y))$: we assume that $\mathit{graph}(m) \models_\sigma s(x) = s(y)$. So we have $s_{\mathit{graph}(m)}(\sigma(x)) = s_{\mathit{graph}(m)}(\sigma(y))$ and since φ is a graph morphism it holds that $s_G(\rho(x)) = s_G(\varphi(\sigma(x))) = s_G(\varphi(\sigma(y))) = s_G(\rho(y))$.
So $M_1[s(x) = s(y), \rho, P] = \mathit{true}$ and $m \models M_1[s(x) = s(y), \rho, P]$ holds.

$F = \neg F'$: we assume that $\mathit{graph}(m) \models_\sigma \neg F'$ holds. Now $\mathit{graph}(m) \not\models_\sigma F'$ and from the induction hypothesis it follows that $m \not\models M_1[F', \rho, P]$. Therefore $m \models \neg M_1[F', \rho, P]$.

$F = F_1 \vee F_2$: we assume that $\mathit{graph}(m) \models_\sigma F_1 \vee F_2$ holds. This implies that $\mathit{graph}(m) \models_\sigma F_1$ or $\mathit{graph}(m) \models_\sigma F_2$. We assume that the first condition holds, the other case can be handled analogously.
From the induction hypothesis it follows that $m \models M_1[F_1, \rho, P]$. Therefore $m \models M_1[F_1, \rho, P] \vee M_1[F_2, \rho, P]$.

$F = F_1 \wedge F_2$: analogous to the case of $\vee$.

$F = \exists x.F'$**:** we assume that $graph(m) \models_\sigma \exists x.F'$ holds. This implies that there exists an edge $e \in E_{graph(m)}$ such that $graph(m) \models_{\sigma\cup\{x\mapsto e\}} F'$ holds (see Definition 14.3.3).

We distinguish the following two cases:

- there is no $w \in free(F)$ such that $\sigma(w) = e$. We define ρ', σ', P' as follows: [1]
  - $\rho' : free(F') = free(F) \cup \{x\} \to E_G$ with $\rho' = \rho \cup \{x \mapsto \varphi(e)\}$.
  - $\sigma' : free(F') = free(F) \cup \{x\} \to E_{graph(m)}$ with $\sigma' = \sigma \cup \{x \mapsto e\}$.
  - P' is an equivalence on $free(F')$ with $P' = P \cup \{\{x\}\}$.

  We show that ρ', σ' and P' satisfy the conditions stated in the proposition. First, it obviously holds that $\varphi \circ \sigma' = \rho'$.
  Now let $y\,P'\,z$ for $y, z \in free(F')$. It might either be the case that $y, z \in free(F)$ which implies that $y\,P\,z$ and therefore $\sigma(y) = \sigma(z)$ and $\sigma'(y) = \sigma'(z)$. Or it might be the case that $y = x = z$ and it immediately follows that $\sigma'(y) = \sigma'(z)$.
  Now let $\sigma'(y) = \sigma'(z)$. Because of the construction of σ and because of the fact that no element of $free(F)$ maps to e, it follows that either $y, z \in free(F)$ and $\sigma(y) = \sigma(z)$ and therefore $y\,P\,z$ and $y\,P'\,z$, or $y = x = z$, $\sigma'(y) = e = \sigma'(z)$ and therefore also $y\,P'\,z$.
  From this it follows immediately that $y\,P'\,z$ implies $\rho'(y) = \rho'(z)$.
  Since $graph(m) \models_{\sigma'} F'$ holds we can infer from the induction hypothesis that $m \models M_1[F', \rho', P']$ is true.
  Furthermore it holds that

  $$\begin{aligned} & m(\varphi(e)) \\ = \; & |\{e' \in E_{graph(m)} \mid \varphi(e') = \varphi(e)\}| \\ \geq \; & |\{e' \in E_{graph(m)} \mid \varphi(e') = \varphi(e) \wedge \exists y \in free(F) \colon (\sigma(y) = e')\}| \\ & \quad +1 \\ = \; & |\{k \in P \mid \varphi(\sigma(rep(k))) = \varphi(e)\}| + 1 = n_{P,\rho}(\varphi(e)) + 1 \end{aligned}$$

  This implies that $m \models (\#\varphi(e) \geq n_{P,\rho}(\varphi(e)) + 1)$.
  Since $\varphi(e)$ is an element of E_G it follows from the considerations above that at least one element of the second part of the disjunction in the formula $M_1[\exists x.F', \rho, \sigma]$ is true. And this implies $m \models M_1[\exists x.F', \rho, \sigma]$.
- there exists a variable $w \in free(F)$ such that $\sigma(w) = e$. Let k be the equivalence class of P that contains w. We define ρ' and σ' as above and $P' = P\backslash\{k\} \cup \{k \cup \{x\}\}$.
  As before it holds that $\varphi \circ \sigma' = \rho'$.
  Now let $y\,P'\,z$. If $y, z \in free(F)$, then it follows that $y\,P\,z$, that furthermore $\sigma(y) = \sigma(z)$ and therefore $\sigma'(y) = \sigma'(z)$. If $y = x$

[1] We assume that in a formula of the form $\exists x.F'$ respectively $\forall x.F'$ the variable x occurs free in F'. Otherwise we could just remove the quantifier.

and $z \in free(F)$, then it holds that z is in the equivalence class of w wrt. P and we can conclude that $\sigma'(y) = \varphi(e) = \sigma'(w) = \sigma'(z)$. If $y = x = z$, then $\sigma'(y) = \sigma'(z)$ follows immediately.

We assume that $\sigma'(y) = \sigma'(z)$. If $y, z \in free(F)$ then it follows that $\sigma(y) = \sigma(z)$ and therefore $y\,P\,z$, which implies $y\,P'\,z$. If, however, $y = x$ and $z \in free(F)$, then it follows that $\sigma(z) = \sigma'(z) = \varphi(e) = \sigma(w)$. This implies that $z\,P\,w\,P'\,y$ and therefore $y\,P'\,z$. If $y = x = z$, then it follows immediately that $y\,P'\,z$.

From the fact that $\varphi \circ \sigma' = \rho'$ and the considerations above, it follows that $y\,P'\,z$ implies $\rho'(y) = \rho'(z)$.

Since $graph(m) \models_{\sigma'} F'$ holds, it follows from the induction hypothesis that $m \models M_1[F', \rho', P']$ is true. Since furthermore $\rho(rep(k)) = \rho(w) = \varphi(\sigma(w)) = \varphi(e)$, it follows that at least one of the elements of the first disjunction in the formula $M_1[\exists x.F', \rho, P]$ is true and therefore $m \models M_1[\exists x.F, \rho, P]$ holds.

$\Leftarrow$: Again we proceed by induction on the structure of F.

$F = (x = y)$: we assume that $m \models M_1[x = y, \rho, P]$. Therefore we can conclude that $x\,P\,y$, since otherwise $M_1[x = y, \rho, P] = false$. This implies that $\sigma(x) = \sigma(y)$ and we can conclude that $graph(m) \models_\sigma x = y$.

$F = lab(x) = \ell)$: we assume that $m \models M_1[lab(x) = \ell, \rho, P]$. This implies that $lab_G(\rho(x)) = \ell$, since otherwise $M_1[lab(x, A), \rho, P] = false$.

So it holds that $lab_{graph(m)}(\sigma(x)) = lab_G(\varphi(\sigma(x))) = lab_G(\rho(x)) = \ell$ and we can infer that $graph(m) \models_\sigma lab(x) = \ell$.

$F = (s(x) = s(y))$: since $m \models M_1[s(x) = s(y), \rho, P]$, it holds that $s(\rho(x)) = s(\rho(y))$, since otherwise $M_1[s(x) = s(y), \rho, P] = false$.

We can infer that $\varphi(s(\sigma(x))) = \varphi(s(\sigma(y)))$ and since φ is injective on nodes this implies that $s(\sigma(x)) = s(\sigma(y))$. Therefore $graph(m) \models_\sigma s(x) = s(y)$.

$F = \neg F'$: we assume that $m \models M_1[\neg F', \rho, P] = \neg M_1[F', \rho, P]$, which implies that $m \not\models M_1[F', \rho, P]$. From the induction hypothesis it follows that $graph(m) \not\models_\sigma F'$ and therefore also $graph(m) \models_\sigma \neg F'$.

$F = F_1 \vee F_2$: we assume that $m \models M_1[F_1 \vee F_2, \rho, P] = M_1[F_1, \rho, P] \vee M_1[F_2, \rho, P]$, which implies that $m \models M_1[F_1, \rho, P]$ or $m \models M_1[F_2, \rho, P]$. We assume that the first condition holds, the other case can be handled analogously.

From the induction hypothesis it follows that $graph(m) \models_\sigma F_1$ and therefore also $graph(m) \models_\sigma F_1 \vee F_2$.

$F = \exists x.F'$: we assume that $m \models M_1[\exists x.F', \rho, P]$ which means that at least one of the elements in the disjunction is true. We consider the following two (overlapping) cases:

- it holds that $m \models M_1[F', \rho \cup \{x \mapsto \rho(rep(k))\}, P \backslash \{k\} \cup \{k \cup \{x\}\}]$. We set $\rho' = \rho \cup \{x \mapsto \rho(rep(k))\}$, $P' = P \backslash \{k\} \cup \{k \cup \{x\}\}$ and $\sigma' = \sigma \cup \{x \mapsto \sigma(rep(k))\}$.

It is immediately clear that $\varphi \circ \sigma' = \rho'$.
Now let $y\,P'\,z$. It might either be the case that $y, z \in \mathit{free}(F)$ and therefore $y\,P\,z$ which implies $\sigma(y) = \sigma(z)$ and also $\sigma'(y) = \sigma'(z)$. Or it holds that $y = x$ and $z \in \mathit{free}(F)$ which means that z is an element of the equivalence class k, which implies that $\sigma'(y) = \sigma(\mathit{rep}(k)) = \sigma(z) = \sigma'(z)$. If, however $y = x = z$, then it follows immediately that $\sigma'(y) = \sigma'(z)$.
Now let $\sigma'(y) = \sigma'(z)$. If $y, z \in \mathit{free}(F)$, then it follows that $\sigma(y) = \sigma(z)$, which implies that $y\,P\,z$ and also $y\,P'\,z$. If $y = x$ and $z \in \mathit{free}(F)$, then it holds $\sigma(z) = \sigma'(z) = \sigma'(y) = \sigma(\mathit{rep}(k))$, which implies that z is in the equivalence class k and therefore $y\,P'\,z$. If, however, $y = x = z$, then it follows immediately that $y\,P'\,z$.
Induction hypothesis implies that $\mathit{graph}(m) \models_{\sigma \cup \{x \mapsto \sigma(\mathit{rep}(k))\}} F'$, which in turn implies that $\mathit{graph}(m) \models_\sigma \exists x.F$.

- if holds that $m \models M_1[F, \rho \cup \{x \mapsto e\}, P \cup \{\{x\}\}] \wedge (\#e \geq n_{P,\rho}(e) + 1)$.
  We set $\rho' = \rho \cup \{x \mapsto e\}$, $P' = P \cup \{\{x\}\}$ and define σ' as follows: since $m \models (\#e \geq n_{P,\rho}(e) + 1)$, it holds that

$$\begin{aligned} & |\{e' \in E_{\mathit{graph}(m)} \mid \varphi(e') = e\}| \\ = \;& m(e) > n_{P,\rho}(e) = |\{k \in P \mid \rho(\mathit{rep}(k)) = e\}| \\ = \;& |\{\hat{e} \in E_{\mathit{graph}(m)} \mid \varphi(e') = e \wedge \exists y \in \mathit{free}(F).(\sigma(y) = \hat{e})\}| \end{aligned}$$

  This implies that there exists at least one edge $e' \in E_{\mathit{graph}(m)}$ such that $\varphi(e') = e$ and for any $y \in \mathit{free}(F)$ it holds that $\sigma(y) \neq e'$.
  We can now define $\sigma' = \sigma \cup \{x \mapsto e'\}$ and it is immediate to see that $\varphi \circ \sigma' = \rho'$.
  Now let $y\,P'\,z$. We first assume that $y, z \in \mathit{free}(F)$, which implies that $y\,P\,z$ and therefore $\sigma(y) = \sigma(z)$ and also $\sigma'(y) = \sigma'(z)$. If $y = x = z$, then it follows immediately that $\sigma'(y) = \sigma'(z)$.
  We now assume that $\sigma'(y) = \sigma'(z)$. If $y, z \in \mathit{free}(F)$, then it holds that $\sigma(y) = \sigma(z)$, which implies that $y\,P\,z$ and therefore $y\,P'\,z$. If $y = x$ then also $z = x$, since otherwise $\sigma'(y) = e' \neq \sigma(z)$ which is a contradiction. So if $y = x = z$ it trivially holds that $y\,P'\,z$.
  From the fact that $\varphi \circ \sigma' = \rho'$ and the considerations above it follows immediately that $y\,P'\,z$ implies $\rho(y) = \rho(z)$.
  Since $m \models M_1[F, \rho', P']$ it follows from the induction hypothesis that $\mathit{graph}(m) \models_{\sigma \cup \{x \mapsto e'\}} F'$ which implies that $\mathit{graph}(m) \models_\sigma \exists x.F'$.

The cases $F = F_1 \wedge F_2$ and $F = \forall x.F'$ can be treated analogously to cases shown above or can be shown by using deMorgan laws. $\square$

Whenever F is closed the proposition above trivially gives us the expected result. i.e., $\mathit{graph}(m) \models F$ iff $m \models M_1(F)$.

Example 14.6.3 Consider the formula $F = \exists x.(\underbrace{lab(x) = A \wedge \overbrace{\forall y.\neg(t(x) = s(y))}^{F_2}}_{F_1})$.

The graph under consideration is the graph G on the right in Fig. 14.4 (containing a looping B-edge e_1 and an A-edge e_2). The encoding goes as follows (with some simplifications of the formula along the way):

$$
\begin{aligned}
& M_1[F, \emptyset, \emptyset] \\
= \;& (M_1[F_1, \{x \mapsto e_1\}, \{\{x\}\}] \wedge (\#e_1 \geq 1)) \vee (M_1[F_1, \{x \mapsto e_2\}, \{\{x\}\}] \wedge (\#e_2 \geq 1)) \\
= \;& (\underbrace{M_1[lab(x) = A, \{x \mapsto e_1\}, \{\{x\}\}]}_{=false} \wedge M_1[F_2, \{x \mapsto e_1\}, \{\{x\}\}] \wedge (\#e_1 \geq 1)) \vee \\
& (\underbrace{M_1[lab(x) = A, \{x \mapsto e_2\}, \{\{x\}\}]}_{=true} \wedge M_1[F_2, \{x \mapsto e_2\}, \{\{x\}\}] \wedge (\#e_2 \geq 1)) \\
\equiv \;& \underbrace{M_1[\neg(t(x) = s(y)), \{x, y \mapsto e_2\}, \{\{x, y\}\}]}_{=true} \wedge \\
& (\#e_1 \geq 1 \Rightarrow \underbrace{M_1[\neg(t(x) = s(y)), \{x \mapsto e_2, y \mapsto e_1\}, \{\{x\}, \{y\}\}]}_{=false}) \wedge \\
& (\#e_2 \geq 2 \Rightarrow \underbrace{M_1[\neg(t(x) = s(y)), \{x, y \mapsto e_2\}, \{\{x\}, \{y\}\}]}_{=true}) \wedge (\#e_2 \geq 1) \\
\equiv \;& \neg(\#e_1 \geq 1) \wedge (\#e_2 \geq 1)
\end{aligned}
$$

14.7 Encoding Monadic Second-Order Graph Logic

In this section we show that also general monadic second-order graph formulae in $\mathcal{L}2$ can be encoded into multiset formulae. Differently from the first-order case, the encoding is not defined inductively, but, still, quantifier elimination is possible. We start with an easy but useful lemma.

Lemma 14.7.1 (Edge Permutations) *Let σ, Σ be valuations such that $G \models_{\sigma,\Sigma} F$. Furthermore let $\pi : G \to G$ be an automorphism such that $s_G(e) = s_G(\pi(e))$ and $t_G(e) = t_G(\pi(e))$. Then $G \models_{\pi\circ\sigma,\pi\circ\Sigma} F$.*

The encoding uses the fact that multiple copies of an edge are distinguished only by their identity, but have the same source and target nodes and the same label. Hence whenever we want to encode a first-order quantifier, we only have to check all the edges that have already appeared so far and a fresh copy of every edge in G. From this, as we will see, one can infer that for checking the validity of a formula F it is sufficient to consider only up to $\mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}$ copies of every edge in the template graph G.

The following proposition basically states that if there are enough parallel edges which belong to the same sets of the form $\Sigma(X)$, where Σ is a second-order valuation and X a second-order variable, then one of these edges can be removed—provided that it is not in the range of the first-order valuation σ—without changing the validity of a formula F.

Proposition 14.7.2 *Let G be a graph, F a graph formula in $\mathcal{L}2$, let σ, Σ be valuations for the free variables in F and let $e \in E_G$ be a fixed edge. Assume that*

(1) the edge e is not in the range of σ and

(2) $|E^G_\Sigma(e)| > (\mathrm{qd}_1(F) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F)}$ where

$$\begin{aligned} E^G_\Sigma(e) \quad = \{e' \in E_G \mid \quad & s_G(e) = s_G(e'), t_G(e) = t_G(e'), l_G(e) = l_G(e'), \\ & \forall X \in \mathrm{dom}(\Sigma).(e \in \Sigma(X) \iff e' \in \Sigma(X))\} \end{aligned}$$

Then $G \models_{\sigma,\Sigma} F \iff G \setminus \{e\} \models_{\sigma,\Sigma_e} F$, where $G \setminus \{e\}$ is obtained by removing the edge e from graph G and $\Sigma_e(X) = \Sigma(X) - \{e\}$.

Proof: We go by structural induction on F.

$F = (x = y)$**:** It holds that

$$G \models_{\sigma,\Sigma} x = y \iff \sigma(x) = \sigma(y) \iff G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} x = y,$$

since e is not in the range of σ.

$F = (s(x) = t(y))$**:** It holds that

$$\begin{aligned} & G \models_{\sigma,\Sigma} s(x) = t(y) \iff s_G(\sigma(x)) = t_G(\sigma(y)) \\ \iff \quad & s_{G\backslash\{e\}}(\sigma(x)) = t_{G\backslash\{e\}}(\sigma(y)) \iff G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} s(x) = t(y). \end{aligned}$$

$F = lab(x, A)$**:** It holds that

$$\begin{aligned} & G \models_{\sigma,\Sigma} lab(x, A) \iff l_G(\sigma(x)) = A \iff l_{G\backslash\{e\}}(\sigma(x)) = A \\ \iff \quad & G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} lab(x, A). \end{aligned}$$

$F = x \in X$**:** It holds that

$$\begin{aligned} & G \models_{\sigma,\Sigma} x \in X \iff \sigma(x) \in \Sigma(X) \iff \sigma(x) \in \Sigma(X)\backslash\{e\} \\ \iff \quad & \sigma(x) \in (\Sigma\backslash\{e\})(X) \iff G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} x \in X. \end{aligned}$$

$F = \neg F_1$**:** It holds that

$$\begin{aligned} & G \models_{\sigma,\Sigma} \neg F_1 \iff G \not\models_{\sigma,\Sigma} F_1 \iff G\backslash\{e\} \not\models_{\sigma,\Sigma\backslash\{e\}} F_1 \\ \iff \quad & G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} \neg F_1 \end{aligned}$$

with the induction hypothesis and with the fact that F and F_1 have the same quantifier depth.

$F = F_1 \wedge F_2$**:** It holds that

$$G \models_{\sigma,\Sigma} F_1 \wedge F_2 \iff G \models_{\sigma,\Sigma} F_1 \text{ and } G \models_{\sigma,\Sigma} F_2 \iff$$

$$G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} F_1 \text{ and } G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}} F_2 \iff G \models_{\sigma,\Sigma\backslash\{e\}} F_1 \wedge F_2$$

with induction hypothesis and the fact that $\mathrm{qd}_1(F_1) \leq \mathrm{qd}_1(F)$, $\mathrm{qd}_1(F_2) \leq \mathrm{qd}_1(F)$, $\mathrm{qd}_2(F_1) \leq \mathrm{qd}_2(F)$ and $\mathrm{qd}_2(F_2) \leq \mathrm{qd}_2(F)$.

$F = \forall x.F_1$**:** $G \models_{\sigma,\Sigma} \forall x.F_1$ holds if and only if for all $e' \in E_G$ we have that $G \models_{\sigma \cup \{x \mapsto e'\},\Sigma} F_1$ (Condition (A)).

Let us first observe that Condition (2) is satisfied for the formula F_1: i.e. for all $e' \in E_G$ we have

$$\begin{aligned} |E^G_\Sigma(e)| &> (\mathrm{qd}_1(F) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F)} = \\ & \quad (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma \cup \{x \mapsto e'\})|) \cdot 2^{\mathrm{qd}_2(F_1)}. \end{aligned}$$

We have to show that (A) is equivalent to $G\backslash\{e\} \models_{\sigma \cup \{x \mapsto e'\},\Sigma\backslash\{e\}} F_1$ for all $e' \in E_G\backslash\{e\}$ (Condition (B)).

- We first assume that (A) holds and we show that $G\backslash\{e\} \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma\backslash\{e\}} F_1$ for a fixed $\bar{e} \in E_G\backslash\{e\}$.
  From (A) it follows that $G \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma} F_1$ is satisfied and since e is not in the range of σ and Condition (2) is also satisfied, it follows from the induction hypothesis that $G\backslash\{e\} \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma\backslash\{e\}} F_1$.
- We now assume that (B) holds and we show that $G \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma} F_1$ for a fixed $\bar{e} \in E_G$.
  We distinguish the following two cases:
  - If $\bar{e} \neq e$, Condition (B) implies that $G\backslash\{e\} \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma\backslash\{e\}} F_1$. Then it follows with the induction hypothesis that $G \models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma} F_1$.
  - Now let $\bar{e} = e$ and we assume that $G \not\models_{\sigma \cup \{x \mapsto \bar{e}\},\Sigma} F_1$. Since

    $$|E^G_\Sigma(e)| > (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma \cup \{x \mapsto e\})|) \cdot 2^{\mathrm{qd}_2(F_1)}$$

    and therefore $|E^G_\Sigma(e)| > |\mathrm{dom}(\sigma \cup \{x \mapsto \bar{e}\})|$, it follows that there is an edge $\hat{e} \in E^G_\Sigma(e)$, which is not in the range of $\sigma \cup \{x \mapsto e\}$, i.e. $\hat{e}$ is not in the range of σ and $e \neq \hat{e}$.
    Let π be a permutation on E_G that exchanges e and $\hat{e}$ and is the identity otherwise. Lemma 14.7.1 implies that

    $$G \not\models_{\pi \circ (\sigma \cup \{x \mapsto e\}),\pi \circ \Sigma} F_1.$$

    And since $\pi \circ (\sigma \cup \{x \mapsto e\}) = \sigma \cup \{x \mapsto \hat{e}\}$ and $\pi \circ \Sigma = \Sigma$[2], this implies $G \not\models_{\sigma \cup \{x \mapsto \hat{e}\},\Sigma} F_1$.
    Since now e is not in the range of $\sigma \cup \{x \mapsto \hat{e}\}$ and Condition (2) is also satisfied, it follows with the induction hypothesis that

    $$G\backslash\{e\} \not\models_{\sigma \cup \{x \mapsto \hat{e}\},\Sigma\backslash\{e\}} F_1,$$

    which is a contradiction to Condition (B).

$F = \forall X.F_1$**:** $G \models_{\sigma,\Sigma} \forall X.F_1$ holds if and only if for all $E \subseteq E_G$ we have that $G \models_{\sigma,\Sigma \cup \{X \mapsto E\}} F_1$ (Condition (A)).

We have to show that (A) is equivalent to $G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\} \cup \{X \mapsto E\}} F_1$ for all $E \subseteq E_G\backslash\{e\}$ (Condition (B)).

[2] sloppy for $\{\pi(d) \mid d \in \Sigma(X)\} = \Sigma(X)$. This is true because $\hat{e}$ was chosen from the set $E^G_\Sigma(e)$.

- First assume that (A) holds and show $G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\bar{E}\}} F_1$ for a fixed $\bar{E} \subseteq E_G\backslash\{e\}$.
  We distinguish the following two cases:
  - It holds that $|E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| > (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}$. We know that $G \models_{\sigma,\Sigma\cup\{X\mapsto\bar{E}\}} F_1$ and we can apply the induction hypothesis and obtain $G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\bar{E}\backslash\{e\}\}} F_1$. And since $e \notin \bar{E}$, it follows that $G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\bar{E}\}} F_1$.
  - It holds that $|E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| \leq (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}$. Notice the following:

    $$\begin{aligned} E^G_\Sigma(e) \quad &> \quad (\mathrm{qd}_1(F) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F)} = \\ &\quad\quad (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)} \cdot 2, \end{aligned}$$

    so we are given that

    $$|E^G_\Sigma(e)\backslash E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| > (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}.$$

    Since $e \notin \bar{E}$, it holds that

    $$E^G_{\Sigma\cup\{X\mapsto\bar{E}\cup\{e\}\}}(e) = [E^G_\Sigma(e)\backslash E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)] \cup \{e\}.$$

    Hence

    $$|E^G_{\Sigma\cup\{X\mapsto\bar{E}\cup\{e\}\}}(e)| > (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}.$$

    Since Condition (A) implies that $G \models_{\sigma,\Sigma\cup\{X\mapsto\bar{E}\cup\{e\}\}} F_1$ and Condition (2) is satisfied, the induction hypothesis implies

    $$G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\bar{E}\}} F_1.$$

- We now assume that (B) holds and we show that $G \models_{\sigma,\Sigma\cup\{X\mapsto\bar{E}\}} F_1$ for a fixed $\bar{E} \subseteq E_G$.
  We distinguish the following two cases:
  - It holds that $|E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| > (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}$. We know that $G\backslash\{e\} \models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\bar{E}\backslash\{e\}\}} F_1$ and we can apply the induction hypothesis and obtain $G \models_{\sigma,\Sigma\cup\{X\mapsto\bar{E}\}} F_1$.
  - It holds that $|E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| \leq (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)}$ and we assume that $G \not\models_{\sigma,\Sigma\cup\{X\mapsto\bar{E}\}} F_1$.
    Since

    $$\begin{aligned} E^G_\Sigma(e) \quad &> \quad (\mathrm{qd}_1(F) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F)} \\ &= \quad (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)} \cdot 2, \end{aligned}$$

    it holds that

    $$\begin{aligned} |E^G_\Sigma(e)\backslash E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)| \quad &> \quad (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)} \\ &> \quad |\mathrm{dom}(\sigma)|. \end{aligned}$$

Therefore, there is an edge $\hat{e} \in E^G_\Sigma(e)\backslash E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)$, which is not in the range of σ. We pick a permutation π which exchanges e and $\hat{e}$ and is the identity otherwise.
Since $\pi \circ \sigma = \sigma$ and $\pi \circ (\Sigma \cup \{X \mapsto \bar{E}\}) = \Sigma \cup \{X \mapsto \pi(\bar{E})\}$, it follows from Lemma 14.7.1 that $G \not\models_{\sigma,\Sigma\cup\{X\mapsto\pi(\bar{E})\}} F_1$.
Furthermore

$$E^G_{\Sigma\cup\{X\mapsto\pi(\bar{E})\}}(e) = [E^G_\Sigma(e)\backslash E^G_{\Sigma\cup\{X\mapsto\bar{E}\}}(e)]\backslash\{\hat{e}\} \cup \{e\}$$

and therefore

$$|E^G_{\Sigma\cup\{X\mapsto\pi(\bar{E})\}}(e)| \quad > \quad (\mathrm{qd}_1(F_1) + |\mathrm{dom}(\sigma)|) \cdot 2^{\mathrm{qd}_2(F_1)},$$

so Condition (2) is satisfied and we can apply the induction hypothesis. It follows that

$$G\backslash\{e\} \not\models_{\sigma,\Sigma\backslash\{e\}\cup\{X\mapsto\pi(\bar{E})\backslash\{e\}\}} F_1,$$

which is a contradiction to Condition (B).

□

From Proposition 14.7.2 we infer the following corollary.

Corollary 14.7.3 *Let F be a closed graph formula in $\mathcal{L}2$. Let furthermore G be a graph and $m \in E^\oplus_G$ be a multiset over (the set of edges of) G. Then graph$(m) \models F$ if and only if graph$(m') \models F$, where $m' \in E^\oplus_G$ is defined by $m'(e) = \min\{m(e), \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}\}$.*

Proof: If F has no free variables then $E^{graph(m)}_\Sigma(e) = \{(e,i) \mid 1 \le i \le m(e)\}$. Using Proposition 14.7.2, we can thus reduce the number of copies for every edge to the number $\mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}$, without changing the truth value of F. □

The following corollary shows that every graph-statement of full monadic second-order logic can be encoded into a multiset formula.

Corollary 14.7.4 *Let G be a fixed template graph. A closed graph formula F in $\mathcal{L}2$ can be encoded into a logical formula $M_2(F)$ on multisets as follows. For any multiset $k \in E^\oplus_G$, let C_k be the conjunction over the following formulae:*

- *$\#e = k(e)$ for every $e \in E_G$ satisfying $k(e) < \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}$ and*
- *$\#e \ge k(e)$ for every $e \in E_G$ satisfying $k(e) = \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}$.*

Define $M_2(F)$ to be the disjunction of all C_k such that $k \in E^\oplus_G$, graph$(k) \models F$ and $k(e) \le \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}$ for every $e \in E_G$.

Then graph$(m) \models F \iff m \models M_2(F)$ for every $m \in E^\oplus_G$.

Proof: Let $m \in E^\oplus_G$ be an arbitrary multiset and let m' be a multiset defined as in Corollary 14.7.3, i.e. $m'(e) = \min\{m(e), \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}\}$. for $e \in E_G$.

If $graph(m) \models F$ then, by Corollary 14.7.3, $graph(m') \models F$. Hence, by definition of M_2, $C_{m'}$ appears as a disjunct in $M_2(F)$. Since, clearly, $m \models C_{m'}$, we conclude that $m \models M_2(F)$.

Vice versa, let $m \models M_2(F)$. Then $m \models C_k$ for some $k \in E_G^{\oplus}$ and $graph(k) \models F$. By the shape of C_k, it is immediate to see that this implies $k = m'$. Therefore $graph(m') \models F$, and thus, by Corollary 14.7.3, $graph(m) \models F$. □

Proof: Fix an arbitrary $m \in E_G^{\oplus}$. Let m' be a multiset defined as in Corollary 14.7.3, i.e.

$$m'(e) = \min\{m(e), \mathrm{qd}_1(F) \cdot 2^{\mathrm{qd}_2(F)}\}.$$

First we note that $graph(m) \models C_{m'}$ by the definition of $C_{m'}$. But if the marking $k \in E_G^{\oplus}$ is below the bound given by m' and $k \neq m'$ then C_k is false in $graph(m)$.

Now look at the definition of $M_2(F)$, we throw C_k into the disjunction $M_2(F)$ only if $graph(k) \models F$. So the disjunction $M_2(F)$ holds in $graph(m)$ if and only if F holds in $graph(m')$. But $graph(m') \models F$ if and only if $graph(m) \models F$ by Corollary 14.7.3. Hence $M_2(F)$ is a correct encoding of the formula F. □

To conclude let us show how the general schema outlined at the end of Section 14.2 applies to our running example. We want to verify that Sys satisfies a safety property, i.e., the absence of vicious cycles, including two distinct P_2 processes, in all reachable graphs. Let $\Box L_\mu$ be a fragment of the μ-calculus without negation and "possibility operator" $\Diamond$ (see [LGS+95]), where basic predicates are formulae F taken from our graph logic $\mathcal{L}2$, which can be typed as "reflected by graph morphisms", i.e., such that $F :\leftarrow$ is provable. The property of interest can be expressed in $\Box L_\mu$ as:

$$T_{NC} = \mu\varphi.(NC_{P_2} \wedge \Box\varphi\ )$$

where NC_ℓ is the formula considered in a previous example. Then T_{NC} can be translated into a formula over markings, by translating its graph formula components according to the techniques described in Sections 14.6 and 14.7. This will lead to the formula $M_2(T_{NC}) = \mu\varphi.(M_2(NC_{P_2}) \wedge \Box\varphi\ )$. By the results in this paper and by the results in [2], for T in $\Box L_\mu$, if $\mathcal{U}(\mathsf{Sys}) \models M_2(T)$ then $\mathsf{Sys} \models T$. Therefore the formula T_{NC} can be checked by verifying $M_2(T_{NC})$ on the Petri net component of the approximated unfolding. In this case it can be easily verified that $M_2(T_{NC})$ actually holds in $\mathcal{U}(\mathsf{Sys})$ and thus we conclude that Sys satisfies the desired property.

14.8 Conclusion

We have presented a logic for specifying graph properties, useful for the verification of graph transformation systems. A type system allows us to identify formulae of this logic reflected by edge-bijective morphisms, which can therefore be verified on the covering, i.e., on the finite Petri graph approximation of a GTS. Furthermore we have shown how, given a fixed approximation of the original system, we can perform quantifier-elimination and encode these formulae into boolean combination of atomic predicates on multisets. Combined with the approximated unfolding algorithm of [1], this gives a method for the verification and analysis of graph transformation systems. This form of abstraction is different from the usual forms of abstract interpretation since it abstracts the *structure* of a system rather than its *data*. Maybe the closest relation is shape analysis, abstracting the data structures of a program [NNH99, SRW96].

We would like to add some remarks concerning the practicability of this approach: we are currently developing an implementation of the approximated unfolding algorithm, which inputs and outputs graphs in the Graph Exchange Language (GXL) format, based on XML. It remains to be seen up to which size of a GTS the computation of the approximation is still feasible.

Furthermore encoding a formula into multiset logic may result in a blowup of the size of the formula which is at least exponential. However, provided that formulae are rather small if compared to the size of the system or its approximation, this blowup should be manageable. It is also conceivable to simplify a formula during its encoding (see the example at the end of Section 14.6). The encoding itself is not yet implemented, but we plan to do so in the future.

Finally the Petri net produced by the approximated unfolding algorithm and the formula itself have to be analysed by a model checker or a similar tool, based on the procedures described in [HRY91, HR89, Jan90]. Note that formulae on multisets can not be combined with the temporal operators of CTL* in an arbitrary way. First, we have to make sure that the resulting formula is still reflected, with respect to the simulation, hence no existential path quantification is allowed. Furthermore, arbitrary combinations of the temporal operators "eventually" and "generally" might make the model-checking problem undecidable. However, important fragments are still decidable, for example a property like "all reachable graphs satisfy F", where F is a multiset formula, can be checked. As far as we know, there is not much tool support for model-checking unbounded Petri nets, but these algorithms usually rely on the computation of the coverability graph of a Petri net, which is a well-studied problem [Rei85].

Currently we are mainly interested in proving safety properties, liveness properties require some more care (see [PXZ02]). Another interesting line of future research is to adopt techniques used for the analysis of transition systems specified by integer constraints [Del00].

Acknowledgements: We are very grateful to Andrea Corradini for his contribution to the development of the approximated unfolding technique on which this paper is based. We would also like to thank Ingo Walther who is currently working on an implementation. We are also grateful to the anonymous referees for their valuable comments.

Chapter 15

Verifying Finite-State Graph Grammars: an Unfolding-Based Approach

(Joint work with Paolo Baldan and Andrea Corradini)

Abstract

We propose a framework where behavioural properties of finite-state systems modelled as graph transformation systems can be expressed and verified. The technique is based on the unfolding semantics and it generalises McMillan's complete prefix approach, originally developed for Petri nets, to graph transformation systems. It allows to check properties of the graphs reachable in the system, expressed in a monadic second order logic.

15.1 Introduction

Graph transformation systems (GTSs) are recognised as an expressive specification formalism, properly generalising Petri nets and especially suited for concurrent and distributed systems [EKMR99]: the (topo)logical distribution of a system can be naturally represented by using a graphical structure and the dynamics of the system, e.g., the reconfigurations of its topology, can be modelled by means of graph rewriting rules.

The concurrent behaviour of GTSs has been thoroughly studied and a consolidated theory of concurrency for GTSs is available, including the generalisation of several semantics of Petri nets, like process and unfolding semantics (see, e.g., [CMR96, Rib96, BCM99]). However, only recently, building on these semantical foundations, some efforts have been devoted to the development of frameworks where behavioural properties of GTSs can be expressed and verified (see [GHK98, Kön00b, Hec98, Var02, Ren03, 1]).

As witnessed, e.g., by the approaches in [McM93, Esp94] for Petri Nets, truly concurrent semantics are potentially useful in the verification of finite-state systems, in that they help to avoid the combinatorial explosion arising when one explores all possible interleavings of events. Still, to the best of our knowledge, no technique based on partial order (process or unfolding) semantics has been proposed for the verification of finite-state GTSs.

In this paper we contribute to this topic by proposing a verification framework for *finite-state graph transformation systems* based on their unfolding semantics. Our technique is inspired by the approach originally developed by McMillan for Petri nets [McM93] and further developed by many authors (see, e.g., [Esp94, ERV66, VSY98]). More precisely, our technique applies to any *graph grammar*, i.e., any set of graph rewriting rules with a fixed start graph (the initial state of the system), which is *finite-state* in a liberal sense: the set of graphs which can be reached from the start graph, considered not only *up to isomorphism*, but also *up to isolated nodes*, is finite. Hence in a finite-state graph grammar in our sense there is not actually a bound to the number of nodes generated in a computation, but only to the nodes which are connected to some edge at each stage of the computation. Existing model-checking tools, such as SPIN [Hol97], usually do not directly support the creation of an arbitrary number of objects while still maintaining a finite state space, making entirely non-trivial their use for checking finite-state GTSs (similar problems arise for process calculi agents with name creation).

As a first step we face the problem of identifying a finite, still useful fragment of the unfolding of a GTS. In fact, the unfolding construction for GTSs produces a structure which fully describes the concurrent behaviour of the system, including all possible steps and their mutual dependencies, as well as all reachable states. However, the unfolding is infinite for non-trivial systems, and cannot be used directly for model-checking purposes.

Following McMillan's approach, we show that given any finite-state graph grammar $\mathcal{G}$ a *finite* fragment of its unfolding which is *complete*, i.e., which provides full information about the system as far as reachability (and other) properties are concerned, can be characterised as the maximal prefix of the unfolding not including *cut-off events*. The greater expressiveness of GTSs, and specifically, the possibility of performing "contextual" rewritings (i.e., of preserving part of the state in a rewriting step), a feature which leads to multiple local histories for a single event (see, e.g., the work on contextual nets [PP95, Vog97, BCM01, VSY98]), imposes a generalisation of the original notion of cut-off.

Unfortunately the characterisation of the finite complete prefix is not constructive. Hence, while leaving as an open problem the definition of a general algorithm for constructing such a prefix, we identify a significant subclass of graph grammars where an adaptation of the existing algorithms for Petri nets is feasible. These are called *read-persistent* graph grammars by analogy with the terminology used in the work on contextual nets [VSY98].

In the second part we consider a logic $\mathcal{L}2$ where graph properties of interest can be expressed, like the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (related to security properties) or cycles (related to deadlock-freedom). This is a monadic second-order logic over graphs where quantification is allowed over (sets of) edges. (Similar logics are considered in [Cou97] and, in the field of verification, in [Ren03, 3].) Then we show how a complete finite prefix of a grammar $\mathcal{G}$ can be used to verify properties, expressed in $\mathcal{L}2$, of the graphs reachable in $\mathcal{G}$. This is done by exploiting both the graphical structure underlying the prefix and the concurrency information it provides.

The rest of the paper is organised as follows. Section 15.2 introduces graph transformation systems and their unfolding semantics. Section 15.3 studies finite complete prefixes for finite-state GTSs. Section 15.4 introduces a logic for GTSs, showing how it can be checked over a finite complete prefix. Finally, Section 15.5 draws some conclusions and indicates directions of further research. A more detailed presentation of the material in this paper can be found in [BCK04a].

15.2 Unfolding semantics of graph grammars

This section presents the notion of graph rewriting used in the paper. Rewriting takes place on so-called *typed graphs*, namely graphs labelled over a structure that is itself a graph [CMR96]. It can be seen as a set-theoretical presentation of an instance of algebraic (single- or double-pushout) rewriting (see, e.g., [CMR+97]). Next we review the notion of occurrence grammar, which is instrumental in defining the unfolding of a graph grammar [BCM99, Rib96].

15.2.1 Graph Transformation Systems

In the following, given a set A we denote by A^* the set of finite strings of elements of A. Given $u \in A^*$ we write $|u|$ to indicate the length of u. If $u = a_0 \dots a_n$ and $0 \leq i \leq n$, by $[u]_i$ we denote the i-th element a_i of u. Furthermore, if $f : A \to B$ is a function then we denote by $f^* : A^* \to B^*$ its extension to strings.

A *(hyper)graph* G is a tuple (V_G, E_G, c_G), where V_G is a set of nodes, E_G is a set of edges and $c_G : E_G \to {V_G}^*$ is a connection function. A node $v \in V_G$ is called *isolated* if it is not connected to any edge. Given two graphs G, G', a *graph morphism* $\varphi : G \to G'$ is a pair $\langle \varphi_V : V_G \to V_{G'},\ \varphi_E : E_G \to E_{G'} \rangle$ of total functions such that for all $e \in E_G$, ${\varphi_V}^*(c_G(e)) = c_{G'}(\varphi_E(e))$. When obvious from the context, the subscripts V and E will be omitted.

Definition 15.2.1 (typed graph) *Given a graph (of types) T, a* typed graph *G over T is a graph $|G|$, together with a morphism $type_G : |G| \to T$. A* morphism *between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with the typing, i.e., such that $type_{G_1} = type_{G_2} \circ f$.*

A typed graph G is called *injective* if the typing morphism $type_G$ is injective. More generally, given $n \in \mathbb{N}$, the graph is called *n-injective* if for any item x in T, $|type_G^{-1}(x)| \leq n$, namely if the number of "instances of resources" of any type x is bounded by n. Given two (typed) graphs G and G' we will write $G \simeq G'$ to mean that G and G' are *isomorphic*, and $G \ddot{\simeq} G'$ when G and G'

are *isomorphic up to isolated nodes*, i.e., once their isolated nodes have been removed.

In the sequel we extensively use the fact that given a graph G, any subgraph of G without isolated nodes is identified by the set of its edges. Precisely, given a subset of edges $X \subseteq E_G$, we denote by $graph(X)$ the least subgraph of G (actually the unique subgraph, up to isolated nodes) having X as set of edges.

We will use some set-theoretical operations on (typed) graphs with "componentwise" meaning. Let G and G' be T-typed graphs. We say that G and G' are *consistent* if $G \cup G'$ defined as $(V_{|G|} \cup V_{|G'|}, E_{|G|} \cup E_{|G'|}, c_G \cup c_{G'})$, typed by $type_G \cup type_{G'}$, is a well-defined T-typed graph. In this case also the intersection $G \cap G'$, constructed in a similar way, is well-defined. Given a graph G and a set (of edges) E we denote by $G - E$ the graph obtained from G by removing the edges in E. Sometimes we will also refer to the items (nodes and edges) in $G - G'$, where G and G' are graphs, although the structure resulting as the componentwise set-difference of G and G' might not be a well-defined graph.

Definition 15.2.2 (production) *Given a graph of types T, a* T-typed production *is a pair of finite consistent T-typed graphs $q = (L, R)$, often written $L \to R$, such that 1) $L \cup R$ and L do not include isolated nodes; 2) $V_{|L|} \subseteq V_{|R|}$; and 3) $E_{|L|} - E_{|R|}$ and $E_{|R|} - E_{|L|}$ are non-empty.*

A rule $L \to R$ specifies that, once an occurrence of L is found in a graph G, then G can be rewritten by removing (the images in G of) the items in $L - R$ and adding those in $R - L$. The (images in G of the) items in $L \cap R$ instead are left unchanged: they are, in a sense, preserved or read by the rewriting step.

This informal explanation should also motivate Conditions 1–3 above. Condition 1 essentially states that we are interested only in rewriting up to isolated nodes: by the requirement on $L \cup R$, no node is isolated when created and, by the requirement on L, nodes that become isolated have no influence on further reductions. Thus one can safely assume that isolated nodes are removed by some kind of garbage collection. Consistently with this view, by Condition 2 productions cannot delete nodes (deletion can be simulated by leaving that node isolated). Condition 3 ensures that every production consumes and produces at least one edge: a requirement corresponding to T-restrictedness in Petri net theory.

Definition 15.2.3 (graph rewriting) *Let $q = L \to R$ be a T-typed production. A* match *of q in a T-typed graph G is a morphism $\varphi : L \to G$, satisfying the* identification condition, *i.e., for $e, e' \in E_{|L|}$, if $\varphi(e) = \varphi(e')$ then $e, e' \in E_{|R|}$. In this case G rewrites to the graph H, obtained as $H = ((G - \varphi(E_{|L|} - E_{|R|})) \uplus R)/_{\equiv}$, where $\equiv$ is the least equivalence on the items of the graph such that $x \equiv \varphi(x)$. We write $G \Rightarrow_{q,\varphi} H$ or simply $G \Rightarrow_q H$.*

A rewriting step is schematically represented in Fig. 15.1. Intuitively, in the graph $H' = G - \varphi(E_{|L|} - E_{|R|})$ the images of all the edges in $L - R$ have been removed. Then in order to get the resulting graph, merge R to H' along the image through φ of the preserved subgraph $L \cap R$. Formally the resulting graph H is obtained by first taking $H' \uplus R$ and then by identifying, via the equivalence

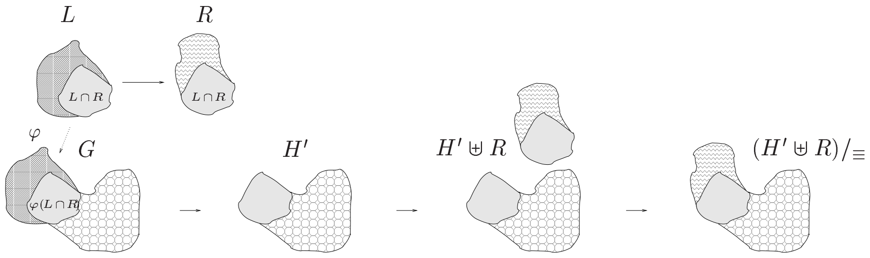


Figure 15.1: A rewriting step, schematically.

$\equiv$, the image through φ of each item in $L \cap R$ with the corresponding item in R.

Definition 15.2.4 (graph transformation system and graph grammar) *A* graph transformation system (GTS) *is a triple $\mathcal{R} = \langle T, P, \pi \rangle$, where T is a graph of types, P is a set of production names and π is a function mapping each production name $q \in P$ to a T-typed production $\pi(q) = L_q \to R_q$. A* graph grammar *is a tuple $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ where $\langle T, P, \pi \rangle$ is a GTS and G_s is a finite T-typed graph, without isolated nodes, called the* start graph*. We denote by $Elem(\mathcal{G})$ the (disjoint) union $E_T \uplus P$, i.e., the set of edges in the graph of types and the production names. We call $\mathcal{G}$* finite *if the set $Elem(\mathcal{G})$ is finite.*

A T-typed graph G is *reachable* in $\mathcal{G}$ if $G_s \Rightarrow^*_{\mathcal{G}} G'$ for some $G' \simeq G$, where $\Rightarrow^*_{\mathcal{G}}$ is the transitive closure of the rewriting relation induced by productions in $\mathcal{G}$.

We remark that Place/Transition Petri nets can be viewed as a special subclass of typed graph grammars. Say that a graph G is *edge-discrete* if its set of nodes is empty (and thus edges have no connections). Given a P/T net P, let T_P be the edge-discrete graph having the set of places of P as edges. Then any finite edge-discrete graph typed over T_P can be seen as a marking of P: an edge typed over s represents a token in place s. Using this correspondence, a production $L_t \to R_t$ faithfully represents a transition t of P if L_t encodes the marking *pre-set*(t), R_t encodes *post-set*(t), and $L_t \cap R_t = \emptyset$. The graph grammar corresponding to a Petri net is finite iff the original net has finitely many places and transitions. Observe that the generalisation from edge-discrete to proper graphs radically changes the expressive power of the formalism. For instance, unlike P/T Petri nets, the class of grammars in this paper is Turing complete.

Example 15.2.5 Consider the graph grammar $\mathcal{CP}$, modeling a system where three *processes* of type P are connected to a *communication manager* of type CM (see the start graph in Fig. 15.2, where edges are represented as rectangles and nodes as small circles). Two processes may establish a new connection with each other via the communication manager, becoming *processes engaged* in communication (typed PE, the only edge with more than one connection). This transformation is modelled by the production [engage] in Fig. 15.2: observe that a new node connecting the two processes is created. The second production [release] terminates the communication between two partners. A

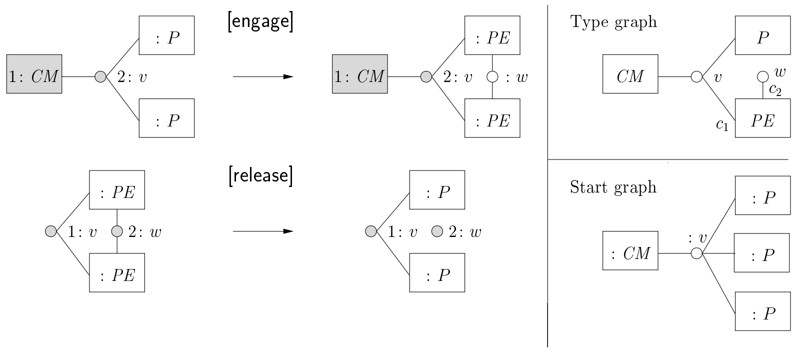


Figure 15.2: The finite-state graph grammar $\mathcal{CP}$.

typed graph G over $T_{\mathcal{CP}}$ is drawn by labeling each edge or node x of G with "$: type_G(x)$". Only when the same graphical item x belongs to both the left- and the right-hand side of a production we include its identity in the label (which becomes "$x : type_G(e)$"): in this case we also shade the item, to stress that it is preserved by the production.

The notion of safety for graph grammars [CMR96] generalises the one for P/T nets which requires that each place contains at most one token in any reachable marking. More generally, we extend to graph grammars the notion of n-boundedness.

Definition 15.2.6 (bounded/safe grammar) *For a fixed $n \in \mathbb{N}$, we say that a graph grammar $\mathcal{G}$ is n-*bounded *if for all graphs H reachable in $\mathcal{G}$ there is an n-injective graph H' such that $H' \mathrel{\ddot{\simeq}} H$. A 1-bounded grammar will be called* safe.

The definition can be understood by thinking of edges of the graph of types T as a generalisation of places in Petri nets. In this view the number of different edges of a graph which are typed on the same item of T corresponds to the number of tokens contained in a place. Observe that for *finite* graph grammars, n-boundedness amounts to the property of being finite-state (up to isomorphism and up to isolated nodes). In the sequel when considering a finite-state graph grammar we will (often implicitly) assume that it is also finite.

For instance, the graph grammar $\mathcal{CP}$ in Fig. 15.2 is clearly 3-bounded and thus finite-state (but only up to isolated nodes).

15.2.2 Nondeterministic Occurrence Grammars

When a graph grammar $\mathcal{G}$ is safe, and thus reachable graphs are injectively typed, at every step, for any item t in the type graph every production can consume, preserve and produce a single item typed t. Hence we can safely think that a production, according to its typing, *consumes*, *preserves* and *produces* items of the graph of types. Using a net-like language, we speak of *pre-set* ${}^\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a production q. Since we work with graphs considered up to isolated nodes, we will record in these sets only edges. Formally, for any production q of a graph grammar $\mathcal{G} = \langle T, G_s, P, \pi \rangle$, we define

$$^\bullet q = type_{L_q}(E_{|L_q|} - E_{|R_q|}) \quad \underline{q} = type_{L_q}(E_{|L_q \cap R_q|}) \quad q^\bullet = type_{R_q}(E_{|R_q|} - E_{|L_q|})$$

Furthermore, for any edge e in T we define $^\bullet e = \{q \in P : e \in q^\bullet\}$, $\underline{e} = \{q \in P : e \in \underline{q}\}$, $e^\bullet = \{q \in P : e \in {}^\bullet q\}$. This notation is extended also to nodes in the obvious way, e.g., for $v \in V_T$ we define $^\bullet v = \{q \in P : v \in type_{R_q}(V_{|R_q|} - V_{|L_q|})\}$.

An example of safe grammar can be found in Fig. 15.3 (for the moment ignore its relation to grammar $\mathcal{CP}$ in Fig. 15.2). For this grammar, $^\bullet\mathsf{engage1} = \{2\colon P, 3\colon P\}$, $\underline{\mathsf{engage1}} = \{1\colon CM\}$ and $\mathsf{engage1}^\bullet = \{5\colon PE, 6\colon PE\}$, while $^\bullet 1\colon CM = \emptyset$, $\underline{1\colon CM} = \{\mathsf{engage1}, \mathsf{engage2}, \mathsf{engage3}\}$ and $3\colon P^\bullet = \{\mathsf{engage1}, \mathsf{engage3}\}$.

Definition 15.2.7 (causal relation) *The* causal relation *of a safe grammar $\mathcal{G}$ is the least transitive relation $<$ over $Elem(\mathcal{G})$ satisfying, for any edge e in the graph of types T, and for productions $q, q' \in P$:*

1. $e \in {}^\bullet q \;\Rightarrow\; e < q$; *2.* $e \in q^\bullet \;\Rightarrow\; q < e$; *3.* $q^\bullet \cap \underline{q'} \neq \emptyset \;\Rightarrow\; q < q'$.

As usual $\leq$ is the reflexive closure of $<$. Moreover, for $x \in Elem(\mathcal{G})$ we denote by $\lfloor x \rfloor$ the set of causes of x in P, namely $\lfloor x \rfloor = \{q \in P : q \leq x\}$.

Note that the fact that an item is preserved by q and consumed by q', i.e., $\underline{q} \cap {}^\bullet q' \neq \emptyset$ does not imply $q < q'$. In this case, the dependency between the two productions is a kind of *asymmetric conflict* (see [BCM01, PP95, Lan92, VSY98]): The application of q' prevents q from being applied, so that q can never follow q' in a derivation (or, equivalently, if both q and q' occur in a derivation then q must precede q').

Definition 15.2.8 (asymmetric conflict) *The* asymmetric conflict $\nearrow$ *of a safe grammar $\mathcal{G}$ is the relation over the set of productions P, defined by $q \nearrow q'$ if:*

1. $\underline{q} \cap {}^\bullet q' \neq \emptyset$; *2.* $^\bullet q \cap {}^\bullet q' \neq \emptyset$ *and* $q \neq q'$; *3.* $q < q'$.

Condition 1 is justified by the discussion above. Condition 2 essentially expresses the fact that the ordinary symmetric conflict is encoded, in this setting, as an asymmetric conflict in both directions. More generally, we will write $q \# q'$ and say that q and q' are in *conflict* when the causes of q and q', i.e., $\lfloor q \rfloor \cup \lfloor q' \rfloor$, includes a cycle of asymmetric conflict. Finally, since $<$ represents a global order of execution, while $\nearrow$ determines an order of execution only locally to each computation, it is natural to impose $\nearrow$ to be an extension of $<$ (Condition 3).

Definition 15.2.9 ((nondeterministic) occurrence grammar) *A* (nondeterministic) occurrence grammar *is a safe grammar $\mathcal{O} = \langle T, G_s, P, \pi \rangle$ such that*

1. *$\leq$ is a partial order; for any $q \in P$, $\lfloor q \rfloor$ is finite and $\nearrow$ is acyclic on $\lfloor q \rfloor$;*
2. *G_s is the graph $graph(Min(\mathcal{O}))$ generated by the set $Min(\mathcal{O})$ of minimal elements of $\langle Elem(\mathcal{O}), \leq \rangle$, typed over T by the inclusion;*

3. *any item x in T is created by at most one production in P, i.e., $|\ {}^{\bullet}x\ | \leq 1$;*

4. *for each $q \in P$, the typing $type_{L_q}$ is injective on the "consumed" items in $|L_q| - |R_q|$, and $type_{R_q}$ is injective on the "produced" items in $|R_q| - |L_q|$.*

Since the start graph of an occurrence grammar $\mathcal{O}$ is determined by $Min(\mathcal{O})$, we often do not mention it explicitly.

Intuitively, Conditions 1–3 recast in the framework of graph grammars the conditions of occurrence nets (actually of occurrence contextual nets [BCM01, VSY98]). In particular, in Condition 1, the acyclicity of asymmetric conflict on $\lfloor q \rfloor$ corresponds to the requirement of irreflexivity for the conflict relation in occurrence nets. Condition 4, instead, is closely related to safety and requires that each production consumes and produces items with multiplicity one. An example of an occurrence grammar is given in Fig. 15.3.

15.2.3 Concurrent Subgraphs, Configurations and Histories

The finite computations of an occurrence grammar are characterised by special subsets of productions closed under causal dependencies and with no conflicts (i.e., cycles of asymmetric conflict), suitably ordered.

Definition 15.2.10 (configuration) *Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A configuration of $\mathcal{O}$ is a finite subset of productions $C \subseteq P$ such that $\nearrow_C$ (the asymmetric conflict restricted to C) is acyclic, and for any $q \in C$, $\lfloor q \rfloor \subseteq C$. Given two configurations C_1, C_2 we write $C_1 \sqsubseteq C_2$ if $C_1 \subseteq C_2$ and for any $q_1 \in C_1$, $q_2 \in C_2$, if $q_2 \nearrow q_1$ then $q_2 \in C_1$.*

The set of all configurations of $\mathcal{O}$, ordered by $\sqsubseteq$, is denoted by $Conf(\mathcal{O})$.

Proposition 15.2.11 (reachability of graphs generated by configurations) *Let $\mathcal{O}$ be an occurrence grammar, $C \in Conf(\mathcal{O})$ be a configuration and*

$$\mathbf{G}(C) = graph((Min(\mathcal{O}) \cup \textstyle\bigcup_{q \in C} q^{\bullet}) - \bigcup_{q \in C} {}^{\bullet}q).$$

Then a graph G such that $G \ddot{\simeq} \mathbf{G}(C)$ can be obtained from the start graph of $\mathcal{O}$, by applying all the productions in C in any order compatible with $\nearrow$.

Due to the presence of asymmetric conflicts, given a production q, the history of q, i.e., the set of events which must precede q in a computation is not uniquely determined by q, but it depends also on the particular computation: the history of q can or can not include the productions in asymmetric conflict with q.

Definition 15.2.12 (history) *Let $\mathcal{O}$ be an occurrence grammar, let $C \in Conf(\mathcal{O})$ be a configuration and let $q \in C$. The* history *of q in C is the set of events $C[\![q]\!] = \{q' \in C : q' \nearrow_C^* q\}$. We denote by $Hist(q)$ the set of histories of q, i.e., $Hist(q) = \{C[\![q]\!] : C \in Conf(\mathcal{O})\}$.*

Reachable states can be characterised in terms of a concurrency relation.

Definition 15.2.13 (concurrent graph) *Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A finite subset of edges $E \subseteq E_T$ is called* concurrent, *written $co(E)$, if*

1. *$\nearrow_E$, the asymmetric conflict restricted to $\bigcup_{x \in E} \lfloor x \rfloor$, is acyclic;*
2. *$\neg(x < y)$ for all $x, y \in E$.*

A subgraph G of T is called concurrent, written $co(G)$, if $co(E_G)$.

It can be shown that the maximal concurrent subgraphs G of T correspond exactly (up to isolated nodes) to the graphs reachable from the start graph.

15.2.4 Unfolding of graph grammars

The unfolding construction, when applied to a grammar $\mathcal{G}$, produces a nondeterministic occurrence grammar $\mathcal{U}(\mathcal{G})$ describing the behaviour of $\mathcal{G}$. A construction for the double-pushout algebraic approach to graph rewriting has been proposed in [BCM99]: the one sketched here is simpler because productions cannot delete nodes and thus the dangling edge condition does not play a role.

The construction begins from the start graph of $\mathcal{G}$, and then applies in all possible ways its productions to concurrent subgraphs, recording in the unfolding each occurrence of production and each new graph item generated in the rewriting process.

Definition 15.2.14 (unfolding - sketch) *Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. The unfolding $\mathcal{U}(\mathcal{G}) = \langle T', G'_s, P', \pi' \rangle$ is the "componentwise" union of the following inductively defined sequence of occurrence grammars $\mathcal{U}(\mathcal{G})^{[n]}$.*

$(\mathbf{n} = \mathbf{0})$ *$\mathcal{U}(\mathcal{G})^{[0]}$ consists of the start graph $|G_s|$, with no productions.*

$(\mathbf{n} \to \mathbf{n}+\mathbf{1})$ *Take $q \in P$ and let m be a match of q in the graph of types of $\mathcal{U}(\mathcal{G})^{[n]}$, satisfying the identification condition, such that $m(|L_q|)$ is* concurrent.

Then the occurrence grammar $\mathcal{U}(\mathcal{G})^{[n+1]}$ is obtained by "recording" in $\mathcal{U}(\mathcal{G})^{[n]}$ the application of q at the match m. More precisely, a new production $q' = \langle q, m \rangle$ is added and the graph of types $T^{[n]}$ is extended by adding to it a copy of each item generated by the application q, without deleting any item.

The unfolding is mapped over the original grammar by the so-called folding morphism $\chi = \langle \chi_T, \chi_P \rangle : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. The first component $\chi_T : T' \to T$ is a graph morphism mapping each graph item in the (graph of types of) the unfolding to the corresponding item in the (graph of types of) the original grammar $\mathcal{G}$. The second component $\chi_P : P' \to P$ maps any production occurrence $\langle q, m \rangle$ in the unfolding to the corresponding production q of $\mathcal{G}$.

The occurrence grammar in Fig. 15.3 is an initial part of the (infinite) unfolding of the grammar $\mathcal{CP}$ in Fig. 15.2. For instance, production engage1 is an occurrence of production engage in $\mathcal{CP}$, applied at the match consisting of the edges $1\colon CM$, $2\colon P$, $3\colon P$. Unfolding such a match, three new graph items, two edges $5\colon PE$, $6\colon PE$ and a node, are added to the graph of types of the unfolding. Note that the graph of types of the (partial) unfolding (call it $T_{\mathcal{T}}$) is typed over the graph of types $T_{\mathcal{CP}}$ of the original grammar (via the folding morphism $\chi_T : T_{\mathcal{T}} \to T_{\mathcal{CP}}$). This explains why the edges of the graphs in the productions of the unfolding, which are typed over $T_{\mathcal{T}}$, are marked with names including two colons.

The unfolding provides a compact representation of the behaviour of $\mathcal{G}$, and in particular it represents all the graphs reachable in $\mathcal{G}$, in the following sense.

Theorem 15.2.15 (completeness of the unfolding) *Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. A T-typed graph G is reachable in $\mathcal{G}$ iff there exists a maximal concurrent subgraph X' of the graph of types of $\mathcal{U}(\mathcal{G})$ such that $G \simeq \langle X', \chi_{T|X'} \rangle$.*

15.3 Finite Prefix for Graph Grammars

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ denote a graph grammar, fixed throughout the section, and let $\mathcal{U}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be its unfolding with $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$ the folding morphism, as in Definition 15.2.14. Given a configuration C of $\mathcal{U}(\mathcal{G})$, recall from Proposition 15.2.11 that $\mathbf{G}(C)$ denotes the subgraph of T' reached after the execution of the productions in C (up to isolated nodes). We shall denote by $Reach(C)$ the same graph, seen as a graph typed over T by the restriction of the folding morphism, i.e., $Reach(C) = \langle \mathbf{G}(C), \chi_{T|\mathbf{G}(C)} \rangle$.

To identify a finite prefix of the unfolding the idea consists of avoiding to keep in the unfolding useless productions, i.e., productions which do not contribute to generating new graphs. The definition of "cut-off event" introduced by McMillan for Petri nets in order to formalise such a notion has to be adapted to this context, since for graph grammars a production may have different histories.

Definition 15.3.1 (cut-off) *A production $q \in P'$ of the unfolding $\mathcal{U}(\mathcal{G})$ is a* cut-off *if there exists $q' \in P'$ such that $Reach(\lfloor q \rfloor) \simeq Reach(\lfloor q' \rfloor)$ and $|\lfloor q' \rfloor| < |\lfloor q \rfloor|$.*

A production q is a strong cut-off *if for all $C_q \in Hist(q)$ there exist $q' \in P'$ and $C_{q'} \in Hist(q')$ such that $Reach(C_q) \simeq Reach(C_{q'})$ and $|C_{q'}| < |C_q|$. The* truncation *of $\mathcal{G}$ is the greatest prefix $\mathcal{T}(\mathcal{G})$ of $\mathcal{U}(\mathcal{G})$ not including strong cut-offs.*

Theorem 15.3.2 (completeness and finiteness of the truncation) *The truncation $\mathcal{T}(\mathcal{G})$ is a complete prefix of the unfolding, i.e., for any reachable graph G of $\mathcal{G}$ there is a configuration C in $Conf(\mathcal{T}(\mathcal{G}))$ such that $Reach(C) \ddot{\simeq} G$. Furthermore, if $\mathcal{G}$ is n-bounded then the truncation $\mathcal{T}(\mathcal{G})$ is finite.*

Unfortunately, the proof of the above theorem does not suggest a way of constructing the truncation for finite-state graph grammars. The problem essentially resides in the fact that the notion of strong cut-off refers to the set of histories of a production, which is, in general, infinite. While leaving the solution for the general case as an open problem, we next discuss how a finite complete prefix can be derived for a class of grammars for which this problem disappears. This still interesting class of graph grammars is characterised by a property that we call "read-persistence", since it appears as the graph grammar theoretical version of read-persistence as defined for contextual nets [VSY98].

Definition 15.3.3 (read-persistence) *An occurrence grammar $\mathcal{O} = \langle T, P, \pi \rangle$ is called* read-persistent *if for any $q_1, q_2 \in P$, if $q_1 \nearrow q_2$ then $q_1 \leq q_2$ or $q_1 \# q_2$. A graph grammar $\mathcal{G}$ is called* read-persistent *if its unfolding $\mathcal{U}(\mathcal{G})$ is read-persistent.*

It can be shown that an adaptation of the algorithm originally proposed in [McM93] for ordinary nets and extended in [VSY98] to read-persistent contextual nets, works for read-persistent graph grammars. In particular, the notion of strong cut-off can be safely replaced by the weaker notion of (ordinary) cut-off. An obvious class of read-persistent graph grammars consists of all the grammars $\mathcal{G}$ where any edge preserved by productions is never consumed.

For instance, the grammar $\mathcal{CP}$ in our running example is read-persistent, since the communication manager CM, the only edge preserved by productions, is never consumed. Its truncation is the graph grammar $\mathcal{T}(\mathcal{CP})$ depicted in Fig. 15.3. Denote by $T_{\mathcal{T}}$ its type graph. Note that applying the production [release] to any subgraph of $T_{\mathcal{T}}$ matching its left-hand side would result in a cut-off: this is the reason why $\mathcal{T}(\mathcal{CP})$ does not include any instance of production [release]. The start graph of the truncation is isomorphic to the start graph of grammar $\mathcal{CP}$ and it is mapped injectively to the graph of types $T_{\mathcal{T}}$ in the obvious way.

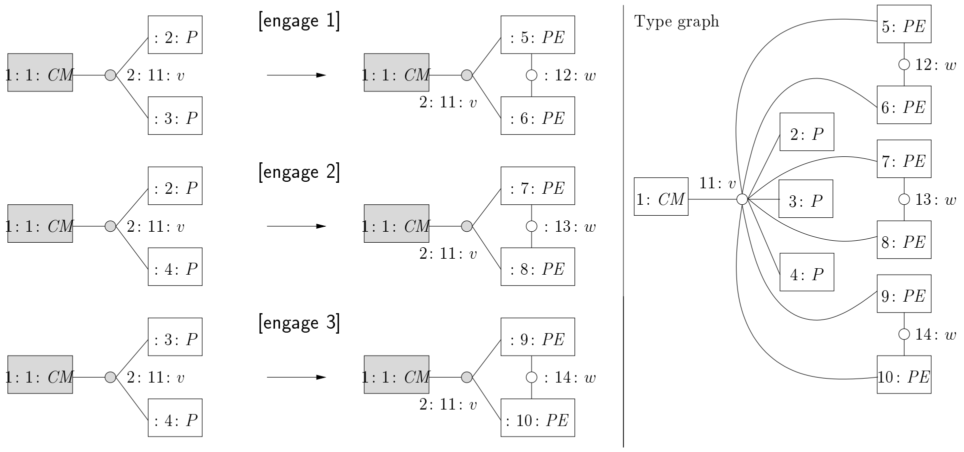


Figure 15.3: The truncation $\mathcal{T}(\mathcal{CP})$ of the graph grammar in Fig. 15.2.

In general, the truncation of a grammar such as $\mathcal{CP}$ where n processes are connected to CM in the start graph, will contain $\frac{n(n-1)}{2}$ productions. Considering instead all possible interleavings, we would end up with an exponential number of productions.

15.4 Exploiting the prefix

In this section we propose a monadic second-order logic $\mathcal{L}2$ where some graph properties of interest can be expressed. Then we show how the validity of a property in $\mathcal{L}2$ over all the reachable graphs of a finite-state grammar $\mathcal{G}$ can be verified by exploiting a complete finite prefix.

15.4.1 A logic on graphs

We first introduce the monadic second order logic $\mathcal{L}2$ for specifying graph properties. Quantification is allowed over edges, but not over nodes (as, e.g., in [Cou97]).

Definition 15.4.1 (Graph formulae) *Let $\mathcal{X}_1 = \{x, y, \ldots\}$ be a set of (first-order) edge variables and let $\mathcal{X}_2 = \{X, Y, \ldots\}$ be a set of (second-order) variables representing edge sets. The set of* graph formulae *of the logic $\mathcal{L}2$ is defined as follows, where $\ell \in \Lambda$, $i, j \in \mathbb{N}$:*

$$\begin{array}{lll} F & ::= & x = y \ \mid\ c_i(x) = c_j(y) \ \mid\ \mathit{type}(x) = \ell \ \mid\ x \in X \qquad \textit{(Predicates)} \\ & & F \vee F \ \mid\ \neg F \ \mid\ \exists x.F \ \mid \exists X.F \qquad \textit{(Connectives / Quantifiers)} \end{array}$$

We denote by $\mathit{free}(F)$ and $\mathit{Free}(F)$ the sets of first-order and second-order variables, respectively, occurring free in F, defined in the obvious way.

Given a T-typed graph G, a formula F in $\mathcal{L}2$, and two valuations $\sigma : \mathit{free}(F) \to E_{|G|}$ and $\Sigma : \mathit{Free}(F) \to \mathcal{P}(E_{|G|})$ for the free first- and second-order variables of F, respectively, the *satisfaction relation* $G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance $G \models_{\sigma,\Sigma} x = y$ iff $\sigma(x) = \sigma(y)$ and $G \models_{\sigma,\Sigma} x \in X$ iff $\sigma(x) \in \Sigma(X)$.

A simple, but fundamental observation is that, while for n-bounded graph grammars the graphical nature of the state plays a basic role, for any occurrence grammar $\mathcal{O}$ we can can forget about it and view $\mathcal{O}$ as an occurrence contextual net (i.e., a Petri net with read arcs, see, e.g., [BCM01, VSY98]).

Definition 15.4.2 (Petri net underlying a graph grammar) *The* contextual Petri net underlying *an occurrence grammar $\mathcal{O} = \langle T', P', \pi' \rangle$, denoted by* $\mathsf{Net}(\mathcal{O})$*, is the Petri net having the set of edges $E_{T'}$ as places and a transition for every production $q \in P'$, with pre-set ${}^\bullet q$, post-set $q^\bullet$ and context $\underline{q}$.*

For instance, the Petri net $\mathsf{Net}(\mathcal{T}(\mathcal{CP}))$ underlying the truncation of $\mathcal{CP}$ (see Fig. 15.3) is depicted in Fig. 15.4. Read arcs are represented as dotted undirected lines.

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a fixed finite-state graph grammar and consider the truncation $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ (actually, all the results hold for any complete finite prefix of the unfolding). Notice that, by completeness of $\mathcal{T}(\mathcal{G})$, any graph reachable in $\mathcal{G}$ is (up to isolated nodes) a subgraph of the graph of types T' of $\mathcal{T}(\mathcal{G})$, typed over T by the restriction of the folding morphism $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. Also observe that a safe marking m of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ can be seen as a graph typed over the type graph T of the original grammar $\mathcal{G}$: take the least subgraph of T'

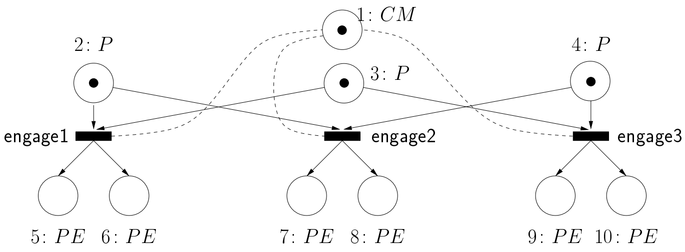


Figure 15.4: The Petri net underlying the truncation $\mathcal{T}(\mathcal{CP})$ in Fig. 15.3

having m as set of edges, i.e., $graph(m)$, and type it over T by the restriction of the folding morphism. With a slight abuse of notation this typed graph will be denoted simply as $graph(m)$.

We show how any formula φ in $\mathcal{L}2$ can be translated to a formula $M(\varphi)$ over the safe markings of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ such that, for any marking m *reachable* in $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$

$$graph(m) \models \varphi \quad \text{iff} \quad m \models M(\varphi).$$

The syntax of the formulae over markings is

$$\varphi ::= e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi,$$

where the basic formulae e are place (edge) names, meaning that the place is marked, i.e., $m \models e$ if $e \in m$. Logical connectives are treated as usual.

Definition 15.4.3 (Encoding graph into multiset formulae) *Let $\mathcal{T}(\mathcal{G})$ be the truncation of a graph grammar $\mathcal{G}$, as above. Let F be graph formula in $\mathcal{L}2$, let $\sigma : free(F) \to E_{T'}$ and $\Sigma : Free(F) \to \mathcal{P}(E_{T'})$. The* encoding M *is defined as:*

$$\begin{array}{rcl}
M[x = y, \sigma, \Sigma] & = & \textit{true if } \sigma(x) = \sigma(y) \textit{ and false otherwise} \\
M[c_i(x) = c_j(y), \sigma, \Sigma] & = & \left\{ \begin{array}{ll} \textit{true} & \textit{if } |c_{T'}(\sigma(x))| \geq i \wedge |c_{T'}(\sigma(y))| \geq j \\ & \quad \wedge\ [c_{T'}(\sigma(x))]_i = [c_{T'}(\sigma(y))]_j \\ \textit{false} & \textit{otherwise} \end{array} \right. \\
M[type(x) = \ell, \sigma, \Sigma] & = & \textit{true if } \chi_T(\sigma(x)) = \ell \textit{ and false otherwise} \\
M[x \in X, \sigma, \Sigma] & = & \textit{true if } \sigma(x) \in \Sigma(X) \textit{ and false otherwise} \\
M[F_1 \vee F_2, \sigma, \Sigma] & = & M[F_1, \sigma, \Sigma] \vee M[F_2, \sigma, \Sigma] \\
M[\neg F, \sigma, \Sigma] & = & \neg M[F, \sigma, \Sigma] \\
M[\exists x.F, \sigma, \Sigma] & = & \bigvee_{e \in E_{T'}} (e \ \wedge \ M[F, \sigma \cup \{x \mapsto e\}, \Sigma]) \\
M[\exists X.F, \sigma, \Sigma] & = & \bigvee_{E \subseteq E_{T'}, co(E)} (\bigwedge E \ \wedge \ M[F, \sigma, \Sigma \cup \{X \mapsto E\}])
\end{array}$$

where, for $E = \{e_1, \ldots, e_n\}$, the symbol $\bigwedge E$ stands for $e_1 \wedge \ldots \wedge e_n$. If F is closed formula (i.e., without free variables), we define $M[F] = M[F, \emptyset, \emptyset]$.

Note that, since every reachable graph in $\mathcal{G}$ is isomorphic to a subgraph of T', typed by the restriction of χ_T, the encoding resolves the basic predicates by exploiting the structural information of T'. When a first-order variable x in a formula is mapped to an edge e, we take care that the edge is marked, and, similarly, when a second-order variable X in a formula is mapped to a set of edges E, such a set must be covered. Observe that in this case E is limited to range only over concurrent subsets of edges. In fact, if E is a non-concurrent set, then no reachable marking m will include E, i.e., $m \not\models \bigwedge E$.

It is possible to show that the above encoding is correct, i.e., for any formula $\varphi \in \mathcal{L}2$, for any pair of valuations $\sigma : \mathcal{X}_1 \to E_{T'}$ and $\Sigma : \mathcal{X}_2 \to \mathcal{P}(E_{T'})$, and for any safe marking m over $E_{T'}$, we have $\mathit{graph}(m) \models_{\sigma,\Sigma} \varphi$ iff $m \models M[\varphi, \sigma, \Sigma]$.

15.4.2 Checking properties of reachable graphs

Let $\mathcal{G} = \langle G_s, T, P, \pi \rangle$ be a finite-state graph grammar. We next show how a complete finite prefix of $\mathcal{G}$ can be used to check whether, given a formula $F \in \mathcal{L}2$, there exists some reachable graph which satisfies F. In this case we will write $\mathcal{G} \models \Diamond F$. The same algorithm allows to check "invariants" of a graph grammars, i.e., to verify whether a property $F \in \mathcal{L}2$ is satisfied by all graphs reachable in $\mathcal{G}$, written $\mathcal{G} \models \Box F$. In fact, it trivially holds that $\mathcal{G} \models \Box F$ iff $\mathcal{G} \not\models \Diamond \neg F$.

Let $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be the truncation of $\mathcal{G}$ (or any complete prefix of the unfolding) and let $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ be the underlying Petri net. The formula produced by the encoding in Definition 15.4.3 can be simplified by exploiting the mutual relationships between items as expressed by the causality, (asymmetric) conflict and concurrency relation.

Proposition 15.4.4 (simplification) *Let F be any formula in $\mathcal{L}2$, let $\sigma : \mathit{free}(F) \to E_{T'}$ and $\Sigma : \mathit{Free}(F) \to \mathcal{P}(E_{T'})$ be valuations. If m is a marking reachable in $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ and η is a marking formula obtained by simplifying $M[F, \sigma, \Sigma]$ with the* Simplification Rule *below:*

> *If $S \subseteq E_{T'}$ and $\neg co(S)$ then replace the subformula $\bigwedge S$ by false.*

then $\mathit{graph}(m) \models_{\sigma,\Sigma} F$ iff $m \models \eta$.

Algorithm. The question "$\mathcal{G} \models \Diamond F$?" is answered by working over $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$:

- Consider the formula over markings $M[F]$ (see Definition 15.4.3);
- Express $M[F]$ in disjunctive normal form as below, where $a_{i,j}$ can be e or $\neg e$ for $e \in E_{T'}$:
$$\eta = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} a_{i,j}$$
- Apply the Simplification Rule in Proposition 15.4.4, as far as possible, thus obtaining a formula η';
- For any conjunct in η' of the kind $e_1 \wedge \ldots \wedge e_h \wedge \neg e'_1 \wedge \ldots \wedge \neg e'_l$:
  - Take the configuration $C = \lfloor \{e_1, \ldots, e_h\} \rfloor$.

– Consider the safe marking reached after C, i.e., $m_C = (m_0 \cup \bigcup_{t \in C} t^\bullet) - \bigcup_{t \in C} {}^\bullet t$, where m_0 is the initial marking of $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$ (consisting of all minimal places). Surely m_C includes $\{e_1, \dots, e_h\}$. Hence, the only reason why the conjunct may not be true is that m_C includes some of the $\{e'_1, \dots, e'_l\}$. In this case look for a configuration $C' \supseteq C$, which enriches C with transitions which consume the e'_j but not the e_i.

- The formula $\Diamond F$ holds iff this check succeeds for at least one conjunct.

For instance, suppose that we want to check that our sample graph grammar $\mathcal{CP}$ satisfies $\Box F$, where F is a $\mathcal{L}2$ formula specifying that every engaged process is connected through connection c_2 to exactly one other engaged process, i.e.,

$$\begin{aligned} F \;=\; & \forall x.(\mathit{type}(x) = PE \;\Rightarrow\; \exists y.(x \neq y \,\wedge\, \mathit{type}(y) = PE \,\wedge\, c_2(x) = c_2(y) \\ & \wedge\, \forall z.(\mathit{type}(z) = PE \,\wedge\, x \neq z \,\wedge\, c_2(x) = c_2(z) \;\Rightarrow\; y = z))). \end{aligned}$$

The encoding $\varphi = M[F]$ simplifies to

$$\varphi \equiv (5\colon PE \iff 6\colon PE) \wedge (7\colon PE \iff 8\colon PE) \wedge (9\colon PE \iff 10\colon PE)$$

and we have to check that the truncation does not satisfy

$$\begin{aligned} \Diamond\neg\varphi \;=\; & \Diamond[(5\colon PE \wedge \neg 6\colon PE) \vee (\neg 5\colon PE \wedge 6\colon PE) \vee (7\colon PE \wedge \neg 8\colon PE) \\ & \vee (\neg 7\colon PE \wedge 8\colon PE) \vee (9\colon PE \wedge \neg 10\colon PE) \vee (\neg 9\colon PE \wedge 10\colon PE)], \end{aligned}$$

which can be done by using the described verification procedure.

15.5 Conclusions

We have discussed how the finite prefix approach, originally introduced by McMillan for Petri nets, can be generalised to graph transformation systems. A complete finite prefix can be constructed for some classes of graph grammars, but the problem of constructing it for general, possibly non-read-persistent grammars remains open and represents an interesting direction of further research. Also, it would be interesting to try to determine an upper bound on the size of the prefix, with respect to the number of reachable graphs.

We have shown how the complete finite prefix can be used to model-check some properties of interest for graph transformation systems. We plan to generalise the verification technique proposed here to allow the model-checking of more expressive logics, like the one studied in [Esp94] for Petri nets, where temporal modalities can be arbitrarily nested. We intend to implement the model-checking procedure described in the paper and, as in the case of Petri nets, we expect that its efficiency could be improved by refined cut-off conditions (see, e.g., [ERV66]) which help to decrease the size of the prefix.

As mentioned in the introduction, some efforts have been devoted recently to the development of suitable verification techniques for GTSs. A general theory of verification is presented in [GHK98, Hec98], but without providing directly applicable techniques. In [Kön00b, 1, 3] one can find techniques which are applicable to infinite-state systems: the first defines a general framework based on types for graph rewriting, while the second is based on the construction of suitable approximations of the behaviour of a GTS. Instead, the papers [Var02, Ren03] concentrate on finite-state GTSs. They both generate a suitable labelled transition system out of a given finite-state GTS and then [Var02] resorts to model-checkers like SPIN, while [Ren03] discusses the decidability of the model-checking problem for a logic, based on regular path expressions, allowing to talk about the history of nodes along computations. The main difference with respect to our work is that they do not exploit a partial order semantics, with an explicit representation of concurrency, and thus considering the possible interleavings of concurrent events these techniques may suffer of the state-explosion problem.

Acknowledgements: We would like to thank the anonymous referees for their helpful comments. We are also grateful to Javier Esparza for interesting and helpful discussions on the topic of this paper.

Bibliography

[Aba99] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[AG97] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *Proc. of CONCUR '97*, pages 59–73. Springer-Verlag, 1997. LNCS 1243.

[AJKP98] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In *Proc. of CAV '98*, pages 379–390. Springer, 1998. LNCS 1427.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[Bal00] Paolo Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000. Available as technical report n. TD-1/00.

[Bar90] Henk P. Barendregt. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 321–364. Elsevier, 1990.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BBN04] Johannes Borgström, Sébastien Briais, and Uwe Nestmann. Symbolic bisimulation in the spi calculus. In *Proc. of CONCUR '04*, pages 161–176. Springer-Verlag, 2004. LNCS 3170.

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[BCK02] Paolo Baldan, Andrea Corradini, and Barbara König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In H. Ehrig, B. Krämer, and A. Ertas, editors, *Proc. of IDPT '02 (Sixth International Conference on Integrated Design & Process Technology)*. Society for Design and Process Science, 2002.

[BCK04a] Paolo Baldan, Andrea Corradini, and Barbara König. An unfolding-based approach for the verification of finite-state graph grammars. Technical Report CS-2004-10, Dipartimento di Informatica, Università Ca' Foscari di Venezia, 2004.

[BCK04b] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.

[BCM98] P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 283–295. Springer Verlag, 1998.

[BCM99] P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.

[BCM01] P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures and processes. *Information and Computation*, 171(1):1–49, 2001.

[BCM02] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Bisimulation equivalences for graph grammars. In *Formal and Natural Computing - Essays Dedicated to Grzegorz Rozenberg*, pages 158–190. Springer, 2002. LNCS 2300.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In Davide Sangiorgi and Robert de Simone, editors, *Proc. of CONCUR '98*, pages 84–98. Springer-Verlag, 1998. LNCS 1466.

[BFEMOM00a] R. Bruni, D. Frutos-Escrig, N. Martí-Oliet, and U. Montanari. Bisimilarity congruences for open terms and term graphs via tile logic. In *Proc. of CONCUR 2000*, pages 259–274. Springer-Verlag, 2000. LNCS 1877.

[BFEMOM00b] R. Bruni, D. Frutos-Escrig, N. Martí-Oliet, and U. Montanari. Tile bisimilarity congruences for open terms and term graphs. Technical Report TR-00-06, Dipartimento di Informatica, Università di Pisa, 2000.

[BFK00] Michael R. Berthold, Ingrid Fischer, and Manuel Koch. Attributed graph transformation with partial attribution. In *Proc. Joint APPLIGRAPH/GETGRATS Worshop on Graph Transformation Systems (GRATRA 2000)*, pages 171–178, 2000.

[BGM98] Roberto Bruni, Fabio Gadducci, and Ugo Montanari. Normal forms for partitions and relations. In *Recents Trends in Algebraic Development Techniques, 12th International Workshop, WADT '98*, pages 31–47. Springer-Verlag, 1998. LNCS 1589, full version to appear in TCS.

[Bir67] G. Birkhoff. *Lattice Theory.* American Mathematical Society, third edition, 1967.

[BK02] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.

[BKK03] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03 (International Static Analysis Symposium)*, pages 255–272. Springer-Verlag, 2003. LNCS 2694.

[BKS04] Paolo Baldan, Barbara König, and Ingo Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *Proc. of ICGT '04 (International Conference on Graph Transformation)*, pages 194–209. Springer-Verlag, 2004. LNCS 3256.

[BMS00] Roberto Bruni, Ugo Montanari, and Vladimiro Sassone. Open ended systems, dynamic bisimulation and tile logic. In *Proc. of IFIP TCS 2000.* Springer, 2000. LNCS 1872.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.

[Bus98] Nadia Busi. *Petri Nets with Inhibitor and Read Arcs: Semantics, Analysis and Application to Process Calculi.* PhD thesis, Dipartimento di Matematica, Università degli Studi di Siena, 1998.

[BvEG+87] Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *Proc. of PARLE '87, Volume 2*, pages 141–158. Springer, 1987. LNCS 259.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL '77 (Los Angeles, California)*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL '79 (San Antonio, Texas)*, pages 269–282. ACM Press, 1979.

[CDFR04] Andrea Corradini, Fernando Luís Dotti, Luciana Foss, and Leila Ribeiro. Translating java code to graph transformation systems. In *Proc. of ICGT '04*, LNCS 3256, pages 383–398. Springer, 2004.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV 2000*, pages 154–169. Springer-Verlag, 2000.

[CGL99] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 1999.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000.

[CMR96] Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.

[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, Vol.1: Foundations*, chapter 3. World Scientific, 1997.

[Cor96] Andrea Corradini. Concurrent graph and term graph rewriting. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 438–464. Springer-Verlag, 1996. LNCS 1119.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2), 1996.

[Cou97] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 5. World Scientific, 1997.

[DDK93] Gnanamalar David, Frank Drewes, and Hans-Jörg Kreowski. Hyperedge replacement with rendezvous. In *Proc. of TAP-SOFT (Theory and Practice of Software Development)*, pages 167–181. Springer-Verlag, 1993. LNCS 668.

[Del00] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. of CAV '00*, pages 53–68. Springer-Verlag, 2000. LNCS 1855.

[DFRS03] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, pages 261–275. Springer, 2003. LNCS 2884.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. on Programming Languages and Systems*, 19(2):253–291, 1997.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[DJFB02] Giorgio Delzanno, cois Raskin Jean-Fran and Laurent Van Begin. Towards the automated verification of multithreaded java programs. In *Proc. of TACAS '03*, pages 173–187. Springer, 2002. LNCS 2280.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DM87] P. Degano and U. Montanari. A model of distributed systems based on graph rewriting. *Journal of the ACM*, 2(34):411–449, 1987.

[dS85] Roberto de Simone. Higher level synchronizing devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2: Applications, Languages and Tools*. World Scientific, 1999.

[EGPP99] H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. High-level replacement systems with applications to algebraic specifications and Petri nets. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Con-*

currency, Parallellism, and Distribution, chapter 6, pages 341–400. World Scientific, 1999.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation—part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 4. World Scientific, 1997.

[EHPP04] Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement categories and systems. In *Proc. of ICGT '04*, pages 144–160. Springer, 2004. LNCS 3256.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. 1st International Workshop on Graph Grammars*, pages 1–69. Springer-Verlag, 1979. LNCS 73.

[Ehr02] H. Ehrig. Bigraphs meet double pushouts. *EATCS Bulletin*, 78:72–85, October 2002.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. of CAV'2000*, pages 232–247. Springer, 2000. LNCS 1855.

[EK04a] Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Proc. of FOSSACS '04*, pages 151–166. Springer, 2004. LNCS 2987.

[EK04b] Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. Technical Report 01/2004, Universität Stuttgart, 2004.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution.* World Scientific, 1999.

[EN94] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets. Technical Report RS-94-8, BRICS, May 1994.

[Eng91] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.

[EPS73] H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: An algebraic approach. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 167–180, 1973.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *Proc. of ICGT '04*, LNCS 3256, pages 161–177. Springer, 2004.

[ER99] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proc. of CONCUR'99*, number 1664 in Lecture Notes in Computer Science, pages 2–20. Springer-Verlag, 1999.

[ERV66] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96*, pages 87–106. Springer-Verlag, 1966. LNCS 1055.

[ERV02] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[Esp97] J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[FGM00] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Non interference for the analysis of cryptographic protocols. In *Proc. of ICALP '00*, pages 354–372. Springer-Verlag, 2000. LNCS 1853.

[Fin91] Alain Finkel. The minimal coverability graph for Petri nets. In *Proc. of ATPN (Applications and Theory of Petri Nets)*, pages 210–243. Springer, 1991. LNCS 674.

[FM98] Maribel Fernández and Ian Mackie. Coinductive techniques for operational equivalence of interaction nets. In *Proc. of LICS '98*. IEEE Computer Society Press, 1998.

[Gar99] Philippa Gardner. Closed action calculi. *Theoretical Computer Science*, 228(1–2):77–103, 1999.

[Gen98] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proc. of RTA '98*, pages 151–165. Springer-Verlag, 1998. LNCS 1379.

[GH97] Fabio Gadducci and Reiko Heckel. An inductive view of graph transformation. In *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT '97*, pages 223–237. Springer-Verlag, 1997. LNCS 1376.

[GHK98] Fabio Gadducci, Reiko Heckel, and Manuel Koch. A fully abstract model for graph-interpreted temporal logic. In *Proc.*

of TAGT '98 (Theory and Application of Graph Transformations), pages 310–322. Springer-Verlag, 1998. LNCS 1764.

[GM99] F. Gadducci and U. Montanari. The tile model. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999.

[GM01] F. Gadducci and U. Montanari. A concurrent graph semantics for mobile ambients. In S. Brookes and M. Mislove, editors, *Proceedings of the 17th MFPS*, volume 45 of *Electronic Notes in Computer Science*. Elsevier Science, 2001.

[GM02] Fabio Gadducci and Ugo Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoretical Computer Science*, 285(2):319–358, 2002.

[GV92] J.F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag, 1992. LNCS 643.

[Has97] Masahito Hasegawa. *Models of Sharing Graphs (A Categorical Semantics of Let and Letrec)*. PhD thesis, University of Edingburgh, 1997. available in Springer Distinguished Dissertation Series.

[Hec98] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering (FASE '98)*, 1998.

[HIM00] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Reconfiguration of software architecture styles with name mobility. In António Porto and Gruia-Catalin Roman, editors, *Proc. of COORDINATION 2000*, pages 148–163. Springer-Verlag, 2000. LNCS 1906.

[Hir98] Daniel Hirschkoff. Automatically proving up to bisimulation. In *Proc. of MFCS '98 Workshop on Concurrency*, number 18 in ENTCS, 1998.

[Hir99] Daniel Hirschkoff. On the benefits of using the up-to techniques for bisimulation verification. In *Proc. of TACAS '99*, pages 285–299, 1999. LNCS 1579.

[HM01] Dan Hirsch and Ugo Montanari. Synchronized hyperedge replacement with name mobility (a graphical calculus for mobile systems). In *Proc. of CONCUR '01*, pages 121–136. Springer-Verlag, 2001. LNCS 2154.

[Hod93] Wilfrid Hodges. *Model Theory*. Cambridge University Press, 1993.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[Hon96] Kohei Honda. Composing processes. In *Proc. of POPL'96*, pages 344–357. ACM, 1996.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. of FoSSaCS '01*, pages 230–245. Springer, 2001. LNCS 2030.

[HR89] R. Howell and L. Rosier. Problems concerning fairness and temporal logic for conflict-free Petri net. *Theoretical Computer Science*, 64:305–329, 1989.

[HR00] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proc. of ICALP '00*, pages 415–427. Springer-Verlag, 2000. LNCS 1853.

[HRY91] Rodney R. Howell, Louis E. Rosier, and Hsu-Chun Yen. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science*, 82:341–372, 1991.

[HVY00] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of ESOP '00*, pages 180–199. Springer-Verlag, 2000. LNCS 1782.

[IK00] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear π-calculus with I/O subtyping. *Information and Computation*, 161:1–44, 2000.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proc. of POPL '01*, pages 128–141. ACM, 2001.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A core calculus for Java and GJ. In *Proc. of OOPSLA 1999*, 1999.

[Jan90] Petr Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74:71–93, 1990.

[Jen] Ole Høgh Jensen. *Bigraphs*. PhD thesis, University of Aalborg. to appear.

[JM03] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *Proc. of POPL 2003*, pages 38–49. ACM, 2003.

[JN95] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4: Semantic Modelling*, pages 527–636. Oxford University Press, 1995.

[Kel95] P. Kelb. *Abstraktionstechniken für automatische Verifikationsmethoden.* PhD thesis, Carl-von-Ossietzky-Universität Oldenburg, 1995.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[KM01] Barbara König and Ugo Montanari. Observational equivalence for synchronized graph rewriting with mobility. In *Proc. of TACS '01*, pages 145–164. Springer-Verlag, 2001. LNCS 2215.

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Proc. of SAS '95*, pages 225–242. Springer-Verlag, 1995. LNCS 983.

[Kob98] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(2):436–482, 1998.

[Kob02] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

[Koc00] Manuel Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems.* PhD thesis, Technische Universität Berlin, 2000.

[Kön] Barbara König. Analysing input/output-capabilities of mobile processes with a generic type system. *Journal of Logic and Algebraic Programming.* to appear.

[Kön99a] Barbara König. *Description and Verification of Mobile Processes with Graph Rewriting Techniques.* PhD thesis, Technische Universität München, 1999.

[Kön99b] Barbara König. Generating type systems for process graphs. In *Proc. of CONCUR '99*, pages 352–367. Springer-Verlag, 1999. LNCS 1664.

[Kön00a] Barbara König. A general framework for types in graph rewriting. In *Proc. of FST TCS '00*, pages 373–384. Springer-Verlag, 2000. LNCS 1974.

[Kön00b] Barbara König. A general framework for types in graph rewriting. Technical Report TUM-I0014, Technische Universität München, 2000.

[Kön00c] Barbara König. A graph rewriting semantics for the polyadic π-calculus. In *Workshop on Graph Transformation and Visual Modeling Techniques (Geneva, Switzerland), ICALP Workshops '00*, pages 451–458. Carleton Scientific, 2000.

[Kön00d] Barbara König. Hypergraph construction and its application to the compositional modelling of concurrency. In *GRATRA '00: Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, 2000. Proceedings available as Report Nr. 2000-2 (Technische Universität Berlin).

[Kön02] Barbara König. Hypergraph construction and its application to the static analysis of concurrent systems. *Mathematical Structures in Computer Science*, 12(2):149–175, 2002.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[KSS00] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR 2000*, pages 489–503. Springer-Verlag, 2000. LNCS 1877.

[Laf90] Yves Lafont. Interaction nets. In *Proc. of POPL '90*, pages 95–108. ACM Press, 1990.

[Lan92] R. Langerak. *Transformation and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, 1992.

[Lei01] James J. Leifer. *Operational congruences for reactive systems*. PhD thesis, University of Cambridge Computer Laboratory, September 2001.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting*, chapter 14, pages 185–199. Wiley, 1993.

[LM86] K.G. Larsen and R. Milner. A complete protocol verification using relativized bisimulation. Technical Report ECS-LFCS-86-13, LFCS, University of Edinburgh, 1986.

[LM00] James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In *Proc. of CONCUR 2000*, 2000. LNCS 1877.

[LS04] Stephen Lack and Paweł Sobociński. Adhesive categories. In *Proc. of FOSSACS '04*, pages 273–288. Springer, 2004. LNCS 2987.

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[McM98] K. L. McMillan. Getting started with SMV. Cadence Berkeley Labs, 1998. Tutorial.

[MCP93] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.

[Mes96] José Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In *Concurrency Theory*, pages 331–372. Springer-Verlag, 1996. LNCS 1119.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. LNCS 92.

[Mil93] Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, Heidelberg, 1993.

[Mil96] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Mil01] Robin Milner. Bigraphical reactive systems. In *Proc. of CONCUR '01*, pages 16–35. Springer-Verlag, 2001. LNCS 2154.

[MM90] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990.

[MP95] Ugo Montanari and Marco Pistore. Concurrent semantics for the π-calculus. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[MR99] U. Montanari and F. Rossi. Graph rewriting, constraint solving and tiles for coordinating distributed systems. *Applied Categorical Structures*, 7:333–370, 1999.

[MS92a] R. Milner and D. Sangiorgi. Techniques of weak bisimulation up-to. In *Proc. of CONCUR '92*. Springer-Verlag, 1992. LNCS 630.

[MS92b] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. of ICALP '92*. Springer-Verlag, 1992. LNCS 623.

[MS92c] U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16:171–196, 1992.

[MT01] Panagiotis Manolios and Richard J. Trefler. Safety and liveness in branching time. In *Proc. of LICS '01*, 2001.

[NN01] Hanne Riis Nielson and Flemming Nielson. Shape analysis for mobile ambients. *Nordic Journal of Computing*, 8(2):233–275, 2001.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NNS00] Flemming Nielson, Hanne Riis Nielson, and Mooly Sagiv. A Kleene analysis of mobile ambients. In *Proc. of ESOP 2000*. Springer-Verlag, 2000. LNCS 1782.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[NS97] Uwe Nestmann and Martin Steffen. Typing confluence. In *Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems (Cesena, Italy, July 4–5, 1997)*, pages 77–101, 1997.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science '81*, pages 167–183. Springer, 1981. LNCS 104.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists (Foundations of Computing)*. MIT Press, 1991.

[Pis99] M. Pistore. *History Dependent Automata*. PhD thesis, Department of Computer Science, University of Pisa, 1999.

[PP95] G.M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoretical Computer Science*, 138(2):425–454, 1995.

[Pra94] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of LICS ‘93*, pages 376–385, 1993.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Proc. of CAV ’02*, pages 107–122. Springer-Verlag, 2002. LNCS 2404.

[RE99] Christine Röckl and Javier Esparza. Proof-checking protocols using bisimulations. In *Proc. of CONCUR ’99*, pages 525–540. Springer, 1999. LNCS 1664.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[Rel04] Nicolas Relange. Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade. Master’s thesis, Universität Stuttgart, September 2004.

[Ren03] Arend Rensink. Model checking graph grammars. In *Proc. of AVOCS ’03 (Workshop on Automated Verification of Critical Systems)*, 2003.

[Ren04a] Arend Rensink. Canonical graph shapes. In *Proc. of ESOP ’04*, pages 401–415. Springer, 2004. LNCS 2986.

[Ren04b] Arend Rensink. State space abstraction using shape graphs. In *Proc. of AVIS ’04 (Third International Workshop on Automatic Verification of Infinite-State Systems)*, ENTCS, 2004. to appear.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proc. of ICALP’97*, pages 471–481. Springer-Verlag, 1997. LNCS 1256.

[Rib96] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, Technische Universität Berlin, 1996.

[Rob63] Abraham Robinson. *Introduction to Model Theory and to the Metamathematics of Algebra.* North-Holland, 1963.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

[RV04] Arend Rensink and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. of ICGT '04*, pages 226–241. Springer, 2004. LNCS 3256.

[San95] Davide Sangiorgi. On the proof method for bisimulation. In *Proc. of MFCS '95 (Mathematical Foundations of Computer Science)*, pages 479–488. Springer, 1995. LNCS 969.

[Sas94] Vladimiro Sassone. *On the Semantics of Petri Nets: Processes, Unfolding and Infinite Computations.* PhD thesis, University of Pisa - Department of Computer Science, 1994.

[Sch97] A. Schürr. Introduction to the specification language PROGRES. In M. Nagl, editor, *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*, pages 248–279. Springer, 1997. LNCS 1170.

[Sch00] Karsten Schmidt. LoLA: A low level analyser. In *Proc. of ATPN (Application and Theory of Petri Nets)*, pages 465–474. Springer, 2000. LNCS 1825.

[Sch02] Stefan Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technische Universität München, 2002.

[Sew02] Peter Sewell. From rewrite rules to bisimulation congruences. *Theoretical Computer Science*, 274(1–2):183–230, 2002.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. of POPL '96*, pages 16–31. ACM Press, 1996.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.

[SS03] Vladimiro Sassone and Paweł Sobociński. Deriving bisimulation congruences: 2-categories vs precategories. In *Proc. of FoSSaCS 2003*, pages 409–424, 2003. LNCS 2620.

[SS04] Vladimiro Sassone and Paweł Sobociński. Congruences for contextual graph-rewriting. Technical Report RS-04-11, BRICS, June 2004.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus—A Theory of Mobile Processes.* Cambridge University Press, 2001.

[Tra50] B. A. Trakhtenbrot. The impossibility of an algorithm for the decision problem for finite models. *Doklady Akademii Nauk SSR*, 70:569–572, 1950.

[Tur04] Sinan Turan. Effiziente Berechnung der Überdeckbarkeit bei Petri-Netzen, June 2004. Studienarbeit (Student research project).

[Var02] Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Workshop on Graph Transformation and Visual Modeling Techniques '02*, volume 72 of *ENTCS*. Elsevier, 2002.

[Vog97] W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. In *Proc. of ICALP'97*, pages 538–548. Springer, 1997. LNCS 1256.

[VSY98] W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and finite prefix for nets with read arcs. In Davide Sangiorgi and Robert de Simone, editors, *Proc. of CONCUR '98*, pages 501–516. Springer-Verlag, 1998. LNCS 1466.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, September 1971.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

[Wal03] Ingo Walther. Implementation of an algorithm for the analysis of graph transformation systems, December 2003. Systementwicklungsprojekt (Student research project).

[WG96] A. Wagner and M. Gogolla. Defining operational behaviour of object specifications by attributed graph transformation. *Fundamenta Informaticae*, 26:407–431, 1996.

[Wil97] Robert Paul Wilson. *Efficient, Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, 1997.

[Win87] Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 325–392. Springer, 1987. LNCS 255.

[YH02] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Information and Computation*, 174(2):143–179, 2002.

[Yos94] Nobuko Yoshida. Graph notation for concurrent combinators. In *Proc. of TPPP '94*. Springer-Verlag, 1994. LNCS 907.