Modular Performance Modelling of Mobile Applications using Graph Transformation

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Niaz Hussain Arijo

Department of Computer Science

University of Leicester

April 2012

Dedication

I dedicate this thesis to my loving parents and my beloved brother, Shaman Ali, who are the source of inspiration for me throughout my life.

Declaration

I hereby declare that this submission is my own work and that it is the result of work done during the period of registration. To the best of my knowledge, it contains no previously published materiel written by another person. None of this work has been submitted for another degree at the University of Leicester or any other University.

Parts of this thesis submission appeared in the following publications, to each of which I have made substantial contributions.

- N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore. Modular Performance Modelling for Mobile Applications. In Samuel Kounev, Vittorio Cortellessa, Raffaela Mirandola, and David J. Lilja, editors, ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011. ACM, 2011.

- N. Arijo and R. Heckel, View-based Modelling and State-Space Generation for Graph Transformation Systems, GT-VMT'12, 11th International Workshop on Graph Transformation and Visual Modeling Techniques, March 24-25, 2012, in Tallin, Estonia.

Modular Performance Modelling of Mobile Applications using Graph Transformation

Niaz Hussain Arijo

Abstract

Graph transformation provides a visual and formal notation for modelling systems of dynamic nature. We use graph transformation for modelling mobility and performance, and provide a methodology for modular system modelling to handle scalbility issues of large systems. In our methodology we have distinguished three approaches for system modelling, monolithic, top-down and bottom-up. In the monolithic approach, a system is modelled as a global or whole-world system. In the top-down approach, a global system is projected to its views based on their local type graphs. In the bottom-up approach, a system is modelled as a set of subsystems with shared interface. A whole system is composed from its subsystems.

We generate labelled transition systems (LTSs) from graph transformation systems/views in GROOVE and transform them into Continuous Time Markov Chains (CTMCs). These CTMCs are further translated into the Performance Evalution Process Algebra (PEPA) or PRISM. In PEPA and PRISM subsystems are synchronized over shared labels to compose a global system. We demonstrate that the composed model is bisimilar to its original global model. In addition stochastic analysis of models are also carried out in PEPA or PRISM for performance checking.

We have given tool support for view generation from a global graph transformation system in GROOVE and transforming LTSs generated from graph transformation systems, into CTMCs, and CTMCs into PEPA or PRISM models.

Acknowledgements

My first and foremost thanks go to Allah, The Almighty, for all His blessings. This thesis definitely would not have been possible without my supervisor Professor Reiko Heckel, who guided and encouraged me during my PhD studies. While working on this thesis, he took great interest in reviewing and proof reading the drafts and his valuable feedback has made possible this thesis in its current shape. I sincerely appreciate the time and effort that he put into this thesis.

I am grateful to Dr. Mirco Tribastone from the Institut für Informatik, LMU München and Prof. Stephen Gilmore from the School of Informatics, University of Edinburgh who helped us in the development phase of the tool for translating GROOVE models into PEPA models.

I am very much thankful to the teaching staff, administrative staff and colleagues of the Department of Computer Science, University of Leicester for their timely help and support during my PhD studies. I owe my sincere thanks to my friends, especially Hyder Ali Nizamani, Muhammad Naeem, Bashir Ahmed Memon, Muhammad Zahir, Mumtaz Yatu and Ajab Khan for their valuable company and help during my stay in Leicester.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Mobile applications often have to be location-aware to deliver the desired functionality. The increasing use of mobile technology makes it necessary to address issues such as performance and reliability caused by limitations in resources and connectivity typical of the mobile platforms and the domain of application it is operating in.

This domain is potentially very complex, incorporating features of the virtual as well as the physical environment. To illustrate this we consider the example of a traffic information system (TIS). Such an application lives in an environment determined by the road network (used by vehicles) and existing wireless networks for propagating accident information. The application will consist of central components (servers providing traffic information, a mobile phone network, etc.) as well as mobile devices carried by vehicles. To predict the success or otherwise of such an application, or optimise its parameters, we

have to be aware of this environment and incorporate this into our models.

We use graph transformation as modelling language for developing abstract models of systems of dynamic nature. This allows to capture communication, mobility and dynamic change of software architecture. Like other rewriting techniques, graph transformation has a reduction semantics, applications of rules making local changes in the model without reference to an external environment. This is in contrast to the reactive semantics enjoyed by process algebra where an action typically involves communication with an environment that is unkown at the moment (often called a reaction). In process algebra, the overall system behaviour results from synchronization of different models, in which one plays the role of environment for the other.

To analyze performance of models we combine graph transformation systems, providing a visual notation for structural changes and computations, with stochastic delays for rule applications leading to a semantics in terms of Continuous-Time Markov Chains (CTMCs). Analysis techniques for CTMCs allow us to reason about reliability and performance. However, in order to derive a CTMC from a graph transformation system we have to generate its state space first and face the usual scalability problems. The thesis will attempt to answer this challenge by introducing modularity into the process of state space generation.

1.2 Thesis Statement

Mobile applications are complex in their interaction and communication with other components of a business domain. This new complexity poses challenges

for modelling and analysis, when we have to assess properties such as interoperability, performance and reliability. Using graph transformation to model this complexity, state space generation represents a major bottleneck in the derivation of the CTMC. In order to address this problem we will structure our models into different views. Correspondingly smaller LTSs are then generated from these views independently, with transition labels designed to support subsequent synchronization.

1.3 Our Approach

We distinguish three approaches monolithic, top-down and bottom-up, for modelling a system by graph transformation. In the monolithic approach we model the system by a single global graph transformation system. We have created a global model of the Traffic Information System (TIS). When we analyze a system through state space exploration, we face scalability problems due to state space explosion. It prevents us from the derivation of a CTMC for larger systems.

Through modularity we address this scalability issue by automated generation of local views from a global model. We have identified two modularity approaches i) top-down and ii) bottom-up. In the top-down approach, a global system is decomposed into local views by our View Generator tool, which reduce the rules and start graph of graph transformation system to their local perspectives based on their type graphs. In the bottom-up approach we model individual subsystems and indentify an interface between them over which they cooperate. The interface is used for composing sub-

systems to create a global system. Apart from describing the modular approaches by means of an example, we state explicitly the conditions for their correctness, i.e., the equivalence of the resulting synchronized LTS with the one derived directly from a monolithic system, and evaluate the scalability of the approach.

In order to compose the resulting behaviours and derive the synchronized CTMC we make use of PEPA, the Performance Evaluation Process Algebra [25, 26] or PRISM [32, 31], which allows us to coordinate the individual processes derived from these views over shared labels. In this way we avoid the state space explosion problem while at the same time benefiting from the capabilities of PEPA and PRISM for stochastic analysis of models.

In PEPA we can analyze non-functional properties of mobile applications such as steady-state probabilities and throughput. Steady-state analysis determines the long term probability for the system to be in a certain state and throughput analysis gives the long term frequency of its actions (rule labels). PRISM supports large models compared to PEPA. In PRISM we can analyze the steady-state of a system and service level agreements (SLAs). As PRISM supports continuous stochastic logic, where we can define properties and verify them, for example, the probability of arriving at the destination within a certain period of time, and the time within which an accident will be removed.

1.4 Contributions

Our thesis consists of the following contributions:

1. We model mobile applications with physical mobility, their interaction and communication with other components of a business domain through graph transformation.

2. To avoid scalability problems with state space generation, we give a modular solution where the graph transformation system is decomposed into views, for which labelled transition systems (LTSs) are generated separately to be synchronized in PEPA or PRISM.

3. We generate local views from a global model by our View Generator tool based on a given subgraph of the global model's type graph, creating a projection of the global graph transformation system.

4. We provide tool support for the derivation of CTMCs from LTSs and their translation into PEPA or PRISM.

5. We demonstrate that the result of modular analysis is equivalent to that of the monolithic approach and evaluate practicality by means of a case study.

Parts of this thesis submission appeared in the following publications.

- N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore. Modular Performance Modelling for Mobile Applications. In Samuel Kounev, Vittorio Cortellessa, Raffaela Mirandola, and David J. Lilja, editors, ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011. ACM, 2011.

- N. Arijo and R. Heckel, View-based Modelling and State-Space Generation for Graph Transformation Systems, GT-VMT'12, 11th Interna-

tional Workshop on Graph Transformation and Visual Modeling Techniques, March 24-25, 2012, in Tallin, Estonia.

In [3] we have given monolithic model of our case study, top-down modularity approach for the Car view and the Service view, stochastic analysis of models and formalization of our modularity approach. Parts of this paper have contributed to Chapters 5, 6 and 7.

In [2] we have extended [3] and introduced the bottom-up modularity approach, and have given a more detailed formalization of the modularity approaches. Parts of this paper have contributed to Chapters 6 and 7.

1.5 Overview of the Chapters

In Chapter 2 we discuss different approaches to mobility modelling, performance modelling and analysis, and compositionality of graph transformation systems. In Chapter 3 we give a general introduction of graph transformation, stochastic graph transformation and discuss tools GROOVE, PEPA and PRISM that we used in our work. In Chapter 4 we give our methodology and introduce three approaches (monolithic, top-down and bottom-up) to system modelling. In Chapter 5 we model the global system of our case study, the Traffic Information System (TIS) in GROOVE. In Chapter 6 we decompose the global system of our TIS model into subsystems, i.e., Car view, Recovery view and Service view through our View Generator tool. In Chapter 7 we formalise notions of views for typed attributed graph transformation systems. In Chapter 8 we discuss CTMC and how we generate it from an LTS of graph transformation system of GROOVE and rate. In Chapter 9 we dis-

cuss our tool support for CTMCs generation and the tool that automatically generates views from a global graph transformation system based on their local type graphs. In Chapter 10 we give comparative analysis of global and synchronized models and discuss the limitations of GROOVE, PRISM and PEPA. In Chapter 11 we give conclusions and future work.

Chapter 2

Related Work

In this chapter we will discuss different approaches to modelling mobility, performance modelling and analysis, and compositionality of graph transformation systems. We have chosen several parameters to compare our work with related work. These parameters include the ability to model mobility, visual notation, modularity (at type level and instance level), operational semantics (loose and concrete), stochastic analysis, synchronization over shared labels.

2.1 Compositionality in Graph Transformation

In [39, 41] a notion of composition of graph transformations is defined. Rules are decomposed into smaller subrules using the concept of graph interface, which works on subgraphs of the complete or whole-world start graph and its resulting graphs. The behaviour of the original rules is captured by merging

corresponding subrules, but there is no actual synchronization of rules as achieved in our case by PEPA and PRISM, where we synchronize rules over shared labels and projection extended to NACs. Another difference is that they do decomposition at instance level not at type level, as we demonstrate in our case study.

More recently, the idea of "borrowed context" has been used to achieve modularity [5]. Specifically, this idea has been applied in [40] to generate transition systems in GROOVE in a compositional way. This work shares some of our motivation, but it does not consider negative application conditions.

In the work [17, 20, 21], a common reference model is given out of which views are derived and later integrated to compose a system model. Each view is a restriction of the conceptual model, which incorporates particular aspects (perspectives) of a system. Inconsistencies between the different views are identified, i.e.,

(1) The same concept, e.g., operation, is specified in two different views using different names.

(2) The same names are used in two different views denoting semantically different concepts.

(3) Execution of a view operation violates the constraints defined by another view.

These inconsistencies are handled by a model manager. If needed, the original reference model may be extended. These inconsistencies do not occur in our top-down approach, because we are deriving views by the projection

of a global system, which is already defined, and synchronization of views is carried out in PEPA or PRISM over the shared labels of a transition system. The work in [17, 20, 21] follows a loose semantics, due to its open type specification, where unspecified deletion and creation of nodes and edges could be done. Instead, we are following a concrete semantics to generate transition system and projection extended to NACs.

In [27] the semantics of dynamic sytems is expressed where systems whose topology admits successives transformations. Compositionality is achieved by extending the composition of partial maps in a categorical context. In [27] the main emphasis is on the sequential composition of edges.

In [13, 29, 30] the notions of transformation units and transformation modules are introduced. Large graph transformation systems are constructed from smaller ones. Compositionality is achieved by interleaving ordinary direct derivation steps with calls of imported transformation units. A transformation unit can use other transformation units for reuse and structuring. A cluster of transformation units is called a module. Control conditions are intoduced which cut down the nondeterminism of rule applications.

The compositionality approaches above do not give support for stochastic analysis of a system, which we achieve by transforming LTSs of models into PEPA (stochastic process algebra) or PRISM (stochastic model checker). They do not support synchronization over shared labels. We extend and apply earlier work on stochastic modelling and modularity of graph transformation systems, especially [23, 24].

Using modularity to reduce complexity is an old idea, even in graph transformation. Specifically the present approach is inspired by proposals on view

based modelling [17, 20], but apart from being fully formalised, differs from it in two ways. First, we consider attributes and application conditions, and second, the present approach does not apply a loose semantics. The reason is that, to generate a transition system, loose semantics is not practical because it would allow far too many transitions. At the specification level this means that we use more restrictive conditions on views and their composition.

2.2 Performance Modelling and Analysis for Mobile Applications

The majority of approaches to performance modellings and analysis for mobile systems are stochastic extensions of process algebra such as StoKlaim [11] deriving from Klaim (the Kernel Language for Agents Interaction and Mobility) [7, 9] and mobile stochastic logic (MoSL) [10, 12]. Occasionally, these languages are integrated with standard modelling languages like the UML to provide a more mainstream frontend notation for modelling [43].

Where these approaches address mobility, it is usually code/agent mobility in the context of global computing, rather than physical mobility as in our approach. We propose to model both architecture and interaction of services and physical mobility by graphs and graph transformation. This has the advantage of being able to use a visual formal language, with less emphasis on logical and algebraic notation as used in approaches based on process algebra and logic.

In our work, we (mostly) separate operations which change locations from

operations which change the state of the system. A similar separation of concerns is established in the PEPA Nets modelling language [18], where PEPA terms are used as the tokens of a coloured stochastic Petri net and movement of a token from one place to another represents mobility and changes of state are denoted by rewriting the PEPA terms within a place.

2.3 Comparison with Related Approaches

In [39, 41] graph transformation systems are decomposed, where rules are decomposed into subrules and original rules are constructed by merging corresponding subrules. In [17, 20, 21] views are derived from a common reference model which can be integrated to compose a system model. In [13, 29, 30] compositionality is achieved by interleaving ordinary direct derivation steps, while control conditions cut down the nondeterminism of rule applications. We perform decomposition at the type level and projection is extended to NACs. We synchronize subrules over shared labels in either PEPA or PRISM where we also perform stochastic analysis of models.

StoKlaim [11] and PEPA Nets [18] are stochastic extensions of process algebras for mobile systems. Where these approaches address mobility, it is usually code/agent mobility in the context of global computing. We use graphs and graph transformation for mobility modelling. Graphs give a visual notation for architecture and behaviour modelling.

In Table 2.1, we have taken mobility, visual notation, modularity (type level and instance level), operational semantics (loose and concrete), stochastic analysis and synchronization over shared labels as our parameters. We

Table 2.1: Comparison of our work with other approaches

	Mobility	Visual Notation	Modularity Type	Modularity Instance	Semantics Loose	Semantics Concrete	Stochastic Analysis	Synchronization over Shared Labels
StoKLAIM	√		√			√	√	√
PEPA Nets	√	√	√			√	√	√
Instance based Compositionality [39, 41]		√		√		√		
Type based compositionality [17, 20]		√	√	√	√			
Control Structure based Compositionality [29, 30]		√	√	√		√		
Our Work	√	√	√	√		√	√	√

compare our approach with related approaches, where our approach satisfies all the given parameters, while others satisfy some of them.

2.4 Summary

In this chapter we have discussed different approaches to mobility modelling, performance modelling and analysis, and compositionality of graph transformation systems. We have given a comparative analysis of our work with other related approaches by means of various parameters.

Chapter 3

Background

In this chapter we will informally discuss attributed graphs, morphisms, graph transformation systems, stochastic graph transformation systems and the tools that we have used in our work, i.e., GROOVE [38], PEPA [25, 26, 42] and PRISM [31, 32].

3.1 Graph Transformation

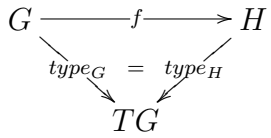
Graphs are an intuitive and very powerful formalism for modelling a system architecture and its behaviour showing the conceptual structure and the functionality of a system. Type graphs depict the architecture of a system and graph transformation rules model the behaviour.

Graph Transformation was introduced in the early 1970s as a generalization of Term Rewriting and Chomsky's (string) grammars [14].

3.1.1 Type and Instance Graphs

Similar to class diagrams in object-oriented models or entity relationship diagrams in data models, we use type graphs to restrict the set of admissible graphs. One common means to restrict the shape of an object is to prescribe a *type* for the object.

A type graph TG is a graph whose nodes represent node types and whose edges represent edge types. A graph that is typed over a type graph TG, also called *instance graph* over TG, is a graph G equipped with a graph morphism $type_G : G \to TG$ that assigns a type to every node and edge in G. A typed graph morphism is a morphism that preserves the typing of graphs.



An edge type in TG represents a structural relationship among nodes of TG-typed graphs. This is because, due to the typing morphism, edges of an edge type may only connect nodes of the node types that are incident to the edge type in TG. A node type can be compared to a class and an edge type can be compared to an association in a UML class diagram. Type graphs can be represented by UML class diagrams, instance graphs can be represented by corresponding UML object diagrams.

3.1.2 Attributed Type Graphs

Attributed graphs are used for modelling data-intensive systems such as object-oriented systems. Attributes of a graph behave similarly as attributes of an object oriented system. Attributed graphs are those graphs in which nodes and edges can be assigned attributes of some data types.

Node attributes can be used to store additional information in a node. As known from object-oriented languages, an attribute consists of a name and a data value. In the context of typed graphs, we have to declare the attributes belonging to a certain node type by their name and data type in the type graph. In a corresponding instance graph, every instance of the node type can carry its own values for these attributes.

In connection with graph transformations, attributes can be used as special labels that identify certain nodes along a sequence of transformation steps. They are also useful to restrict the applicability of transformation rules to situations in which involved nodes have specific attribute values.

Attributed graphs can occur at both levels, as type graphs and as instance graphs. In type graphs, the attributes are in fact attribute declarations, and their values are in fact sort names that determine possible values to be assigned to the attributes in an instance graph.

Similarly to the terminology for database systems, we call the set of all possible values that can be assigned to attributes a *domain*. The values belonging to the domain are partitioned into disjoint *sorts* representing different basic data types. A subset of the value domain consists of *variable names* which can be assigned to attributes instead of concrete values.

Since we want to declare node attributes in a type graph by using sort names as attribute values, type graphs are attributed over the set of sort names.

Fig. 3.1 shows the type graph of the TIS model and Fig. 3.2 shows a possible instance graph. The TIS and Car nodes of Fig. 3.2 are typed over type nodes of Fig. 3.1; id is the attribute of a Car node with attribute values 1 and 2 which show the identity of cars.

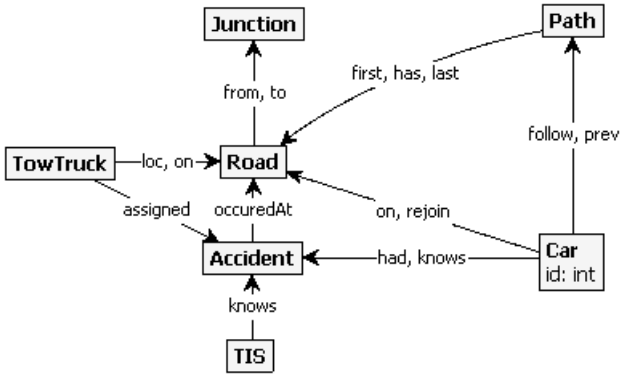


Figure 3.1: Type graph of the Traffic Information System.

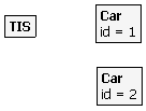


Figure 3.2: Instance graph

3.1.3 Graph Transformation

In the case of graph transformation rules, the left-hand side and the right-hand side are instance graphs. The left-hand side represents the pre-conditions of the rule while the right-hand side describes its effects and post-conditions.

According to the aforementioned paradigm, such a rule can be applied to a concrete instance graph G, also called *host graph*, whenever there is an *occurrence* of the left-hand side L in G. In this context, occurrence means a subgraph of G which has the same structure as L and whose elements conform to the typing and attribute values of L. Sometimes, such an occurrence is also called a *match* of L in G.

Formally a graph transformation rule $p(\bar{x}) : [N]\ L \to R$ consists of a rule name p, a declaration of formal parameters $\bar{x} = x_1, \ldots, x_n$ with variables x_i ranging over attribute values, a left-hand side L representing the pattern of elements required for the application of the rule, a right hand side R describing the situation this pattern is to be replaced with, and a negative application condition N stating the absence of certain elements in the context of this pattern. The intersection graph $L \cap R$ defines the elements that are read by the rule, but are not consumed. A rule application deletes $L \setminus R$ elements from a host graph and adds $R \setminus L$ elements in the host graph. Nodes in L, R, and N are attributed by expressions over variables in X.

Given a graph G and a rule $p(\bar{x}) : [N]\ L \to R$, we can apply the rule if there is a match $m : L \to G$ embedding L into G such that none of the forbidden patterns in N are present. The transformation is denoted by $G \overset{p(\bar{a}),m}{\Longrightarrow} H$, where $\bar{a} = a_1, \ldots, a_n$ is the list of actual parameters, i.e., attribute values occurring in G given by $a_i = m(x_i)$ and m is a match in the graph G. The transformation from graph G to graph H done in zero or more steps is denoted by $G \Longrightarrow^* H$, i.e., $G = G_0 \overset{p_1(\bar{a}_1),m_1}{\Longrightarrow} G_1 \overset{p_2(\bar{a}_2),m_2}{\Longrightarrow} \ldots G_{n-1} \overset{p_n(\bar{a}_n),m_n}{\Longrightarrow} G_n = H$. Thus instantiated rule signatures serve as labels.

The left-hand side L of a transformation rule specifies which pattern has

to be present in a host graph before the rule can be applied. On the contrary, we sometimes require that certain elements are *not* present when a rule is applied. In such a case, we add *negative application conditions* (NAC) to a transformation rule [19].

We specify negative application conditions by so-called *forbidden graphs*. A transformation rule can only be applied to an occurrence of its left-hand side, if the occurrence cannot be extended to the forbidden graph.

A negative application condition is satisfied with respect to a given match $m : L \to G$ if there exists no morphism $n : N \to G$ such that any two graph objects being mapped to one another by l are mapped to the same object in the host graph G by both n and m (more formally: there must be no total morphism $n : N \to G$ such that $n \circ l = m$).

3.1.4 Graph Transformation Systems

Combining a type graph and a set of graph transformation rules that operate on instances of the type graph results in a graph transformation system. A type graph determines the set of valid instance graphs, and graph transformation rules are used to define local transformations of graphs. In software engineering, graph transformation systems can be used for various specification and modelling purposes. Applications range from specifying visual languages to abstract data types, object-oriented programs, and software architectures [15]. In our case, we will use them to model mobile systems and their stochastic analysis.

Graphs represent individual system states, and a transformation step rep-

resents the evolution from one state to a new state. In order to model the behavior of a software system we have to define a set of rules P together with the host graph (start graph). Parameter declarations refer to the numbered *id* attributes, with the nth formal parameter referring to the attribute value labelled $\$n$ as it is shown in Fig 3.4 where \$1 refers to the 1st formal parameter. When a $moveCar(car)$ rule will be applied on the graph of Fig 3.3, the instantiated rule signature $moveCar(1)$ will serve as a label where 1 is the actual parameter.

Formally, a *typed attributed graph transformation system with rule signatures* (GTS) is a tuple $\mathcal{G} = (TG, P, \pi)$ where

- TG is a type graph,
- P is a set of rule names,
- $\pi : P \longrightarrow X^* \times Rules(TG, X)$ assigns to each rule name a pair $\pi(p) = (\bar{x}) : [N]\ L \rightarrow R$ of a parameter declaration $\bar{x} = x_1 \ldots x_n$ and a rule $[N]\ L \rightarrow R$. We call $p(\bar{x}) : [N]\ L \rightarrow R$ a parameterised rule and refer to $p(\bar{x})$ as its signature.

Sometimes, we assume that the actual parameters carry enough information to make transformations deterministic, that is, for transformation steps $G \overset{p(\bar{a})}{\Longrightarrow} H$ and $G \overset{p(\bar{a}')}{\Longrightarrow} H'$, if $a = a'$ then the resulting graphs H and H' are isomorphic.

We will employ the GROOVE syntax for presenting our rules, and will explain this syntax in more detail before returning to an example. In GROOVE notation [38] the various components of a rule (called readers, erasers, creators or embargoes) are combined within a single rule graph, distinguishing

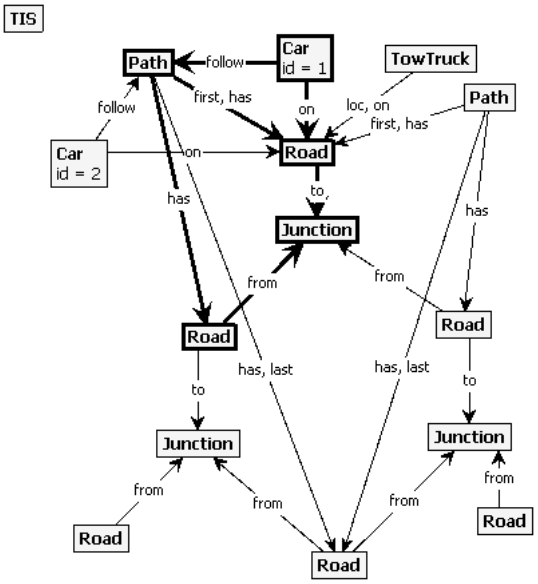


Figure 3.3: Start graph of the Traffic Information System (match for *move-Car(1)* indicated in bold)

them by different colours and styles. Consider for example the rule in Fig. 3.4 modelling the movement of a Car from one Road to another across a Junction. Readers in $L \cap R$, such as the nodes of type *Car, Path, Road, Junction*, and the edges of type *follow, has, to, from*, are thin and solid ordinary graph elements that are required, but preserved by the transformation. Erasers in $L \setminus R$, such as the edge of type *on* pointing to the left-most *Road* node, are shown as thin and dashed elements, to be deleted. Creators in $R \setminus L$, such as the other edge of type *on*, represented by slightly wider, solid outlines, are to be created by the rule. Embargoes in N, such as the nodes of type *Accident* and the edges of type *occurredAt*, *had* and *rejoin*, are represented by wider and dashed outline. They prevent a rule from being applied if the corresponding elements are present in the graph. In a printed version of this thesis, reader elements are shown in black, while all other elements are in

grey. The GROOVE tool uses blue for erasers, green for creators and red for embargoes. Attribute values are depicted as circles pointed to by an edge from the attributed node. For example, in the rule in Fig. 3.4, node Car has attribute *id* whose value is depicted as parameter 1. It corresponds to the 1st formal parameter in the rule's signature *moveCar(car)*.

An instance graph is shown in Fig. 3.3 representing a map of Roads and Junctions as well as two Cars following predefined Paths. To identify Cars, these nodes have been given *id* attributes of type integer. In general we allow graph nodes to be attributed by values of predefined data types, such as strings or natural numbers. In this thesis all attribute values will be positive integers, but see [34] for a general treatment of attributed graphs and their transformation. An application of the rule $moveCar(car)$ in Fig. 3.4 to the Car at the top of the instance graph in Fig. 3.3 will lead to a transition labelled $moveCar(1)$ that will transform the graph by replacing the *on* edge of the Car by one pointing to the next Road. This is possible because the rule's left hand side is matched to the graph as shown by the highlighted nodes and edges in Fig. 3.3. In particular notice that, given the match of the Car to the one with *id = 1*, the dashed (blue) *on* edge in the rule enforces the matching of the left-hand side Road node in the rule to the Road on top, which the Car is following, determines that the other Road node in the rule should be mapped to the next Road, where the new (solid, green) *on* edge will be pointing. The result of the transformation is shown in Fig. 3.5.

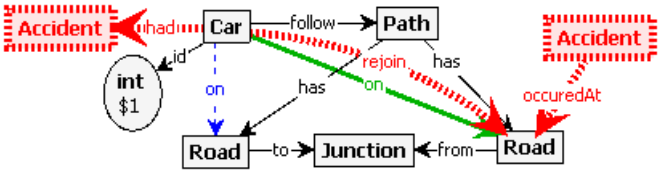


Figure 3.4: Rule, *moveCar*(*car*) models the mobility of a Car following a Path.

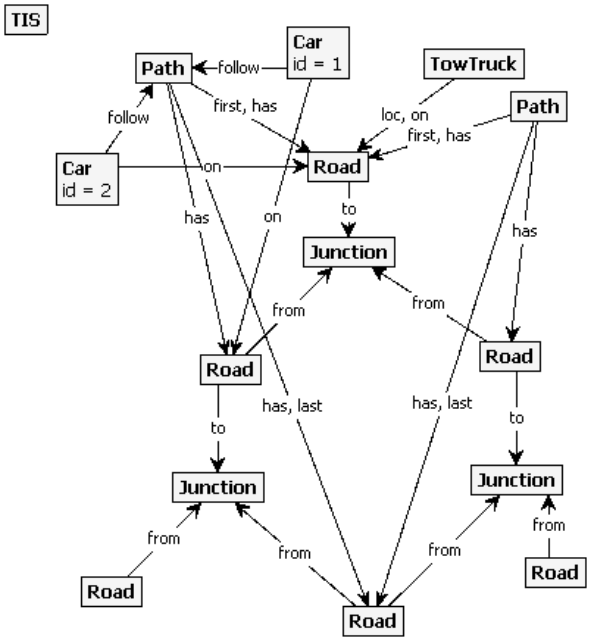


Figure 3.5: Resulting graph after applying *moveCar(1)* to the graph in Fig. 3.3

3.1.5 Graph Transition System

A transition system describes the overall behaviour of a system in terms of its states and transitions. In a graph transition system states represent isomorphism classes of graphs and transitions represent rule applications as it is shown in the Fig. 3.6. Transition systems are frequently used to represent the behavior of software systems. They divide the runtime evolution of a system into discrete *states* and use a binary *transition relation* to define possible state changes. In the case of *graph transition systems* [37], one considers graphs as representations of system states. If used as operational model of a graph transformation system, its state space contains all reachable graphs of the transformation system. Such a transition system can be generated by recursively applying all enabled graph transformation rules of $\mathcal{G}$ at each state and by matching the resulting graphs with already generated isomorphic graphs.

Most analysis techniques based on state transition systems suffer from the state space explosion problem. Therefore, it is desirable to keep the state space as small as possible. In the context of graph transformations, the behavior is determined by the applicability of graph transformation rules on certain states, and the set of applicable rules remains the same for isomorphic graphs. The GROOVE tool [38] that we are using for modelling graph transformation systems uses symmetry reduction for isomorphic graphs to reduce the state space.

After the generation of the labelled transition systems (LTSs) from a graph transformation systems in GROOVE, they are transformed into

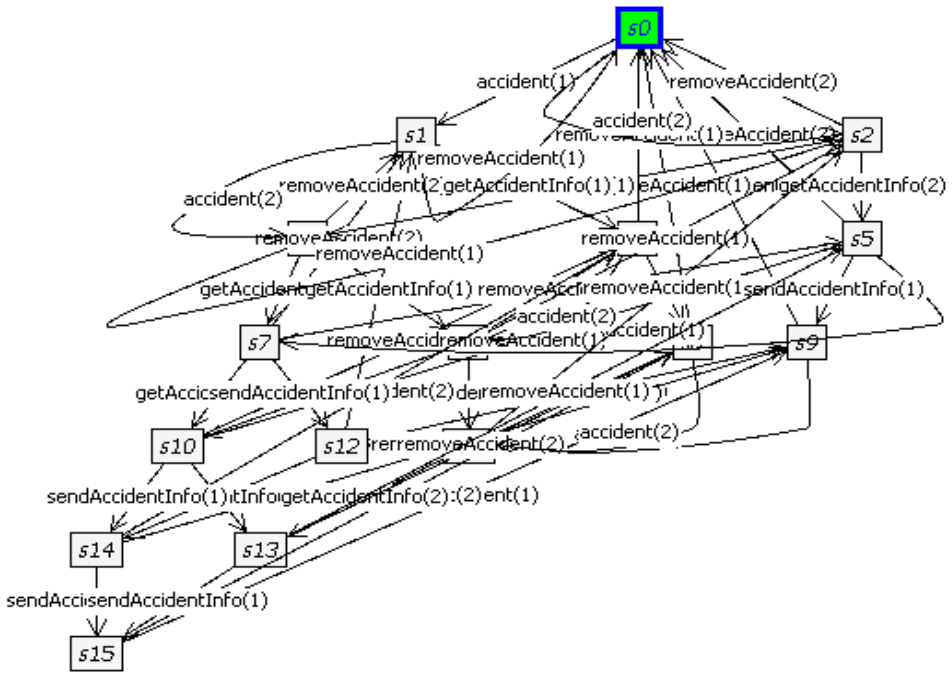


Figure 3.6: A graph transition system

CTMCs and synchronized over shared labels in PEPA or PRISM.

3.2 Stochastic Graph Transformation

Stochastic graph transformation is a blend of graphical modelling and stochastic analysis techniques. For stochastic performance analysis of graph transformation systems we translate the graph transition systems generated by GROOVE into CTMCs, and further into PEPA or PRISM models.

Stochastic graph transformation systems add to each rule a rate, i.e., a positive Real number representing the parameter of the exponential distribution associated with the frequency of execution of the rule (or, equivalently, the reciprocal of the average delay once the rule is enabled for a given match).

Formally, a stochastic GTS is given by a GTS $\mathcal{G} = (TG, P, \pi)$ together with a mapping $\rho : P \to R_+$ assigning each rule name its rate.

3.3 GROOVE

GROOVE (GRaph based Object-Oriented VErification) [38] is a graph transformation tool for software model checking of object oriented systems. It uses graphs to represent states of a system and transitions as rule applications. A graph transformation system given in GROOVE yields a graph transition system as computational model obtained by recursively computing and applying all enabled graph transformation rules at each state. Each newly generated state is compared to all known states up to isomorphism. Finding matches and checking graph isomorphisms are performance-critical parts of the GROOVE simulator. The first problem is NP-complete while the other one is NP.

Labelled transition systems (LTSs) generated by GROOVE [38] minimize the number of states by a form of symmetry reduction: A newly generated graph is checked against all existing graphs for isomorphism (structural equality). If a graph is found to be isomorphic to an existing one, no new state is added to an LTS.

3.4 Performance Evaluation Process Algebra

PEPA [25, 26, 42] extends classical process algebra with the capacity to assign rates to the activities which are described in an abstract model of a

system. Taken together, the information about the rates of performance of activities and the definition of the outcome of performing an activity specify a stochastic process and thus PEPA is said to be a *stochastic process algebra*. The PEPA [25] language is providing a stochastic extension of finite state CSP processes. It has been applied as a modelling language for distributed computer and telecommunications systems such as mobile telephone systems and for components of flexible manufacturing systems such as robotic work-cells. In this thesis, PEPA models are generated from graph transformation systems rather than specified directly.

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. The syntax may be formally introduced by means of the grammar shown in Fig. 3.7. In the grammar S denotes a *sequential component* and P denotes a *model*

$$
\begin{array}{lcll}
S & ::= & & \text{(sequential components)} \\
 & & (\alpha, r).S & \text{(prefix)} \\
 & | & S + S & \text{(choice)} \\
 & | & C_S & \text{(constant)} \\
 & & & \\
P & ::= & & \text{(model components)} \\
 & & P \underset{L}{\bowtie} P & \text{(cooperation)} \\
 & | & P/L & \text{(hiding)} \\
 & | & C & \text{(constant)}
\end{array}
$$

Figure 3.7: The syntax of PEPA

component which executes in parallel. C stands for a constant which denotes

either a sequential or a model component, as defined by a defining equation. C when subscripted with an S stands for constants which denote sequential components. The component combinators, together with their names and interpretations, are presented informally below.

Prefix: The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S. Sequences of actions can be combined to build up a life cycle for a component.

Choice: The life cycle of a sequential component may be more complex than any behaviour which can be expressed using the prefix combinator alone. The choice combinator captures the possibility of competition or selection between different possible activities. The component $P + Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first activity to complete distinguishes one of them: the other is discarded. The system will then behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behaviour associated with components. Constants provide a mechanism for doing this. They are components whose meaning is given by a defining equation.

Hiding: The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted by the division sign in P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component. These activities are not visible to an external observer, nor are they accessible to other components for cooperation. Once an activity is hidden it only appears as the unknown type τ; the rate of the activity, however, remains unaffected.

Cooperation: Most systems are comprised of several components which interact. In PEPA direct interaction, or *cooperation*, between components is represented by the butterfly combinator ($\bowtie$). The set which is used as the subscript to the cooperation symbol determines those activities on which the *cooperands* are forced to synchronise. Thus the cooperation combinator is in fact an indexed family of combinators, one for each possible *cooperation set* L. When cooperation is not imposed, namely for action types not in L, the components proceed independently and concurrently with their enabled activities. However if a component enables an activity whose action type is in the cooperation set it will not be able to proceed with that activity until the other component also enables an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity.

In some cases, when an activity is known to be carried out in coop-

eration with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

If the cooperation set is empty, the two components proceed independently, with no shared activities. We use a compact notation—with the two cooperands separated by parallel lines—to represent this case.

PEPA is a high-level notation for Markov modelling because it is possible to generate directly from a PEPA model a continuous-time Markov process which faithfully encodes the behavioural (same number of states; same transitions between states) and temporal (same rates on the transitions) aspects of the PEPA model. Through the analysis and solution of this Markov process the modeller can undertake an experimental investigation of the system which the model represents.

Following is a PEPA example where we have rates, processes and their cooperation. Rate assignments must precede any process definition. A rate identifier is valid if it is a valid Java identifier starting with a lowercase letter and must end with a semicolon. A process identifier is valid if it is a valid Java identifier starting with an uppercase letter. As usual an identifier can represent a Choice or a Prefix. It can also identify subparts of the system when it is assigned a cooperation. Process assignments must end with a semicolon. In the given example an equation like $P1 = (a, x).P2$ is a process assignment. On the left side of the equation is $P1$ which is a process and on

the right side of the equation is an activity and resulting process $P2$; when an activity is performed process $P1$ is transformed into process $P2$. An activity includes action (a) and its rate (x). In this example processes $P1$ and $Q1$ are cooperationg over shared action (a).

```
// Rates
x = 1.0;
y = 2.0;
z = 1.0;

//Processes
P1 = (a,x).P2;
P2 = (b,y).P1;
Q1 = (a,x).Q2;
Q2 = (c,z).Q1;

//Cooperation
P1 <a> Q1
```

Based on a PEPA process we can extract performance measures such as the *steady-state solution* providing long-term probabilities for all states, the *transition throughput* giving the actual long-term frequencies at which transitions are executed, or the *passage-time* between occurrences of specific transitions.

3.5 PRISM

The PRISM [31, 32] tool is a model checker for the modelling and analysis of systems which exhibit probabilistic behaviour. The PRISM language is a state-based language. We construct PRISM models by translating GROOVE LTSs into PRISM.

The fundamental components of the PRISM language are modules and variables. A model is composed of a number of modules which can interact with each other. A module contains a number of local variables. The values of these variables at any given time constitute the state of the module. The global state of the whole model is determined by the local state of all modules. The behaviour of each module is described by a set of commands. A command takes the form:

$$[a]s = 0 -> x : (s' = 1);$$

Here $(s = 0)$ is a predicate in the model in which s is a variable with 0 as its value and $(s' = 1)$ describes a transition which the module can make if the predicate is true. A transition is specified by assigning new values to the variables in the module. Each transition is given a rate, i.e., x has rate 1.0 in the following example.

The PRISM language can be used to describe three types of probabilistic models: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). To indicate which type is being described, a PRISM model includes one of the keywords dtmc, ctmc or mdp. This is typically at the very start of the file (as in our Example below), but can actually occur anywhere in the file (except inside modules

and other declarations). PRISM supports continuous stochastic logic, where we can define properties and verify them, for example, the probability of arriving at the destination within a certain period of time, and the time within which an accident will be removed.

```
//model type
ctmc

// Rates
const double x = 1.0;
const double y = 2.0;
const double z = 1.0;

//modules
module M1
     s : [0..1];

     [a] s=0 -> x:(s'=1);
     [b] s=1 -> y:(s'=0);
endmodule

module M2
     t : [0..1];

     [a] t=0 -> x:(t'=1);
```

```
     [c] t=1 -> z:(t'=0);
endmodule

//Synchronization
system
M1 |[a]| M2
endsystem
```

3.6 Summary

In this chapter we have given a general introduction of attributed typed graphs, graph transformation, graph transition system and stochastic graph transformation. We have discussed the graph transformation tool GROOVE, in which we have modelled the Traffic Information System (TIS), our case study. The performance analysis tools PEPA and PRISM are also discussed here in which we are going to translate our GROOVE models for stochastic analysis.

Chapter 4

Methodology

In this chapter we will discuss our methodology for system modelling and decomposition into local views. We distinguish three approaches, monolithic, top-down and bottom-up. Modelling a system as a single graph transformation system is the monolithic approach; decomposing a system into views and then synchronizing them is the top-down approach and composing a system from its subsystems is the bottom-up approach.

4.1 Monolithic Approach

In the monolithic approach we model a system as a single graph transformation system. We first define the architecture of a system by means of a type graph. Then we define a start graph and a set of rules. The monolithic approach is demonstrated by our case study, which is modelled in GROOVE and translated into PEPA for analysis. We will discuss it in detail in Chapter 5. There are four steps in the monolithic approach.

(1) System Modeling as Graph Transformation

(2) Generation of the Labelled Transition System (LTS)

(3) Transforming the LTS into a CTMC

(4) Performance Analysis (see 4.3.2)

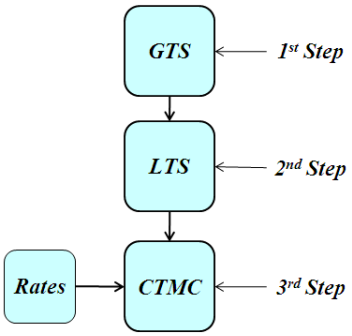


Figure 4.1: Monolithic approach

4.1.1 System Modeling as Graph Transformation

In typed graph transformation systems we define the structural model of a system as a type graph which gives the architecture and imposes constraints on instance graphs. We then define a start graph (an instance graph) and a set of rules which are applied to the start graph non-deterministically to produce the state space of the system.

4.1.2 Generation of the LTS

A labelled transition system (LTS) gives us all the possible evolutions of a system throughout its life cycle. It is generated when all applicable rules are

applied on a start graph, and all subsequent graphs, at all possible matches. In a labelled transition system states represent isomorphism classes of graphs and transitions represent rule applications. Instantiated rule signatures serve as labels.

4.1.3 Transforming the LTS into a CTMC

We translate the LTS into CTMC [4, 8], as GROOVE [38] does not support stochastic analysis of a system such as steady-state analysis, passage time analysis etc. At the same time we get another advantage from CTMC for synchronizing individual views (subsystems) to compose a global model. We can integrate individual CTMCs and synchronize them over shared labels (rule names) to compose resulting CTMC, which is bisimilar to the CTMC of the global model. In this way we get the same behaviour of the global model and avoid scalability problems which arise in the verification of large systems. PEPA provides various performance analysis techniques; but it lacks support for large files; while PRISM has support for large files.

4.2 Top-down Approach

Many system verification approaches based on state space exploration face scalability problems. We mitigate these problems by separating the model into different views. Decomposing a system into views, we then synchronize them over shared labels in the top-down approach. It is composed of the following six steps.

(1) System Modeling as Graph Transformation (see 4.1.1)

(2) Generation of Local Views

(3) Generation of Labelled Transition Systems (see 4.1.2)

(4) Transforming the LTS into a CTMC (see 4.1.3)

(5) Synchronization of Local Views

(6) Performance Analysis (see 4.3.2)

The case study model given in Chapter 5 is decomposed into different views; and these views' LTSs are later synchronized in either PEPA or PRISM. We will discuss modularity and synchronization of views in Chapter 6. Since we have already discussed steps 1, 3, 4 and 6 of top-down approach in the monolithic approach, we focus on the remaining steps here.

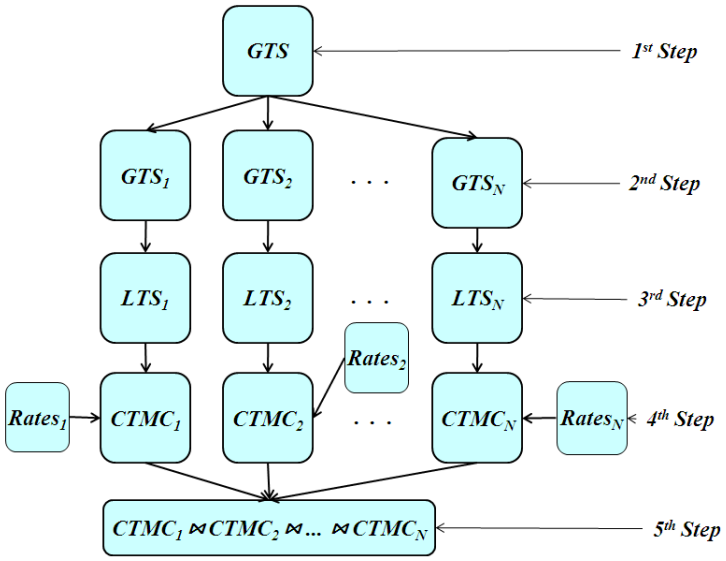


Figure 4.2: Top-down modularity approach

4.2.1 Generation of Local Views

In GROOVE a system is modelled as a single graph transformation system. Therefore, in system analysis there are usually scalability problems. We suggest and demonstrate to model a system as composition of views (subsystems). We have developed a tool that generates local views from a global system based on the views' type graphs.

For a decomposition of a system we identify subsystems (views) on the base of the core functionality they are providing and define projections of the subsystems' type graphs out of the global system's type graph. Types of nodes and edges that are required by the functionality of the view are retained and all other node types and edge types that are not used by the view are removed from the view's type graph. Often these subsystems are not disjoint, but share a common interface over which they co-operate. Rules and instances (start graphs) are then reduced according to the type graphs of local views by our View Generator tool that generates views on the basis of their local type graphs. Projected rules that are reduced to identities in one view and completely remain in the other view are removed from a view, while rules that partially remain are kept in the given views.

4.2.2 Synchronization of Local Views

In PEPA views are represented as individual sequential components. These sequential components are represented by initial states. These sequential components are merged together and are co-ordinated by the PEPA co-operation operator, e.g., P1 $\bowtie$ Q1, while in PRISM views are represented

as individual modules and PRISM synchronizes them over shared labels implicitly. PRISM also uses a CSP-based operator for synchronization, i.e., M1 || M2. We will demonstrate and discuss modularity of a system in Chapter 6.

4.3 Bottom-up Approach

Composing a system from its subsystems is the bottom-up approach. In this approach we have to identify subsystems or modules which are perspectives or parts of a system and their common interface over which subsystems cooperate. The bottom-up approach requires a composition of individual subsystems which share a common interface. It consists of the following five steps. Most of them are analogous to the monolithic and top-down approaches, so we are not repeating them here.

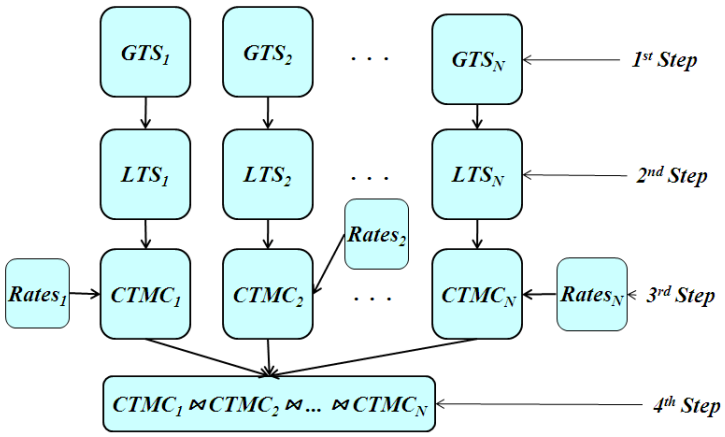


Figure 4.3: Bottom-up modularity approach

(1) Modelling of Views

(2) Generation of the Labelled Transition System (see 4.1.2)

(3) Transforming the LTS into a CTMC (see 4.1.3)

(4) Synchronization of Local Views (see 4.2.2)

(5) Performance Analysis

4.3.1 Modelling of Views

Graph transformation does not allow to model a system as composition of subsystems. Systems are modelled as whole-world/global models. By defining a common interface among subsystems, we achieve system's modularity.

4.3.2 Performance Analysis

Once we have transformed the GROOVE LTS into a PEPA or PRISM model, we can get stochastic analysis results supported by PEPA or PRISM. In PEPA we can analyze non-functional properties of mobile applications such as steady-state probabilities and throughput. Steady-state analysis determines the long term probability for the system to be in a certain state and throughput analysis gives the long term frequency of its actions (rule labels). In PRISM we can analyze the steady-state of a system. As PRISM supports continuous stochastic logic, where we can define properties and verify them, for example, the probability of arriving at the destination within a certain period of time, and the time within which an accident will be removed.

4.4 Summary

In this chapter we have presented our methodology, we have introduced three approaches monolithic, top-down and bottom-up, to system modelling. When we analyze a system through state space exploration there is usually a state space explosion problem. This is the major bottleneck in the derivation of a CTMC of a system. Therefore, we have given the top-down and the bottom-up modularity approaches for modelling a system. Both approaches are demonstrated in detail in Chapter 6.

Chapter 5

Traffic Information System: A Case Study

The Accident Assistance Scenario of the Sensoria Project [6, 28] inspired the Traffic Information System (TIS) model that we have taken as our case study. In this chapter we will discuss in detail our case study. We follow the methodology outlined in Chapter 4 for the monolithic approach.

5.1 System Modeling as Graph Transformation

In this section we will discuss how we model a system as a graph transformation system (GTS). We use graphs to model the structure of the application, including the topology of locations, the current locations of relevant devices, existing links between application components as well as their states. Our graphs come in two flavours, as type and instance graphs. A type graph pro-

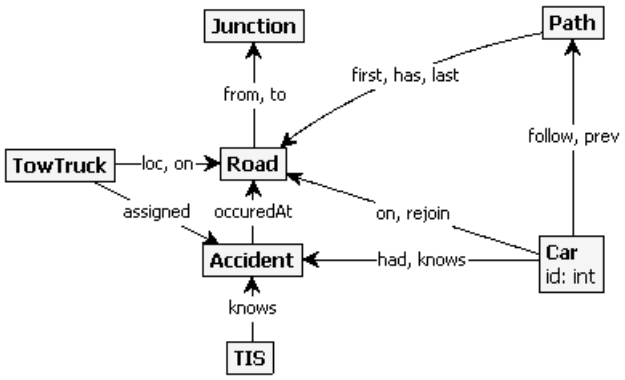


Figure 5.1: Type graph of the Traffic Information System

vides a structural model of the admissible states of a system, similar to the way a class diagram describes valid object structures.

Fig. 5.1 shows the type graph of the TIS model, with an instance graph in Fig. 5.2 representing a map of Roads and Junctions as well as two Cars following predefined Paths. The model allows to represent Accidents that can be reported to the TIS. The TIS then shares the information with other Cars. To identify Cars, these nodes have been given *id* attributes of type integer. We have given attributes to Cars only to make the model simple and to avoid large state space.

Instance graphs are transformed by rules modelling operations, distinguished into three (not strictly disjoint) categories: Mobility operations access and change the location of devices, recovery operations deal with the removal of accidents and service operations capture the state changes brought about by the sending and receiving of messages between services and their client applications. We do not model the communication itself, i.e., rules will not describe the sending and receiving of messages, but only their effects on the states of components.

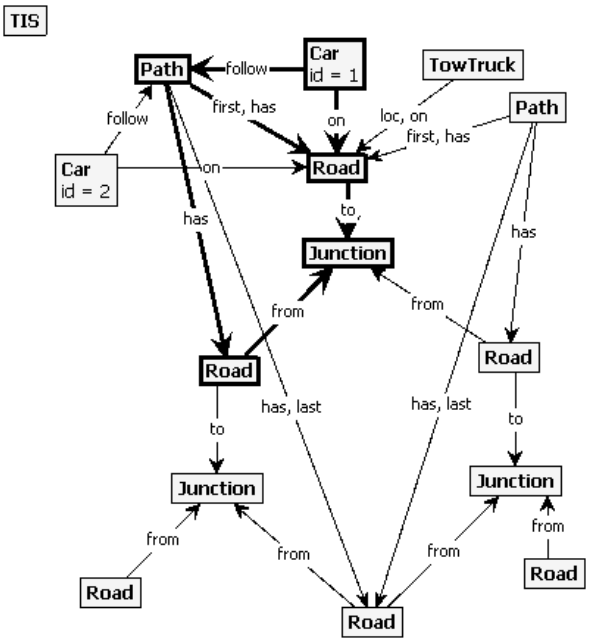


Figure 5.2: Start graph of the Traffic Information System (match for *moveCar(1)* indicated in bold)

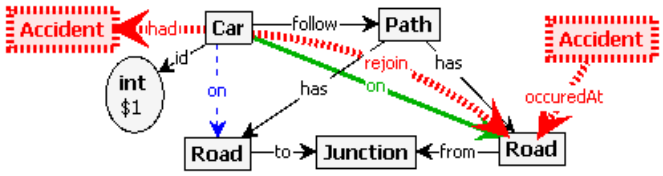


Figure 5.3: Rule $moveCar(car)$ models the mobility of a Car following a Path.

The complete model of the Traffic Information System (TIS) consists of the type graph in Fig. 5.1 and eleven rules with the following rule signatures. Parameter declarations refer to the numbered *id* attributes, with the nth formal parameter referring to the attribute value labelled $\$n$.

- *moveCar(car)*: moves the Car from one Road to the next on allocated Path, see Fig. 5.3;
- *accident(car)*: the Car suffers an Accident, see Fig. 5.5(a);

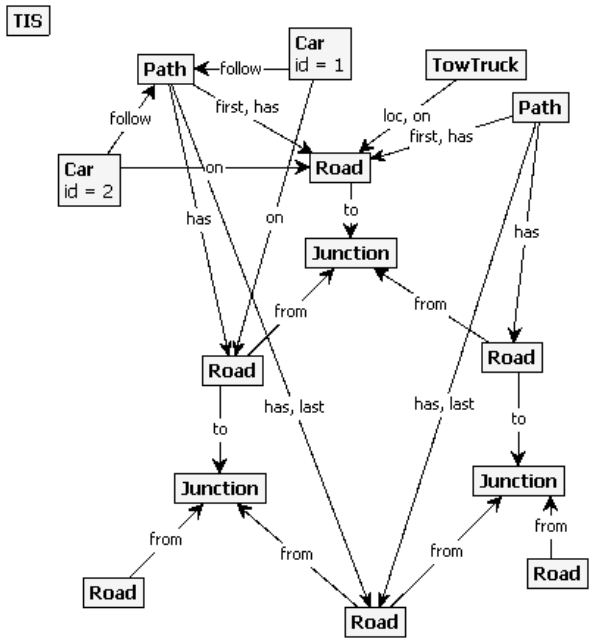
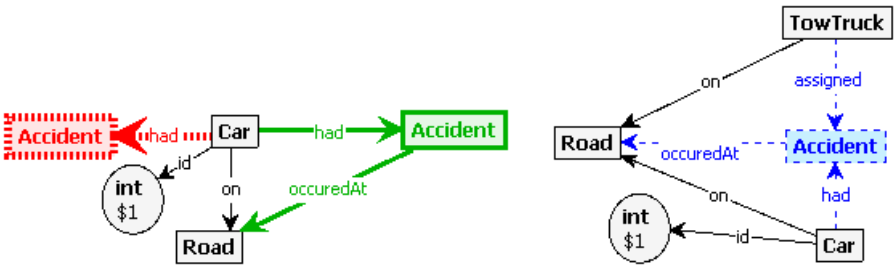


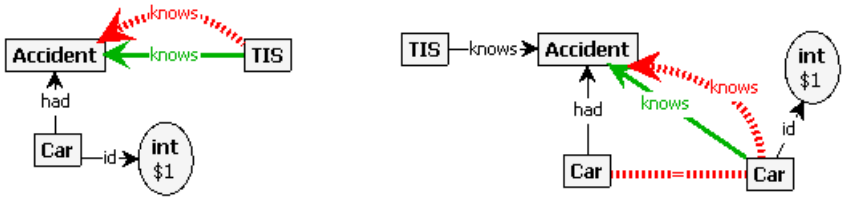
Figure 5.4: Resulting graph after applying *moveCar(1)* to the graph in Fig. 5.2

- *removeAccident(car)*: the Accident is removed, see Fig. 5.5(b);
- *getAccidentInfo(car)*: the TIS receives information about an Accident, see Fig. 5.6(a);
- *sendAccidentInfo(car)*: the TIS informs the Car about an Accident that happened to another one, see Fig. 5.6(b);
- *detour(car)*: the Car, after receiving Accident information, takes a detour leaving current Path and follows another Path, see Fig. 5.7;
- *rejoin(car)*: the Car rejoins its previous Path, leaving the detour Path after passing the Accident, see Fig. 5.8;
- *arriveAtDest(car)*: the Car arrives at its destination, i.e., the end of its



(a) *accident(car)* (b) *removeAccident(car)*

Figure 5.5: Rules modelling the occurrence and removal of Accidents.



(a) *getAccidentInfo(car)* (b) *sendAccidentInfo(car)*

Figure 5.6: Rules for the TIS to receive Accident information and pass it to other Cars

path, see Fig. 5.9;

- *assign(car)*: the TowTruck is assigned to the Car which suffered an Accident, see Fig. 5.10;
- *moveTowTruck(car)*: the TowTruck moves towards the Accident spot, see Fig. 5.11;
- *moveBackTowTruck()*: the TowTruck moves back to its garage, see Fig. 5.12;

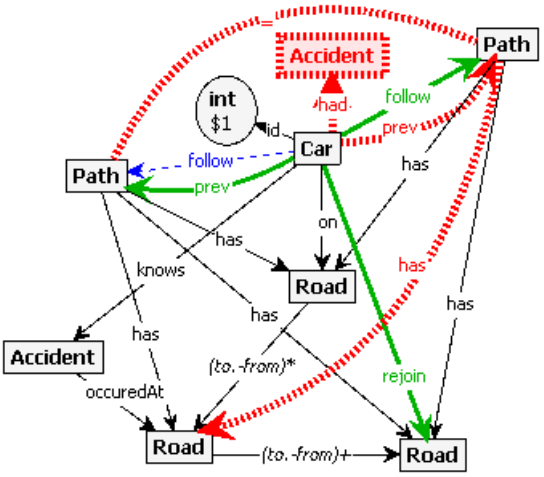


Figure 5.7: Rule $detour(car)$ shows how a Car takes a detour to avoid an Accident spot.

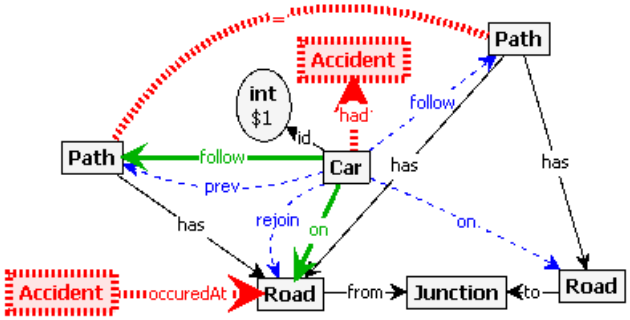


Figure 5.8: Rule $rejoin(car)$ shows how a Car rejoins its previous Path.

5.2 Generation of the LTS

After modelling a system as a graph transformation system in GROOVE, we generate an LTS from it. All possible applications of rules on the start graph and the resulting graphs produce the system's state space, which gives all possible evolutions of a system.

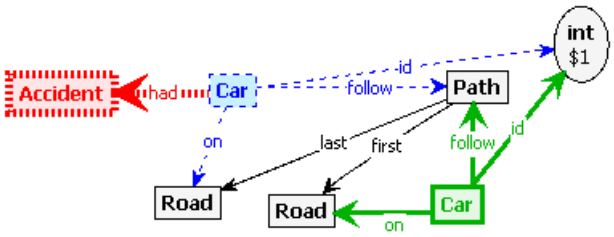


Figure 5.9: Rule $arriveAtDest(car)$ shows a Car finishing its journey.

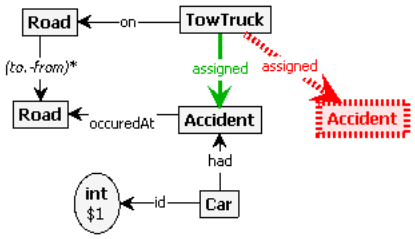


Figure 5.10: Rule $assign(car)$ depicts that TowTruck is assigned to an Accident.

5.3 Transforming the LTS into a CTMC

We have developed a tool that assigns every transition in LTS a rate (stochastic delay) and generates CTMC. This CTMC could be analyzed in any tool supporting CTMCs [36] for stochastic performance analysis.

CTMCs can be translated into any tool supporting them [36]; so we are translating these CTMCs in PEPA or PRISM for stochastic analysis, where individual views can be merged together in a single file and are synchronized over shared labels. PEPA provides various performance analysis techniques; but it lacks support for large files; while PRISM has support for large files.

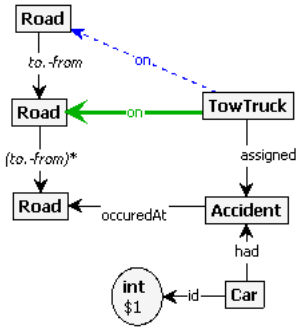


Figure 5.11: Rule $moveTowTruck(car)$ shows a TowTruck moving towards an Accident spot.

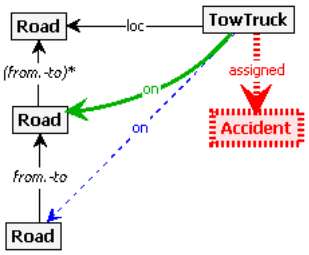


Figure 5.12: Rule $moveBackTowTruck$ depicts a TowTruck moving back to its garage.

5.4 Summary

In this chapter we have discussed the global model of our case study. This case study is modelled in the graph transformation tool GROOVE. Cars can move, Accidents can occur and be removed by Recovery service. The Traffic Information System (TIS) informs Cars about the Accident after receiving Accident information so that Cars may change their route to avoid the difficulties and inconveniences. Cars may rejoin their previous Path, once they have passed the Accident, which they may follow until they arrive at their destinations.

Chapter 6

Modularization: Top-down and Bottom-up Approaches

Scalability problems occur due to state space explosion, where the size of a system grows exponentially. These problems can be tackled by introducing a view-based approach to a system modelling.

6.1 Top-down Approach

In the top-down approach, a system is modelled as a graph transformation system and then it is decomposed into its subsystems. LTSs of subsystem are transformed into CTMCs and later merged together in any tool supporting CTMCs. The top-down approach has the six steps as described in Chapter 4. Some steps are similar to the monolithic approach in Chapter 5, so we are not repeating them here.

6.1.1 Generation of Local Views

Views are perspectives on a real system representing a particular concern or part of the system to focus attention and capability for analysis. In our case we can distinguish three perspectives, the Car and its location and mobility, the Recovery Service that removes accidents and the Service view that sends news about Accidents.

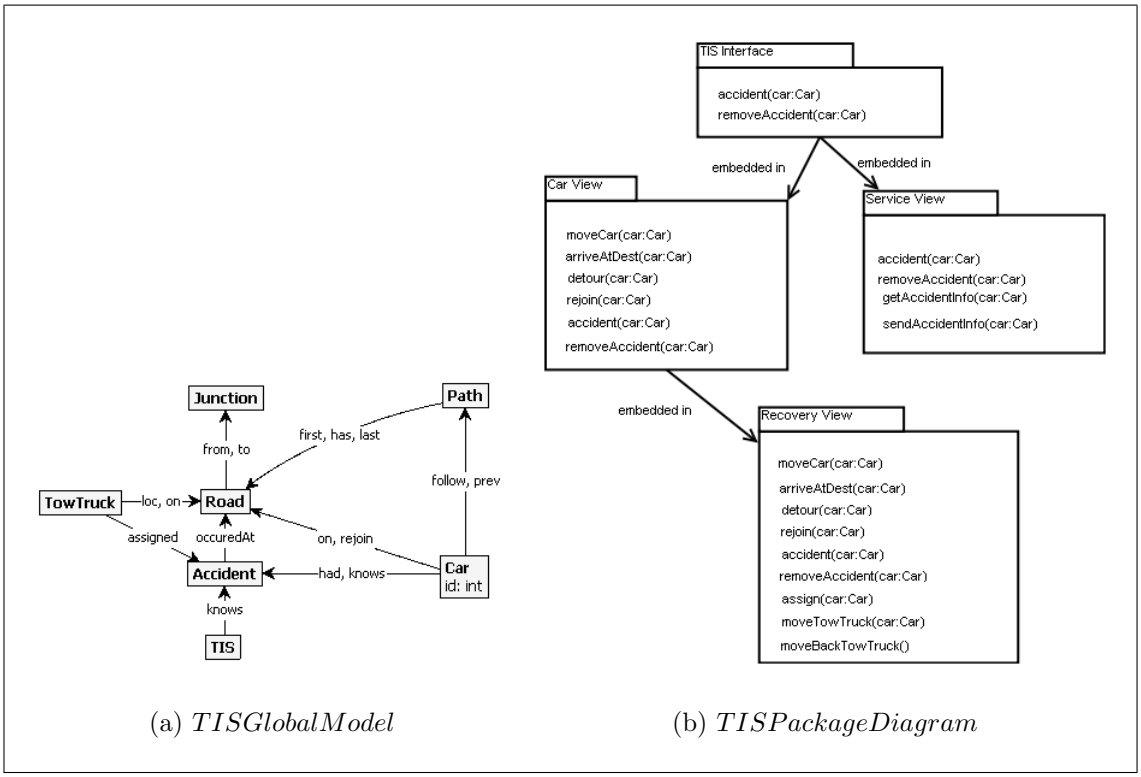
Car View models the dynamicity of the TIS which shows mobility along the Path. Cars may have Accidents before arriving at their destinations. The Car view receives Accident information by the Service view.

Service View sends up-to-date accident information to Cars so that they may take a detour and avoid any delay or inconvenience caused by Accidents on their route journey.

Recovery View extends Car view because it needs all the information about Accidents, i.e., at what places Accidents are occurring and with what frequency/ratio these Accidents are taking place on the roads of the given road network. The Recovery view extends the *removeAccident(car:Car)* rule. All other rules of the Car view remain unaltered.

Fig. 6.2 shows the decomposition of a global graph into its subgraphs, namely Car, Service and Recovery views' graphs. These subgraphs are typed over Car, Service and Recovery views' type graphs which are actually subtype graphs of the global type graph of our case study.

In the global model of a system, intuitively a view is defined by identifying



(a) *TISGlobalModel* (b) *TISPackageDiagram*

Figure 6.1: Type graph of the Traffic Information System and TIS package diagram

in the global type graph of Fig. 6.1(a) the node and edge types that should be abstracted from, i.e., that are not required by the functionality of the view. This includes node types TIS and TowTruck and edge type *knows* in the Car view of Fig. 6.3(a); node type TIS in the Recovery view of Fig. 6.3(b); and node types Road, Junction, Path and TowTruck in the Service view of Fig. 6.3(c). We have developed a tool that requires the type graph of a view and couples the projection of global graphs based on the view's local type graph automatically. It reduces start graphs and rules to the remaining types, removing all instances of types that are no longer present in the view's type graph.

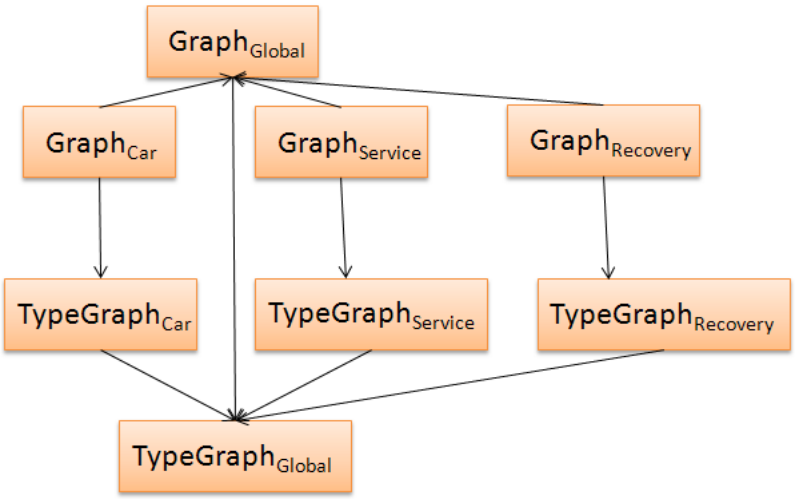
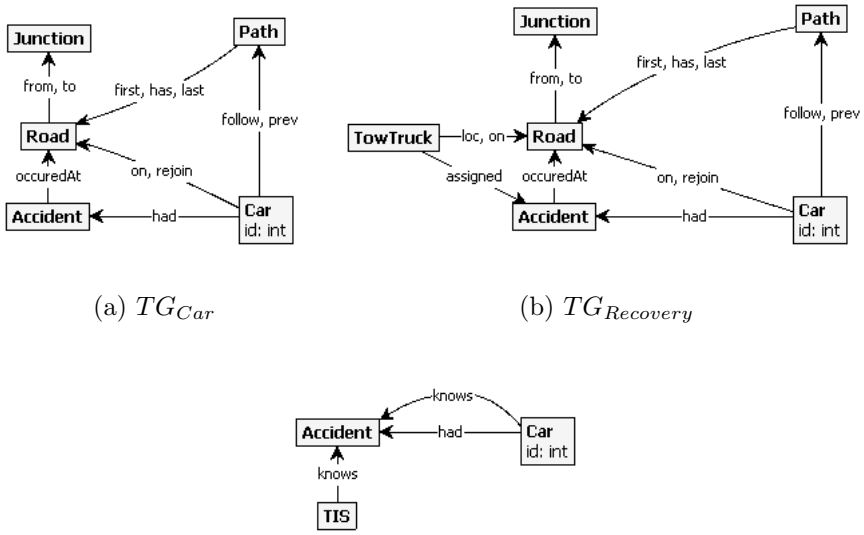


Figure 6.2: Depicting the decomposition of a global graph into subgraphs which are typed over the views' local type graphs

In our case, this is illustrated in Fig. 6.4(a) showing the projection of the global graph in Fig. 5.2 to TG_{Car} in Fig. 6.3(a), Fig. 6.4(b) shows the projection of the same global graph to $TG_{Recovery}$ in Fig. 6.3(b). Similarly, Fig. 6.4(c) shows the projection of the same global graph to $TG_{Service}$ in Fig. 6.3(c).

We generate the projection automatically from the GROOVE model by deleting all nodes and edges which are not part of the projection. Our View Generator tool deletes all incident edges of deleted nodes too, so that there are no dangling edges. It also deletes NACs and attributes of the deleted nodes as there should not remain any dangling nodes.

Let us consider what happens when we apply this definition to the global TIS model, using (sub) type graph TG_{Car} in Fig. 6.3(a) for the projection. Rules *moveCar(car)*, *accident(car)*, *rejoin(car)*, and *arriveAtDest(car)* only contain elements of types occurring in TG_{Car}, so they are kept unchanged as



(a) TG_{Car} (b) $TG_{Recovery}$

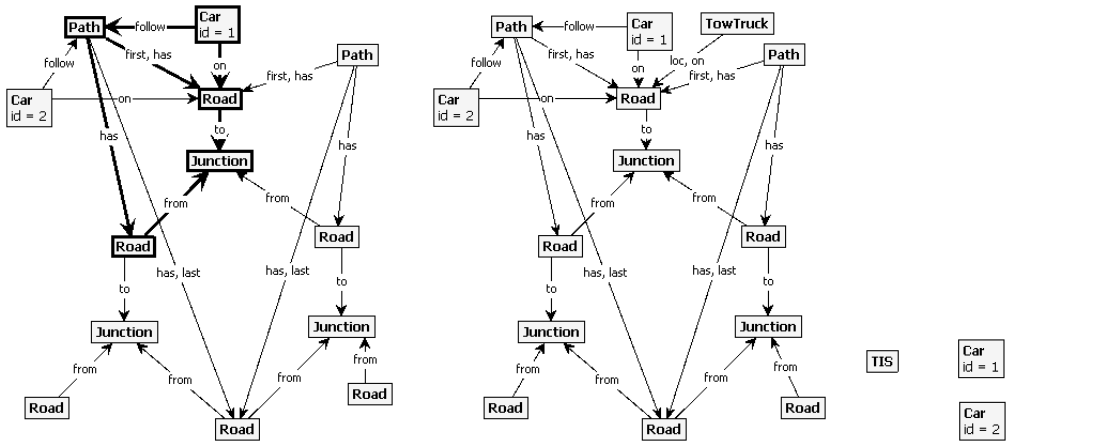
(c) $TG_{Service}$

Figure 6.3: Type graphs of *Car*, *Recovery* and *Service* views

part of the Car view, as shown in Fig. 5.3, Fig. 5.5(a), Fig. 5.8, and Fig. 5.9 respectively.

In the Car view, rules *getAccidentInfo(car)* and *sendAccidentInfo(car)*, *assign(car)*, *moveTowTruck(car)* and *moveBackTowTruck()* are reduced to identities, so they do not have any effect on the graph and (as we shall see later) are not needed for synchronization with rules in the Service view and Recovery view. Rule *detour(car)* in Fig. 5.7 is retained as it is, except for the single *knows* edge, which is not part of the Car view and rule *removeAccident(car)* looses *TowTruck* node and its two edges *on* and *assigned* as seen in Fig. 6.5

The Recovery view extends the Car view so it has all the rules of the Car view as they are except *removeAccident(car)* rule which is extended as shown in the Fig. 5.5(b). In addition the Recovery view has *assign(car)*, *move-*



(a) $StartGraph_{Car}$ (b) $StartGraph_{Recovery}$ (c) $StartGraph_{Service}$

Figure 6.4: Start graphs of *Car*, *Recovery* and *Service* views

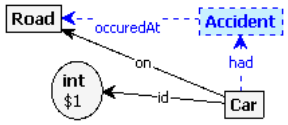
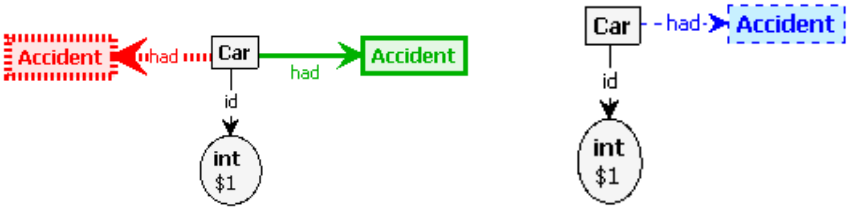


Figure 6.5: In Car view *removeAccident* rule is the projection of the global rule of Fig. 5.5(b).

TowTruck(car) and *moveBackTowTruck()* rules which are given in Fig. 5.10, Fig. 5.11 and Fig. 5.12 respectively. Rules *getAccidentInfo(car)* and *sendAccidentInfo(car)* are reduced to identities and are not needed by the synchronization with rules in the Service view and Car view, so they are removed from this view.

For the Service view, type graph $TG_{Service}$ in Fig. 6.3(c) yields a projection where only rules *getAccidentInfo(car)* and *sendAccidentInfo(car)* are kept unchanged. Rules *moveCar(car), detour(car), rejoin(car), arriveAtDest(car), assign(car), moveTowTruck(car)* and *moveBackTowTruck()* are reduced to identities without visible effect, while rules *accident(car) and removeAcci-*



(a) accident(car) (b) removeAccident(car)

Figure 6.6: Service view projections of *accident*(*car*), *removeAccident*(*car*)

dent(car) survive in reduced form, as seen in Fig. 6.6.

In general, if a rule is completely retained in one view and does not change the graph in the other view, it can be removed from the other view altogether. This is an optimization which does not change the semantics of the overall model, but reduces the cost of generating the state space. This is the case for *moveCar(car)*, *arriveAtDest(car)* and *rejoin(car)* which are only kept in the Car view; while *getAccidentInfo(car)* and *sendAccidentInfo(car)* rules which are only kept in the Service view. For example, in Fig. 6.4(a) the start graph of the Car view indicates a match for *moveCar(1)*. It coincides with the match shown in the global start graph in Fig. 5.2 because the applicability and effect of the rule can be determined entirely in the Car view. Instead, in the start graph for the Service view shown in Fig. 6.4(c), the corresponding match shows that *moveCar(1)* would not have any effect on the state and can therefore be ignored. Since the Recovery view extends the Car view, it has all the rules of the Car view as well as *assign(car)*, *moveTowTruck(car)* and *moveBackTowTruck()*.

We assume (as happens in our example) that node types equipped with *id*

attributes that are used as parameters (just *Car* in our case) are preserved in the projection. This ensures that the actual parameters in labels are preserved and are used consistently in local views, so that synchronisation over shared labels leads to the correctly composed model in PEPA and PRISM.

6.1.2 Synchronization of Local Views

In both PEPA and PRISM individual subsystems are integrated and are synchronized over shared labels. In PEPA after generating sequential PEPA processes from the Car, the Recovery and the Service view, the resulting processes `Car`, `Recovery` and `Service` are synchronised using the PEPA co-operation operator, i.e., as `Car` ⋈ `Recovery` ⋈ `Service`. That means that transitions carrying labels shared between processes must be executed simultaneously, while transitions whose labels occur only in one of the processes are independent. In PEPA we take initial states of a sequential PEPA processes as sequential PEPA components which are actually the initial states of views LTSs. The synchronization is performed over these sequential components.

In PRISM, after generating modules from individual views, these modules are merged together in a single PRISM file. PRISM synchronizes individual modules over shared labels implicitly and it also uses CSP-based operators for synchronization of modules, e.g., M1 || M2 || M3. For synchronization in PRISM initial states of modules should be the same as they are in the GROOVE LTSs of views.

6.2 Bottom-up Approach

Composing a system from subsystems which share a common interface is a bottom-up approach. Instead of modelling a large system and then decomposing it into its subsystems, it is better practice that we should model a system as a set of subsystems which share a common interface. Then LTSs generated from these subsystems can be transformed into CTMCs. These CTMCs can be synchronized over shared labels in any tool supporting them [36].

The bottom-up approach has the five steps as described in Chapter 4. Most of them are analogous to the monolithic and top-down approaches, so we are not repeating them here.

6.2.1 Modelling of Views

In software engineering, usually systems are modelled as the composition of components/subsystems. Though graph transformation is a very powerful specification formalism for modelling software systems, it lacks compositionality. We suggest and demonstrate that if we define a common interface among subsystems, we can synchronize these subsystems over a shared interface in any tool supporting the synchronization. In our case, we make use of PEPA and PRISM for synchronization of subsystems, where CTMCs generated from LTSs of subsystems are synchronized over shared labels.

Fig. 6.1(b) shows the three views of the TIS system plus their common interface. Rules describe the functionality of each view and the TIS interface defines the shared labels over which local views can be synchronized. Here

shared labels are *accident(car)* and *removeAccident(car)*. The TIS interface is embedded in the Car view and Service view, and the Recovery view extends the Car view, so it is in the Recovery view by default. The Car view shows that a Car can receive Accident information, take a detour to avoid the Accident, etc. The Recovery view depicts that Accidents are assigned to the Recovery service to be removed. The Service view describes how the Accident information is shared by the TIS. As we have already given the views of our case study in the top-down approach, we are not repeating them here.

6.2.2 Performance Analysis

Table 6.1 and Fig. 6.7 show the probability distributions of the TIS global model vs the TIS synchronized model. Here, random variable K has a unit of *times per day*. It shows that there is 0.78 probability of a Car arriving at its destination in nearly 14.4 minutes. It also demonstrates that the synchronized model and the global model are equal. The numerical variations are down to the numerical method used by PRISM and cut-off points for the synchronised and global models.

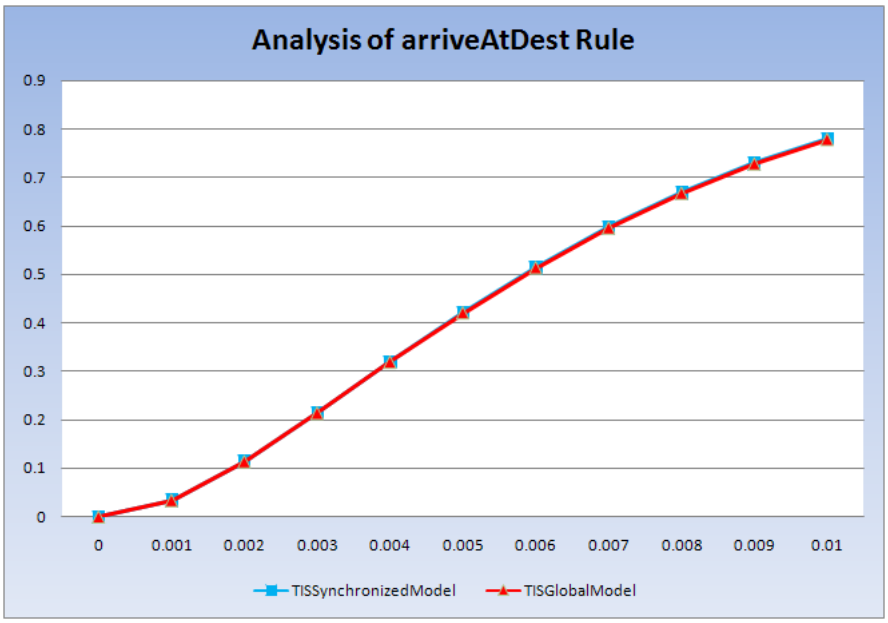


Figure 6.7: The TIS global model vs the TIS synchronized model and their probability distributions of *arriveAtDest* rule

Table 6.1: The TIS global model vs the TIS synchronized model and their probability distributions of *arriveAtDest* rule

K	TISSynchronizedModel	TISGlobalModel
0	0	0
0.001	0.034291191	0.034284062
0.002	0.113960078	0.113911438
0.003	0.214079389	0.213952186
0.004	0.319367304	0.319135669
0.005	0.420924055	0.420573353
0.006	0.513993612	0.513516955
0.007	0.596444815	0.595839628
0.008	0.66775315	0.667018739
0.009	0.728328087	0.72746419
0.01	0.77907684	0.778082931

6.3 Summary

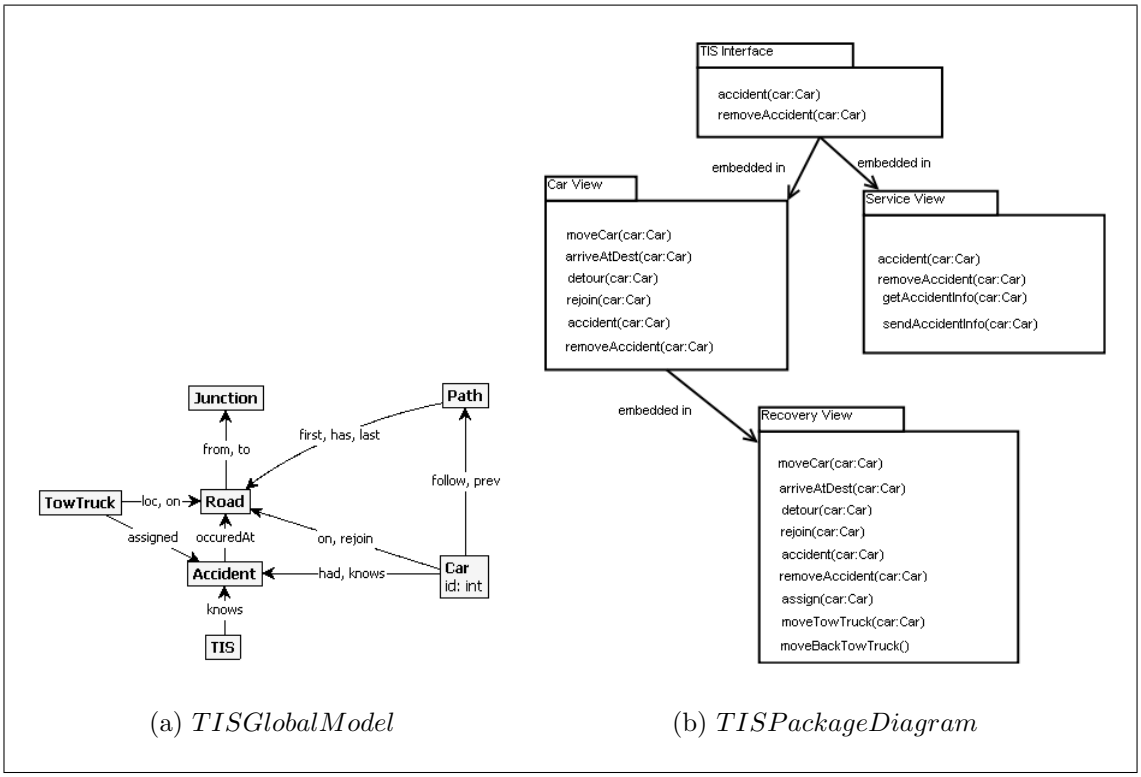
In this chapter we have demonstrated and explained in detail the modularity approaches. We have decomposed the global system of our TIS model into subsystems, i.e., Car view, Recovery view and Service view through our View Generator tool, which supports the projection of graph transformation systems based on the type graphs of corresponding views. These views are transformed into PEPA models or PRISM models. In PEPA or PRISM these views are synchronized over shared labels to compose a global model. We have also discussed the bottom-up approach where subsystems/views share a common interface. Synchronization of views is achieved by the shared labels of a common interface in PEPA or PRISM.

Chapter 7

Composition and Decomposition

In this chapter we are giving the formalization of our modularity approaches. We decompose graph transformation system into views. Transition systems can be generated separately for different views which, when synchronised using a CSP-like operator, yield a system that is bisimilar to the original global system.

In the top-down approach, a view is defined by identifying in the global type graph of Fig. 7.1(a) the node and edge types that should be abstracted from, i.e., that are not required by the functionality of the view. Our View Generator tool projects the start graphs and rules based on the subtype graphs of local views. While in bottom-up approach, we have to identify a common interface and subsystems (views), which could be composed to get the global system as depicted in the Fig. 7.1(b). Where compositionality is achieved by synchronizing subsystems over the common interface. We have



(a) $TISGlobalModel$ (b) $TISPackageDiagram$

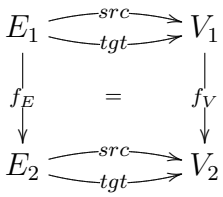
Figure 7.1: Type graph of the Traffic Information System and TIS package diagram

identified views as Car view, Service view and Recovery view.

7.1 Typed Attributed Graph Transformation

This section introduces the basic notions of typed attributed graph transformation with negative application conditions. The formalisation follows the algebraic approach [14]. A graph is a tuple (V, E, src, tgt) where V is a set of nodes (or vertices), E is a set of edges and $src, tgt : E \to V$ associate, respectively, a source and target node for each edge in E. Given graphs G_1 and G_2, a graph morphism is a pair (f_V, f_E) of total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that source and targets of edges are preserved as

shown below.



An E-graph is a graph equipped with an additional set V_A of data nodes (or values) and special sets of edges E_{EA} (edge attributes) and E_{NA} (node attributes) connecting, respectively, edges in E and nodes in V to values in V_A. An attributed graph is a tuple (EG, A) where EG is an E-graph and A is an algebra with signature $\Sigma = (S, OP)$ such that $\biguplus_{s \in S} A_s = V_A$. Intuitively, an attributed graph is an E-graph where V_A is the set of all data values available for attribution. A morphism $f : (EG, A) \to (EG', A')$ of attributed graphs is a pair of an E-graph morphism $f_{EG} : EG \to EG'$ and a compatible algebra homomorphism $f_A : A \to A'$. Fixing an attributed graph TG as *attributed type graph*, we define the category $\mathbf{AGraph_{TG}}$ of TG-typed attributed graphs [14]. Objects are pairs (G, t) of attributed graphs G with typing homomorphisms $t : G \to TG$ and morphisms $f : G \to H$ are attributed graph morphisms compatible with the typing.

We denote by $X = (X_s)_{s \in S}$ a family of countable sets of variables, indexed by sorts $s \in S$, and write $(x : s) \in X$ for $x \in X_s$. A TG-typed graph transformation rule (or production) over X is a span $L \xleftarrow{l} K \xrightarrow{r} R$ where l, r are monomorphisms, the algebra component of L, K, R is $T_\Sigma(X)$ (the term algebra of Σ with variables in X) such that $l_A = r_A = id_{T_\Sigma(X)}$. That means, variable names are preserved across the rule. The class of all rules

over TG with variables in X is denoted $Rules(TG, X)$.

The operational semantics of rules is defined by the double-pushout construction. Given a TG-typed graph G and graph production $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ together with a match (a TG-typed graph morphism) $m : L \rightarrow G$, a *direct derivation* $G \overset{p,m}{\Longrightarrow} H$ exists if and only if the diagram below can be constructed, where squares (1) and (2) are pushouts in $\mathbf{AGraph_{TG}}$ such that G, C, H share the same algebra A and the algebra components l^*_A, r^*_A of morphisms l^*, r^* are identities on A. This ensures that data elements are preserved across derivation sequences, which allows their use as actual parameters in a global namespace. We also write $G \overset{p/d}{\Longrightarrow} H$ to refer to the entire DPO diagram $d = (d_L, d_K, d_R)$. A derivation is a sequence $G_0 \overset{p_1,\, m_1}{\Longrightarrow} G_1 \overset{p_2,\, m_2}{\Longrightarrow} \ldots \overset{p_n,\, m_n}{\Longrightarrow} G_n$ of direct derivations.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m=d_L}\downarrow & (1) & \downarrow{\scriptstyle d_K} & (2) & \downarrow{\scriptstyle m^*=d_R} \\ G & \xleftarrow[l^*]{} & C & \xrightarrow[r^*]{} & H \end{array} \qquad \begin{array}{ccc} L & \overset{n}{\rightarrow} & \hat{L} \\ {\scriptstyle m}\downarrow & \swarrow_{q} & \\ G & & \end{array}$$

A *negative constraint* on a TG-typed graph L is a morphism $n : L \rightarrow \hat{L}$ over TG. A morphism $m : L \rightarrow G$ satisfies n (written $m \models n$) if there is no morphism $q : \hat{L} \rightarrow G$ such that $q \circ n = m$, A negative application condition (NAC) over L is a set of negative constraints N. A morphism $m : L \rightarrow G$ *satisfies* N (written $m \models N$) if m satisfies every constraint in N, i.e., $\forall n \in N : m \models n$.

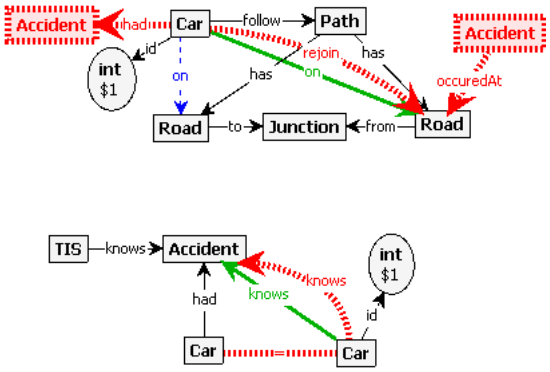


Figure 7.2: Top to bottom: Rules *moveCar* and *sendAccidentInfo*

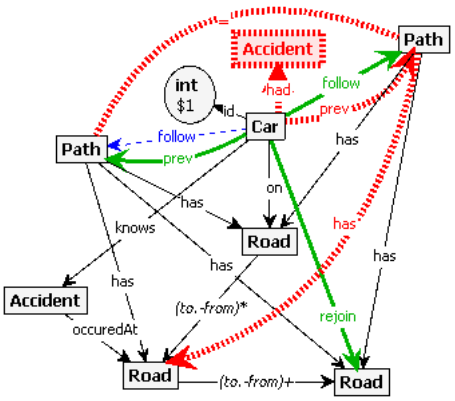


Figure 7.3: Rule *detour* for Car to avoid location of Accident

7.2 Signatures and Systems

We introduce a notion of observation on transformation steps based on rule names with parameters, whose declarations are collected in a transformation signature. Throughout the rest of the paper we assume a signature $\Sigma = (S, OP)$, a family $X = (X_s)_{s \in S}$ of countable sets of variables and a Σ-algebra A.

Definition 1 (transformation signature, labels) *A* transformation signature *(over Σ, X, A) is a tuple $\mathcal{S} = (TG, P, \sigma)$ where*

- *TG is an attributed type graph,*
- *P is a countable set of rule names,*
- *$\sigma : P \longrightarrow X^*$ assigns each rule name a list of parameters $\sigma(p) = (x_1 : s_1, \ldots, x_n : s_n)$ with $x_i \in X_{s_i}$ for $1 \leq i \leq n$.*

For $p \in P$ with $\sigma(p) = (x_1 : s_1, \ldots, x_n : s_n)$ we also write $p(x_1 : s_1, x_2 : s_2, \ldots, x_n : s_n) \in \mathcal{S}$.

Given a rule signature $p(x_1 : s_1, \ldots, x_n : s_n) \in \mathcal{S}$ *we denote by* $p(A)$ *the set of all rule labels* $p(a_1, \ldots, a_n)$ *with* $a_i \in A_{s_i}$*. The label alphabet* L *over* $\mathcal{S}$ *is the union over all rule labels* $\bigcup_{p \in P} p(A)$*.*

For example, the signatures of the rules in Fig. 7.2 are $moveCar(\$1 : int)$ and $sendAccidentInfo(\$1 : int)$, both referring to the *id* attribute of Car nodes. A graph transformation system over a signature adds definitions of rules with application conditions for all rule names.

Definition 2 (TAGTS) *Given a transformation signature* $\mathcal{S} = (TG, P, \sigma)$*, a* typed attributed graph transformation system *(TAGTS) over* $\mathcal{S}$ *is a tuple* $\mathcal{G} = (\mathcal{S}, \pi, N)$ *where*

- $\pi : P \longrightarrow Rules(TG, X)$ *assigns each rule name a span* $sp = L \xleftarrow{l} K \xrightarrow{r} R$ *over* TG *and* X*.*
- $N = (N_p)_{p \in P}$ *is a family of application conditions such that for* $\pi(p)$ *as above,* N_p *is a NAC over* L*.*

Observations on transformations are defined by $obs(G \overset{p,m}{\Longrightarrow} H) = p(a_1, a_2, \ldots, a_n)$ *if* $\sigma(p) = (x_1 : s_1, \ldots, x_n : s_n)$ *and* $a_i = m(x_i)$*. A step* $t = (G \overset{p,m}{\Longrightarrow}_{\mathcal{G}} H)$ *is a transformation such that match* m *satisfies* N_p*. A derivation is a sequence* $G_0 \overset{p_1,m_1}{\Longrightarrow}_{\mathcal{G}} \ldots \overset{p_n,m_n}{\Longrightarrow}_{\mathcal{G}} G_n$*, also written* $G_0 \Longrightarrow^*_{\mathcal{G}} G_n$*. The pair* $(\mathcal{G}, G_0)$ *with* G_0 *a* TG*-typed graph is called a* typed attributed graph grammar (TAGG) *over* $\mathcal{S}$*. Grammar* $(\mathcal{G}, G_0)$ *is* deterministic *if for all reachable graphs* G *and transformations via* $p \in P$*,* $obs(G \overset{p,m}{\Longrightarrow}_{\mathcal{G}} H) = obs(G \overset{p,m'}{\Longrightarrow}_{\mathcal{G}} H')$ *implies* $m = m'$*.*

Spans are attributed over $T_\Sigma(X)$ and parameters $x_i \in X_{s_i}$ are from the same set, so they refer to the variables used in attribute expressions in rules.

The algebra component of attributed graphs is preserved by transformations, allowing actual parameters to be used globally.

Grammars are deterministic if labels carry enough information to determine the match into any graph reachable from the start graph. While the notion abstracts from the mechanism by which this happens, we employ *id* attributes on nodes, referred to by rule parameters as part of transformation labels.

We will see in Sec. 7.4 that the notion of deterministic grammars is required for the composition of grammars to be semantically meaningful. The requirement will be imposed on the interface (intersection) only of the two grammars to be composed, rather than on the component grammars themselves.

Applying the same rule at the same match leads to isomorphic DPO diagrams and resulting graphs. The transition system generated by a TAGTS identifies isomorphic graphs, so deterministic TAGTS lead to deterministic transition systems.

Definition 3 (induced labelled transition system) *Let* $(\mathcal{G}, G_0)$ *be a TAGG and* $\mathcal{G} = (TG, P, \sigma, \pi, N)$*. The* labelled transition system $LTS(\mathcal{G}, G_0)$ *is given by* $(L, S, T, [G_0])$ *where*

- S *is the set of all isomorphism classes of graphs reachable from* G_0*, i.e.* $S = \{\ [G_n] \mid G_0 \Longrightarrow_{\mathcal{G}}^{*} G_n\}$*;*
- L *is the label alphabet over* $\mathcal{S}$*;*
- $T \subseteq S \times L \times S$ *is the transformation relation, where* $\langle [G], l, [H] \rangle \in T$ *if there is a transformation step* $t = (G \overset{p,m}{\Longrightarrow} H)$ *with* $obs(t) = l$*;*

- $[G_0]$ *is the isomorphism class of the initial graph* G_0.

7.3 Projection and Composition of Systems

Given a subgraph $TG' \subseteq TG$, a TG-typed instance graph G can be projected to an instance $G|_{TG'}$ of TG' by removing all elements of G whose types are in TG, but not in TG'. This projection, formally the inverse image of TG' under the morphism typing G in TG, can be described abstractly as a pullback of $TG' \subseteq TG$ and G's typing morphism. The projection extends to morphisms, making it a functor $\_|_{TG'} : \mathbf{AGraph}_{TG} \to \mathbf{AGraph}_{TG'}$ [22].

The functor can be used to define the projection of graph transformation rules and application conditions to a subgraph of their current type graph. We define a more general projection based on a subsignature, which allows to reduce the set of rule names as well as the type graph.

Definition 4 (subsignature, projection) *A* transformation signature $\mathcal{S}' = (TG', P', \sigma')$ *is a subsignature of* $\mathcal{S} = (TG, P, \sigma)$*, written* $\mathcal{S}' \subseteq \mathcal{S}$*, iff* $TG' \subseteq TG$*,* $P' \subseteq P$*, and* $\sigma' = \sigma|_{P'}$.

Given a TAGTS $\mathcal{G} = (\mathcal{S}, \pi, N)$ *we define the restriction* $\mathcal{G}|_{\mathcal{S}'} = (\mathcal{S}', \pi', N')$ *of* $\mathcal{G}$ *to* $\mathcal{S}'$ *for all* $p \in P'$ *by*

- $\pi'(p) = L|_{TG'} \xleftarrow{l|_{TG'}} K|_{TG'} \xrightarrow{r|_{TG'}} R|_{TG'}$ *if* $\pi(p) = L \xleftarrow{l} K \xrightarrow{r} R$
- $N'_p = \{n|_{TG'} : L|_{TG'} \to \hat{L}|_{TG'} \mid n : L \to \hat{L} \in N_p$ *s.t. the diagram below is a pushout.*$\}$

$$\begin{array}{ccc} \hat{L}|_{TG'} & \xleftarrow{n|_{TG'}} & L|_{TG'} \\ \downarrow & & \downarrow \\ \hat{L} & \xleftarrow{n} & L \end{array}$$

The projection is sound *if for all rules $p \in P \setminus P'$, it reduces $\pi(p)$ to a span of isomorphisms. A constraint n such that the diagram above forms a pushout is* preserved by the projection *to TG'.*

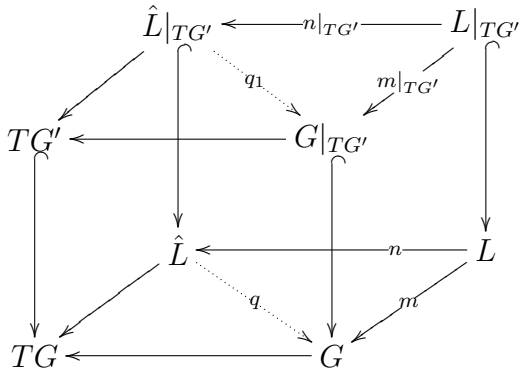
For a projection to be sound, i.e., to preserve transformations, we must only drop rules that are rendered idle, such as *moveCar, detour* from the Service view signature and *sendAccidentInfo* from the Car view signature. Constraints of retained rules are dropped if they are not preserved, i.e., if they loose any negative elements $\hat{L} \setminus n(L)$.

Lemma 1 (projection of steps) *Assume signatures and systems $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{G}$ as above such that the projection $\mathcal{G}' = \mathcal{G}|_{\mathcal{S}'}$ is sound. Then, for each step $t = (G \overset{p,m}{\Longrightarrow}_{\mathcal{G}} H)$, either there exists a step $t|_{\mathcal{S}'} = (G|_{TG'} \overset{p,m|_{TG'}}{\Longrightarrow}_{\mathcal{G}'} H|_{TG'})$ with $obs(t) = obs(t|_{\mathcal{S}'})$, or $G|_{TG'}, H|_{TG'}$ are isomorphic.*

Proof 1 *If $p \in P \setminus P'$ the projection of $\pi(p)$ is a span of identities. Since* **AGraph** *is adhesive [33], the projection functor preserves pushouts where one of the given morphisms is mono, such as the pushouts in a double-pushout diagram. That means, the projection of double pushout t is a DPO over TG' where the top span is made of identities. Since pushouts preserve isos, graphs $G|_{TG'}, H|_{TG'}$ at the bottom of the DPO are isomorphic.*

If $p \in P'$ we have $t|_{\mathcal{S}'}$ because (1) the projection functor preserves double pushouts and (2) $m|_{TG'} \models N'_p$ iff $m \models N_p$. To see (2), consider the cube diagram below, whose back face represents the pushout of Def. 4.1 while the sides and front are pullbacks arising from the projection functor. The back square, with its two opposing monos, is a pullback too. Using the pushout property of the back square it follows that existence of q_1 commuting the

top triangles implies existence of q commuting the bottom triangles and the resulting diagonal vertical square. The reverse implication can be established with the help of the pullback property of the front square. That means, m satisfies n iff $m|_{TG'}$ satisfies $n|_{TG'}$.



Composition of systems is based on composition of rules, defined by the union of two rules agreeing on a common projection. Constraints of the two given rules have to be transferred to the enlarged left-hand side of the new rule.

Definition 5 (extension and union) *Assume rule span* $s_1 = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *embedded into* $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ *by inclusions* $i_{L_1}, i_{K_1}, i_{R_1}$ *as in the diagram below, denoted* $s_1 \subseteq s$.

If n_1 *is a constraint over* L_1*, the extension of* n_1 *to* L *is defined by the pushout (1) where* $i_{\hat{L}_1}$ *is the inclusion of* $\hat{L}_1$ *into* $\hat{L}$ *and* n *is the extended constraint over* L*. If* N_1 *is a set of constraints over* L_1*, we denote by* N_1^L *the*

corresponding set of constraints extended to L.

$$\begin{array}{ccccccc} \hat{L}_1 & \xleftarrow{n_1} & L_1 & \xleftarrow{l_1} & K_1 & \xrightarrow{r_1} & R_1 \\ {\scriptstyle i_{\hat{L}_1}}\downarrow & (1) & \downarrow{\scriptstyle i_{L_1}} & & \downarrow{\scriptstyle i_{K_1}} & & \downarrow{\scriptstyle i_{R_1}} \\ \hat{L} & \xleftarrow{n} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \end{array}$$

The union of two rule spans s_1 and s_2 with $s_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ is defined if they have a well-defined intersection, that is, the componentwise intersection $s_1 \cap s_2 = L_1 \cap L_2 \xleftarrow{l_1 \cap l_2} K_1 \cap K_2 \xrightarrow{r_1 \cap r_2} R_1 \cap R_2$ is a rule span. In this case, their union is $s_1 \cup s_2 = L_1 \cup L_2 \xleftarrow{l_1 \cup l_2} K_1 \cup K_2 \xrightarrow{r_1 \cup r_2} R_1 \cup R_2$ where $X_1 \cup X_2$ is the pushout of X_1, X_2 over $X_1 \cap X_2$ with inclusions as morphisms, for $X \in \{L, K, R\}$.

Extending this notion from rules to systems, we arrive at the composition of TAGTS.

Definition 6 (composition of TAGTS) *Assume a signature $\mathcal{S}$ with subsignatures $\mathcal{S}_1 \subseteq \mathcal{S}$ and $\mathcal{S}_2 \subseteq \mathcal{S}$. Let $\mathcal{S}_0 = (TG_0, P_0, \sigma_0)$ be defined by component-wise intersection $TG_0 = TG_1 \cap TG_2$, $P_0 = P_1 \cap P_2$ and $\sigma_0 = \sigma_1|_{P_0} = \sigma_2|_{P_0}$.*

Given two systems $\mathcal{G}_1 = (\mathcal{S}_1, \pi_1, N_1)$ and $\mathcal{G}_2 = (\mathcal{S}_2, \pi_2, N_2)$, their composition $\mathcal{G}_1 \oplus \mathcal{G}_2$ is defined if $\mathcal{G}_1|_{\mathcal{S}_0} = \mathcal{G}_2|_{\mathcal{S}_0}$. We then call the systems composable, *and define $\mathcal{G}_1 \oplus \mathcal{G}_2 = (\mathcal{S}_1 \cup \mathcal{S}_2, \pi, N)$ by*

- $\mathcal{S}_1 \cup \mathcal{S}_2 = (TG_1 \cup TG_2, P_1 \cup P_2, \sigma_1 \cup \sigma_2)$;
- $\pi(p) =$

  $\pi_1(p)$ *if* $p \in P_1 \setminus P_2$

$\pi_2(p)$ *if* $p \in P_2 \setminus P_1$

$\pi_1(p) \cup \pi_2(p) = L_1 \cup L_2 \overset{l_1 \cup l_2}{\longleftarrow} K_1 \cup K_2 \overset{r_1 \cup r_2}{\longrightarrow} R_1 \cup R_2$ *if* $p \in P_1 \cap P_2$

- $N_p =$

  N_{1_p} *if* $p \in P_1 \setminus P_2$

  N_{2_p} *if* $p \in P_2 \setminus P_1$

  $N_{1_p}^{L_1 \cup L_2} \cup N_{2_p}^{L_1 \cup L_2}$ *if* $p \in P_1 \cap P_2$

The composition of grammars $(\mathcal{G}_1, G_0^1)$ *and* $(\mathcal{G}_2, G_0^2)$ *is defined if* $\mathcal{G}_1, \mathcal{G}_2$ *are composable and* $G_0^0 = G_0^1|_{TG_0} = G_0^2|_{TG_0}$. *In this case,* $(\mathcal{G}_1, G_0^1) \oplus (\mathcal{G}_2, G_0^2) = (\mathcal{G}_1 \oplus \mathcal{G}_2, G^0)$ *with* $G^0 = G_0^1 \cup G_0^2$, *the pushout representing the union of* G_0^1, G_0^2 *over* G_0^0.

Hence two systems are composable if they agree on the projection to $\mathcal{S}_0$, the intersection of their signatures. The same is true of transformations in these two views, i.e., if their projections to $\mathcal{S}_0$ coincide, they give rise to a composed transformation using the composed rule.

Lemma 2 (composition of steps) *Assume two systems* $\mathcal{G}_1$ *and* $\mathcal{G}_2$, *their projection* $\mathcal{G}_0 = \mathcal{G}_1|_{\mathcal{S}_0} = \mathcal{G}_2|_{\mathcal{S}_0}$ *and composition* $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$ *as above. Given steps* $t_i = (G_i \overset{p,m_i}{\Longrightarrow}_{\mathcal{G}_i} H_i)$ *for* $i = 0, 1, 2$ *with* $t_1|_{\mathcal{S}_0} = t_2|_{\mathcal{S}_0} = t_0$, *there exists a step* $t_1 \oplus t_2 = (G_1 \cup G_2 \overset{p, m_1 \cup m_2}{\Longrightarrow}_{\mathcal{G}} H_1 \cup H_2)$.

Proof 2 *Let us start by constructing the underlying DPO diagram of* $t_1 \oplus t_2$ *from those of the given transformations. First, observe that* $t_1 \leftarrow t_0 \rightarrow t_2$ *form a span of DPO diagrams with six-tuples of graph morphisms between corresponding left, right, and interface graphs at rule and transformation level relating them. Because they are obtained as projections, these morphisms form*

pullback diagrams wherever a square can be found. Forming the componentwise pushout $t_1 \to t_1 \oplus t_2 \leftarrow t_2$ *of* $t_1 \leftarrow t_0 \to t_2$*, it follows from adhesiveness that the object obtained is a double pushout and all new squares are pullbacks, too. This is a variant of the Distribution Theorem for graph transformation [16]. It remains to show that the combined DPO forms a step in* $\mathcal{G}$*, i.e., that the constraints of the combined rule are satisfied. By Def. 6 these are given by extending constraints of the two component rules. Thus, by the same argument as in the proof of Lemma 1 and the fact that they are satisfied for* t_1 *and* t_2 *it follows that they are satisfied for* $t_1 \oplus t_2$*.*

7.4 Operations on Transition Systems

With the semantics of a TAGTS defined as its induced labelled transition system, we interpret their composition by a corresponding notion of composition of LTS, synchronising transitions with shared labels while interleaving those whose labels only occur in one or the other LTS. This matches the PEPA coordination operator [26] which itself is based on composition in CSP.

Definition 7 (composition of transition systems) *Given* $LTS_1 = (S_1, L_1, \Longrightarrow_1, s_1^0)$ *and* $LTS_2 = (S_2, L_2, \Longrightarrow_2, s_2^0)$*, their product* $LTS_1 \otimes LTS_2 = (S, L, \Longrightarrow, s^0)$ *has as states* S *all pairs of states* (s_1, s_2) *with* $s_i \in S_i$*. The set of labels is defined by* $L = L_1 \cup L_2$ *and the transition relation is the smallest one satisfying*

- *if* $l \in L_1 \setminus L_2$ *and* $s_1 \stackrel{l}{\Longrightarrow}_1 s_1'$*, then* $(s_1, s_2) \stackrel{l}{\Longrightarrow} (s_1', s_2)$*;*
- *if* $l \in L_2 \setminus L_1$ *and* $s_2 \stackrel{l}{\Longrightarrow}_2 s_2'$*, then* $(s_1, s_2) \stackrel{l}{\Longrightarrow} (s_1, s_2')$*;*

- *if* $l \in L_1 \cap L_2$, $s_1 \stackrel{l}{\Longrightarrow}_1 s_1'$ *and* $s_2 \stackrel{l}{\Longrightarrow}_2 s_2'$, *then* $(s_1, s_2) \stackrel{l}{\Longrightarrow} (s_1', s_2')$.

The initial state s^0 *is* (s_1^0, s_2^0), *the pair of initial states of the two systems.*

Under suitable assumptions, composition of TAGTS is reflected by the corresponding composition of their LTS. This supports the bottom-up approach of our methodology.

Proposition 1 (composition) *Assume grammars* $(\mathcal{G}_1, G_1^0)$ *and* $(\mathcal{G}_2, G_2^0)$, *with* $\mathcal{G}_i = (\mathcal{S}_i, \pi_i, N_i)$ *and* $\mathcal{S}_0 = \mathcal{S}_1 \cap \mathcal{S}_2 = (TG_0, P_0, \sigma_0)$, *such that the grammar's composition is defined,*

1. $(\mathcal{G}_1|_{\mathcal{S}_0}, G_1^0|_{TG_0})$ *is deterministic, and*
2. *for all rules* $p_1 \in P_1 \setminus P_2$ *(resp.,* $p_2 \in P_2 \setminus P_1$*), the projection* $\pi_1(p_1)|_{TG_0}$ *(resp.,* $\pi_2(p_2)|_{TG_0}$*) is a span of isomorphisms.*

Then, transition systems $LTS(\mathcal{G}, G^0)$ *and* $LTS(\mathcal{G}_1, G_1^0) \otimes LTS(\mathcal{G}_2, G_2^0)$ *are bisimilar.*

Proof 3 *We write* $[G_1, G_2] \sim [G]$ *iff* $G_i = G|_{TG_i}$ *for* $i = 1, 2$. *States of the composed* $LTS(\mathcal{G}_1, G_1^0) \otimes LTS(\mathcal{G}_2, G_2^0)$ *are pairs of isomorphism classes* $([G_1], [G_2])$, *denoted as* $[G_1, G_2]$. *States of* $LTS(\mathcal{G}, G^0)$ *are isomorphism classes of graphs* $[G]$. *We will show that relation* $\sim$ *as defined above is a bisimulation between* $LTS(\mathcal{G}_1, G_1^0) \otimes LTS(\mathcal{G}_2, G_2^0)$ *and* $LTS(\mathcal{G}, G^0)$. *Recall that a bisimulation is a relation* R *between the states of two systems such that, if* $P_1 \; R \; P_2$, *then both* $P_1 \stackrel{l}{\longrightarrow}_1 Q_1$ *implies* $P_2 \stackrel{l}{\longrightarrow}_2 Q_2$ *and* $Q_1 \; R \; Q_2$ *as well as* $P_2 \stackrel{l}{\longrightarrow}_2 Q_2$ *implies* $P_1 \stackrel{l}{\longrightarrow}_1 Q_1$ *and* $Q_1 \; R \; Q_2$.

Assume that $[G_1, G_2] \sim [G]$ *and let* $[G] \stackrel{l}{\longrightarrow} [H]$ *be a transition in* $LTS(\mathcal{G}, G^0)$. *That means, there exists a transformation* $t = (G \stackrel{p,m}{\Longrightarrow}_{\mathcal{G}} H)$

with $obs(t) = l$. *The projections* $\mathcal{G}|_{\mathcal{S}_i}$ *are sound because of Assumption 2 in the proposition above.*

Hence, with Lemma 1, $G_i = G|_{TG_i}$ *and* $H_i = H|_{TG_i}$, *there are either two projections* $t_i = (G_i \overset{p,m|_{TG_i}}{\Longrightarrow}_{\mathcal{G}_i} H_i)$ *with* $obs(t) = obs(t_i) = l$ *for* $i = 1, 2$, *or one such projection, say WLOG for* $i = 1$, *while* $G_2 \cong H_2$. *In the first case,* $[G_i] \xrightarrow{l}_i [H_i]$ *are transitions in* $LTS(\mathcal{G}_i, G_i^0)$. *By the third clause in Def. 7,* $[G_1, G_2] \xrightarrow{l} [H_1, H_2]$ *is a transition in* $LTS(\mathcal{G}_1, G_1^0) \otimes LTS(\mathcal{G}_2, G_2^0)$. *In the second case,* $[G_1, G_2] \xrightarrow{l} [H_1, G_2] = [H_1, H_2]$ *by the first clause and* $[G_2] = [H_2]$.

Vice versa, let $[G_1, G_2] \xrightarrow{l} [H_1, H_2]$ *be given in* $LTS(\mathcal{G}_1, G_1^0) \otimes LTS(\mathcal{G}_2, G_2^0)$. *By Def. 7 either there are transitions* $[G_i] \xrightarrow{l} [H_i]$ *in* $LTS(\mathcal{G}_i, G_i^0)$ *for* $i = 1, 2$, *based on steps* $t_i = (G_i \overset{p,m_i}{\Longrightarrow}_{\mathcal{G}_i} H_i)$, *or there is one such transition, say WLOG for* $i = 1$, *and* $[G_2] = [H_2]$.

In the first case, it is easy to see that the projections from $\mathcal{S}_i$ *to* $\mathcal{S}_0$ *are sound because, e.g.,* $P_1 \setminus P_1 \cap P_2 = P_1 \setminus P_2$ *and so soundness from* $\mathcal{S}_1$ *to* $\mathcal{S}_0$ *follows from Assumption 2 above. The projections yield* $t_i|_{\mathcal{S}_0} = (G_i|_{TG_0} \overset{p,m_i|_{TG_0}}{\Longrightarrow} H_i|_{TG_0})$ *with* $obs(t_i|_{TG_0}) = l$. $[G_1, G_2] \sim [G]$ *implies* $G_1|_{TG_0} = G_2|_{TG_0} =: G_0$. *Since* $(\mathcal{G}_0, G_0^0) = (\mathcal{G}_1|_{\mathcal{S}_0}, G_1^0|_{TG_0})$ *is deterministic and* G_0 *is reachable,* $m_0 = m_1|_{TG_0} = m_2|_{TG_0}$, *so* $t_i|_{\mathcal{S}_0} = (G_0 \overset{p,m_0}{\Longrightarrow}_{\mathcal{G}_0} H_i|_{TG_0})$ *for* $i = 1, 2$ *are related by unique isomorphisms on intermediate and derived graphs.*

To apply Lemma 2 we require steps in $\mathcal{G}_1, \mathcal{G}_2$ *which project to the same step in* $\mathcal{G}_0$. *Using the unique isos between* $t_1|_{\mathcal{S}_0}$ *and* $t_2|_{\mathcal{S}_0}$ *and the fact that* $t_2|_{\mathcal{S}_0} \subseteq t_2$ *we can rename* t_2 *into* $t_2' = (G_2 \overset{p,m_2}{\Longrightarrow}_{\mathcal{G}_2} H_2')$ *such that* $t_2'|_{\mathcal{S}_0} = t_1|_{\mathcal{S}_0}$. *In particular,* $H_1|_{TG_0} = H_2'|_{TG_0} = H_1 \cap H_2'$ *and so* $t_1 \oplus t_2' = (G_1 \cup G_2 \overset{p,m_1 \cup m_2}{\Longrightarrow}_{\mathcal{G}} H_1 \cup H_2')$ *with* $obs(t_1 \oplus t_2') = l$ *is a transformation in the composed TAGTS*

delivering transition $[G] \xrightarrow{l} [H_1 \cup H_2']$. *Then,* $[H_1, H_2] \sim [H_1 \cup H_2']$ *because* $H_2 \cong H_2'$ *and therefore* $[H_1, H_2] = [H_1, H_2'] \sim [H_1 \cup H_2']$.

In the second case, by Def. 7, $l \in L_1 \setminus L_2$ *so by Def. 1* $p \in P_1 \setminus P_2$. *By Assumption 2 in the proposition above the projection* $\pi_1(p)|_{TG_0}$ *yields a span of isomorphisms which extends to a transformation* $t_0 = (G_1|_{TG_0} \overset{p,m_1|_{TG_0}}{\Longrightarrow} H_1|_{TG_0})$ *with* $G_1|_{TG_0} \cong H_1|_{TG_0}$. *Any* $G_2 \cong H_2$ *can be extended to a DPO diagram over* TG_2 *that projects to* t_0, *and following a similar argument as above we can build* $(G_1 \cup G_2 \overset{p,m_1}{\Longrightarrow}_{\mathcal{G}_1} H_1 \cup H_2)$, *extending* $(G_1 \overset{p,m_1}{\Longrightarrow}_{\mathcal{G}_1} H_1)$ *by the idle context* $G_2 \cong H_2$. *Thus* $[H_1, H_2] \sim [H_1 \cup H_2]$ *and* $[G] \xrightarrow{l} [H_1 \cup H_2]$.

The condition for the projection to $\mathcal{S}_0$ to be deterministic ensures that, on the interface, labels uniquely determine transformations. Thus transitions carrying the same labels in the two views can be synchronised. The corollary below derives the conditions for decomposition of systems by projection.

Corollary 1 (decomposition) *Assume a system* $\mathcal{G}$ *over* $\mathcal{S}$ *with sub-signatures* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *such that* $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$, *the projections* $\mathcal{G}|_{\mathcal{S}_1}$ *and* $\mathcal{G}|_{\mathcal{S}_2}$ *are sound,*

1. $\mathcal{G}|_{\mathcal{S}_1 \cap \mathcal{S}_2}$ *is deterministic;*
2. *for all rules* $p \in P_1 \setminus P_2$, $\pi(p)|_{TG_1} = \pi(p)$ *and* $N_p|_{TG_1} = N_p$ *while* $\pi(p)|_{TG_2}$ *is a span of isomorphisms, and vice versa swapping* TG_1, P_1 *and* TG_2, P_2;
3. *for all* $p \in P_1 \cap P_2$, *any constraint* $n \in N_p$ *is preserved by projection to either* TG_1 *or* TG_2.

Then $LTS(\mathcal{G}, G)$ *and* $LTS(\mathcal{G}|_{\mathcal{S}_1}, G|_{TG_1}) \otimes LTS(\mathcal{G}|_{\mathcal{S}_2}, G|_{TG_2})$ *are bisimilar.*

Proof 4 *To apply Prop. 1 it suffices to show that* $\mathcal{G}|_{\mathcal{S}_1} \oplus \mathcal{G}|_{\mathcal{S}_2} = \mathcal{G}$ *and Assumptions 1, 2 are satisfied. Assumption 1 follows directly from Assumptions 1 above. Assumption 2 in Prop. 1 follows from Def. 4. Finally,* $\mathcal{G}|_{\mathcal{S}_1} \oplus \mathcal{G}|_{\mathcal{S}_2} = \mathcal{G}$ *because, due to Assumptions 2 and 3 above, rule spans and application conditions in* $\mathcal{G}$ *can be reconstructed from their projections.*

Note that, with $TG = TG_1 \cap TG_2$*, projection of graphs and rules is inverse to their union. This can be shown by formalising the union of type graphs as a van Kampen square, which extends to a cube with the union of instance graphs, such as the left-hand sides of corresponding rules, and their typing morphisms.*

Let us analyse the conditions of Cor. 1 by considering the three rules in Fig. 7.2 and 7.3. Specifically, condition 2 distinguishes rules like *sendAccidentInfo* and *moveCar*, both retained entirely in one view and mapped to spans of isos in the other, from rules like *detour*, also mapped to isos in the Service views, but loosing an edge of type *knows* in the *Car* view. While the reconstruction of the first two examples is trivial, with the rules being complete in one of the two views each, in the third case we require synchronisation to recover the projected edge. Hence in this case the rule must be retained in both views. Similarly, for a rule present in both views, each of its constraints must be preserved in at least one view. In the case of *detour*, all constraints remain in the *Car* view.

The key ingredient is the fact that labels determine matches and transformations up to isomorphism. Therefore, n steps in different views with the same label starting from compatible graphs are consistent as distributed steps

and can therefore be amalgamated. Not surprisingly, therefore, the results of analyzing the model obtained by synchronizing the n projections coincide with those of analyzing the global model.

If we assume PEPA processes `P1, P2, ..., Pn` and let $LTS(\texttt{P})$ be the labelled transition system generated by a PEPA process (ignoring rates for the time being), the product of transition systems is the semantic equivalent of PEPA's co-operation operator, i.e., the reachable portions of $LTS(\texttt{P1} \bowtie \texttt{P2} \bowtie \ldots \bowtie \texttt{Pn})$ and $LTS(\texttt{P1}) \otimes LTS(\texttt{P2}) \otimes ... \otimes LTS(\texttt{Pn})$ are isomorphic. As a consequence, the two systems are bisimilar. Similarly in PRISM we suppose modules `M1, M2, ..., Mn` and $LTS(\texttt{M})$ be the labelled transition system generated by PRISM, then the synchronization of PRISM modules over shared labels and the product of transition systems $LTS(\texttt{M1 || M2 || ... || Mn})$ and $LTS(\texttt{M1}) \otimes LTS(\texttt{M2}) \otimes ... \otimes LTS(\texttt{Mn})$ are isomorphic. Moreover, a shared label will have the same rate in sequential PEPA processes or in individual PRISM modules and in the global PEPA process or in the global PRISM module, and synchronization of transitions in PEPA or PRISM retains that rate. Therefore, the relationship between composition and product carries over to transition systems with rates (or continuous-time Markov chains).

7.5 Summary

We have formalised notions of views for typed attributed graph transformation systems with rule signatures for decomposition and composition of systems that are compatible with corresponding notions of composition of

transition systems based on synchronisation over shared labels. Conceptually, decomposition and composition correspond to the top-down and the bottom-up methodologies of modular graph transformation systems. We show that LTS generated by the synchronization of views is bisimilar to the LTS of global model.

Chapter 8

Generating CTMCs

In this chapter we discuss how we generate CTMCs from LTSs. Generating CTMCs from LTSs requires additional information not present in the LTSs such as event arrival rates. These CTMCs are further translated into PEPA or PRISM models.

8.1 From Transition Systems to Markov Chains

We define Labelled Transition Systems (LTS) and Continuous Time Markov Chains (CTMCs). We discuss some of the basic properties of the $Q-matrix$ and explain the connection with CTMCs [35].

A labelled transition system describes the overall behaviour of a system in terms of its states and transitions. In a graph transition system states represent isomorphism classes of graphs and transitions represent rule applications as it is shown in the Fig. 8.1. Transition systems are frequently

used to represent the behavior of software systems. They divide the runtime evolution of a system into discrete *states* and use a binary *transition relation* to define possible state changes. State space S contains all reachable graphs of the graph transformation system. Such a transition system can be generated by recursively applying all enabled graph transformation rules of $\mathcal{G}$ at each state and by matching the resulting graphs with already generated isomorphic graphs.

Definition 8 (Q-matrix) *Let S be a countable set. A Q-matrix on S is a real-valued matrix $Q = Q(s, s')_{s,s' \in S}$ satisfying the following conditions:*

(i) $0 \leq -Q(s, s) < \infty$ *for all* $s \in S$,

(ii) $Q(s, s') \geq 0$ *for all* $s \neq s'$,

(iii) $\sum_{s' \in S} Q(s, s') = 0$ *for all* $s \in S$.

The Q-matrix is also called *transition rate matrix* of the Markov chain. A CTMC makes transitions from state to state independent of the past for an exponentially distributed amount of time. It has the memoryless property.

Definition 9 (CTMC) *A (homogeneous) Continuous Time Markov Chain is a pair $\langle S, Q \rangle$ where S is a countable set of states and Q is a Q-matrix on S.*

$$Q(s) = \sum_{s \neq s'} Q(s, s') < \infty$$

If $s \neq s'$ and $Q(s, s') > 0$, then there is a transition from s to s'. The transition delay is exponentially distributed with rate $Q(s, s')$. Consequently,

the probability that, being in s, the transition $s \rightarrow s'$ can be triggered within a time interval of length t is $1 - e^{-Q(s,s')t}$. The *total exit rate* $Q(s)$ specifies the rate of leaving a state s to any other state and the diagonal entry $-Q(s)$ makes the total row sum zero. If the set $\{s' \mid Q(s, s') > 0\}$ is not a singleton, then there is a competition between the transitions originating in s. The probability that transition $s \rightarrow s'$ wins the 'race' is [1].

$$\frac{Q(s, s')}{Q(s)}$$

We say that a state s' *can be reached* from s, and write $s \rightharpoonup s'$, if there are states $s = s_0, \ldots, s_n = s'$, such that $Q(s_0, s_1) \cdot Q(s_1, s_2) \cdot \ldots \cdot Q(s_{n-1}, s_n) > 0$. If $s \rightharpoonup s'$ and $s' \rightharpoonup s$, then we say that s and s' *communicate*, and write $s \rightleftharpoons s'$.

Definition 10 (irreducible Q-matrix) *A Q-matrix Q is irreducible if $s \rightleftharpoons s'$ for all $s, s' \in S$.*

Fig. 8.1 shows the LTS of the TIS inteface and Fig. 8.2 shows its $Q - matrix$. We have given *accident* and *removeAccident* 1 and 8 rates respectively. When we are modeling a system as a GTS, we have to take care that the resulting Markov chain should be irreducible. Meaning that every state in the LTS is reachable from every other one. In our case study, when a car reaches its destination, it is replaced at its starting point. Thus the behaviour is cyclic and so the Markov chain is irreducible.

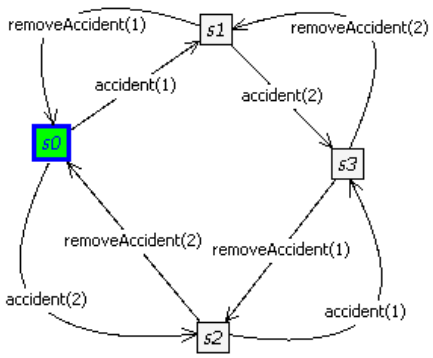


Figure 8.1: The LTS of the TIS inteface

$$\begin{pmatrix} -2 & 1 & 1 & 0 \\ \frac{1}{8} & -\frac{9}{8} & 0 & 1 \\ \frac{1}{8} & 0 & -\frac{9}{8} & 1 \\ 0 & \frac{1}{8} & \frac{1}{8} & -\frac{2}{8} \end{pmatrix}$$

Figure 8.2: Q-matrix of the TIS inteface

8.2 Choosing Transition Rates

Rates are based on observations of real world behaviour and their timings. Delays can be measured in the real world, that the average delay for a certain kind of event can be computed, and how the rate relates to the average delay. In the CTMCs reciprocal of a rate is the average delay. Table 8.1 shows rule labels and their associated rates. We use a unit, on average, of *times per day*, e.g., $accident = 1$, $removeAccident = 8$ and $arriveAtDest = 17280$ meaning that Accidents can occur once a day and are removed within 3 hours and a car is replaced at its starting point within 5 sec, this does not mean a car arrives after 5 secs, but that once the car has arrived, it takes 5 secs on average for it to be replaced.

The exponential distribution is the only continuous memoryless random

Table 8.1: Rule labels and their rates

Rule Labels	Rates	Real World Values
accident	1	1 day
arriveAtDest	17280	5 sec
assign	1440	1 min
detour	17280	5 sec
getAccidentInfo	1440	1 min
moveCar	288	5 min
moveTowTruck	144	10 min
moveBackTowTruck	144	1 min
rejoin	17280	5 sec
removeAccident	8	3 hours
sendAccidentInfo	72	20 min

distribution. If we know how many times in a day, a car takes a detour, or how often accidents are occuring then these events can be considered as following an exponential distribution. The exponential distribution is always the choice to represent the time between events that happen at a constant average rate such as *getAccidentInfo* time and rate of *arriveAtDest*. Exponential distributions tie together mean and variance, respectively $\frac{1}{\lambda}$ and $\frac{1}{\lambda^2}$, where λ is the exponential rate.

Stochastic analysis heavily depends on the rates. If the timing of observations are very close to the real world scenarios, the results of a stochastic analysis can be realistic too. If rates are inaccurate, so will be the stochastic analysis.

8.3 CTMCs Generated from LTSs

We transform LTSs annotated with rates into CTMCs and further translate them into either PEPA or PRISM. We have translated the TIS Interface LTS of Fig 8.1 into CTMC and then into PEPA. Process variables such as $P0$ correspond to a state $s0$ in the labelled transition system. An equation like *P0 = ("accident(2)",accident).P1* means that there exists a transition with rate $accident = 1$ from $P0$ to $P1$, labelled *accident(2)*.

```
// PEPA model
accident = 1; //parameter
removeAccident = 8; //parameter
P0 =
    ("accident(2)",accident).P1+
    ("accident(1)",accident).P2;
P1 =
    ("accident(1)",accident).P3+
    ("removeAccident(2)",removeAccident).P0;
P3 =
    ("removeAccident(1)",removeAccident).P1+
    ("removeAccident(2)",removeAccident).P2;
P2 =
    ("accident(2)",accident).P3+
    ("removeAccident(1)",removeAccident).P0;
P0
// End PEPA model
```

8.4 Summary

In this chapter we have defined Continuous Time Markov Chains (CTMCs) and have shown how we generate a CTMC from the LTS of a GROOVE model and rates.

Chapter 9

Tool Support

In this chapter we will discuss tool support to transform LTSs generated from graph transformation systems in GROOVE into CTMCs. These CTMCs are further translated into PEPA or PRISM models. Tool support for automated generation of local views from a global graph transformation system is also given.

9.1 Transforming GROOVE Models into CTMCs

CTMCs are presented in different ways by different tools. We consider PEPA and PRISM as targets.

9.1.1 PEPA

Our tool generates PEPA sequential components from graph transformation systems in GROOVE. These individual components are synchronised in PEPA by the cooperation operator.

In PEPA we can analyze non-functional properties of a model, such as the steady-state of a system, which gives the long term probability of each state, and the throughput, which gives the long term frequency of its actions. Passage time analysis can be carried out to explore within which time frame a request will be satisfied.

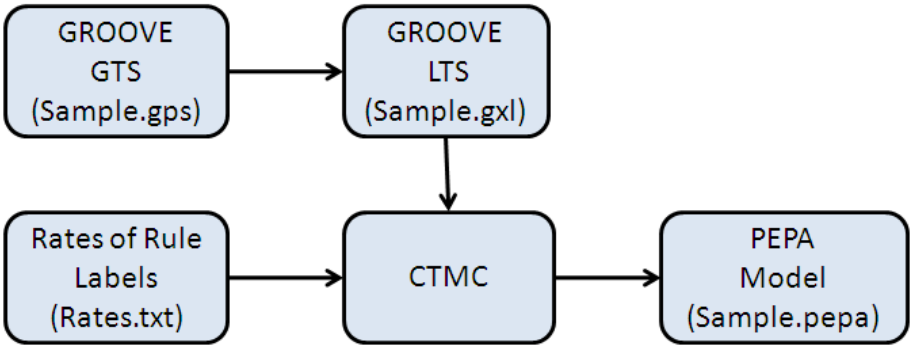


Figure 9.1: The architecture of PEPA model generation

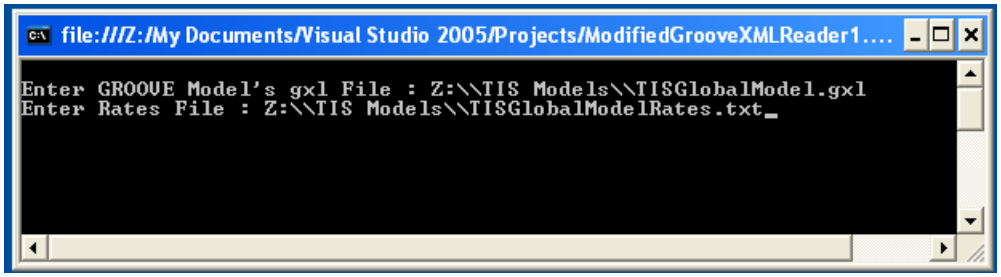


Figure 9.2: Input files needed for a CTMC generation

We generate an LTS from a graph transformation system in GROOVE.

We then transform this LTS into a CTMC by assigning each rule label a rate through our CTMC Generator tool. The CTMC is further translated into a PEPA model. Fig. 9.2 shows the interface of the CTMC Generator tool. The first file is the LTS of the TIS global model and the second is its rates file. The output file is TISGlobalModel.pepa. Its input files are shown below.

```
******* An extract of TISGlobalModel.gxl ******
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
<graph edgemode="directed" edgeids="false" role="graph" id="TIS-
GlobalModel">
        <attr name="$version$">
            <string>curly</string>
        </attr>
        <node id="n2396"/>
        <node id="n1471"/>
        <node id="n1174"/>
        <node id="n2091"/>
...

        <edge to="n317" from="n213">
            <attr name="label">
                <string>accident(1)</string>
            </attr>
        </edge>
```

```
        <edge to="n1500" from="n1316">
            <attr name="label">
                <string>getAccidentInfo(1)</string>
            </attr>
        </edge>
        <edge to="n392" from="n630">
            <attr name="label">
                <string>assign(1)</string>
            </attr>
        </edge>
           ...

    </graph>
</gxl>

******* TISGlobalModelRates.txt ******
accident = 1
arriveAtDest = 17280
assign = 1440
detour = 17280
getAccidentInfo = 1440
moveCar = 288
moveTowTruck = 144
moveBackTowTruck = 144
rejoin = 17280
```

```
removeAccident = 8
sendAccidentInfo = 72
```

Above is the algorithm for generating a PEPA model from an LTS of GROOVE and rates. In the algorithm, in lines 3 to 6 rule labels/actions are assigned rates. At lines 8 to 25 each process is assigned certain activities. When these activities are performed a process transforms to the resulting processes. An extract of the PEPA model (TISGlobalModel.pepa) is given below. Process variables such as $N563261$ correspond to states in the labelled transition system $LTS(\mathcal{G}, G_0)$. An equation like *N563261 = ("removeAccident(1)", removeAccident).N563269* means that there exists a transition with rate *removeAccident* $= 8$ from $N563261$ to $N563269$, labelled *removeAccident(1)*.

```
// Rates
accident = 1;           arriveAtDest = 17280;
detour = 17280;         getAccidentInfo = 1440;
moveCar = 288; rejoin = 17280;
removeAccident = 8;  sendAccidentInfo = 72;
...
//Processes
N563261 =
("removeAccident(1)",  removeAccident).N563269+
("removeAccident(2)",  removeAccident).N563396+
("getAccidentInfo(1)", getAccidentInfo).N563357;
```

Algorithm 1 Generating a PEPA model from an LTS and rates

```
1: Input: states[], transitions[][], rules[]with rates
2: //Parameters
3: for (a=0 to number of rules) do
4:     rate = getRate(rules[a])
5:     print rules[a] = rate
6: end for
7: //Processes
8: for (i=0 to number of states) do
9:     print states[i]
10:    firstIteration = true
11:    for (j=0 to number of transitions) do
12:        transSourceState = transitions[j][0]
13:        transition = transitions[j][1]
14:        transTargetState = transitions[j][2]
15:        if ((states[i]==transSourceState) & isValidTransition(transition)) then
16:            if (firstIteration) then
17:                firstIteration = false
18:            else
19:                print +
20:            end if
21:            rate = getRate(transition)
22:            print transition, rate and transTargetState
23:        end if
24:    end for
25: end for
```

```
N563262 =

("removeAccident(1)",  removeAccident).N563416+

("removeAccident(2)",  removeAccident).N563424;

...
```

In the PEPA Editor we open and edit PEPA model files. PEPA model files have the .pepa extension in the workbench. Once a model is loaded, it is automatically parsed and we can derive its state space. Fig. 9.3 shows a

tabular representation of the derived state space in the State Space View with steady-state probability distribution of the TISSynchronized model. The first column shows the state number and 2nd, 3rd and 4th columns are showing state space of synchronised components (views) Recovery View, Car View and Service View respectively, and 5th column is showing the steady-state probability distribution. In addition, throughput and utilisation are carried out automatically and results are in the Performance Evaluation View.

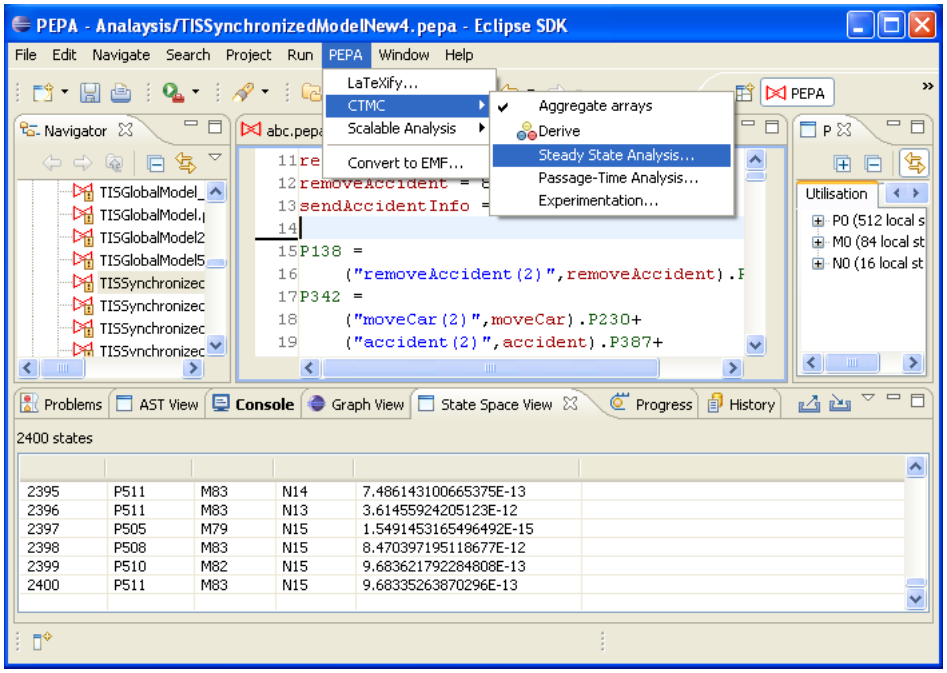


Figure 9.3: Steady-state of TIS synchronized model

The Performance Evaluation View shows information about throughput and utilisation. Throughput gives the actual long-term frequencies at which transitions are executed and utilization gives the long term probability distribution of individual states of components. Fig. 9.4 shows throughput of TISSynchronized model.

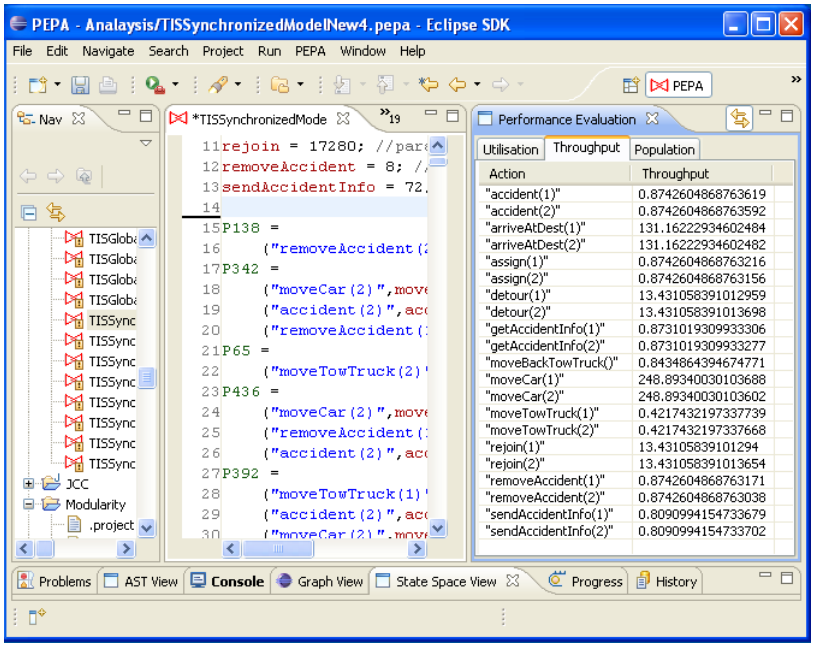


Figure 9.4: Throughput analysis of TIS synchronized model

9.1.2 PRISM

The PRISM [31, 32] tool is a probabilistic model checker, which is used for modelling and analysis of systems which exhibit probabilistic behaviour. The PRISM is a state-based language. We construct PRISM models by transforming GROOVE LTSs into CTMCs and then further translating them into PRISM models. Below is the algorithm for generating a PRISM model from a GROOVE LTS and rates. In the algorithm in lines 3 to 6 constants are assigned rates. In line 19 each state is assigned a transition and in square bracket there is a transition/action label.

We have transformed GROOVE models into PRISM, because PEPA does not support larger models. The TIS global model with two cars, having 2400 states and 7904 transitions, could not be loaded by PEPA, but we can con-

Algorithm 2 Generating a PRISM model from an LTS and rates

```
1: Input: states[], transitions[][], rules[]with rates
2: //Constants
3: for (a=0 to number of rules) do
4:     rate = getRate(rules[a])
5:     print rules[a] = rate
6: end for
7: //Module
8: initialNode = getInitialNode(transitions)
9: print module moduleName
10: state = 'anyVariale'
11: print state: [0.. number of states -1] init initialNode
12: for (i=0 to number of states) do
13:     for (j=0 to number of transitions) do
14:         transSourceState = transitions[j][0]
15:         transition = transitions[j][1]
16:         transTargetState = transitions[j][2]
17:         if ((states[i]==transSourceState) & isValidTransition(transition)) then
18:             rate = getRate(transition)
19:             print [transition] (state = transSourceState) -⟩ rate * 1 : (state' = transTargetState)
20:         end if
21:     end for
22: end for
```

struct the same model from its local views namely Car View, Service View and Recovery View.

Fig. 9.2 shows the interface of the CTMC Generator which takes two files as input and produces the CTMC of a model. The CTMC is further translated into two PRISM files. One is a system model (TISGlobalModel.sm) and the other is continuous stochastic logic (CSL) file (TISGlobalModel.csl) which is a set of formulas and sets of states. The CSL file is used to express and verify properties like the probability of arriving at destinations and the time frame

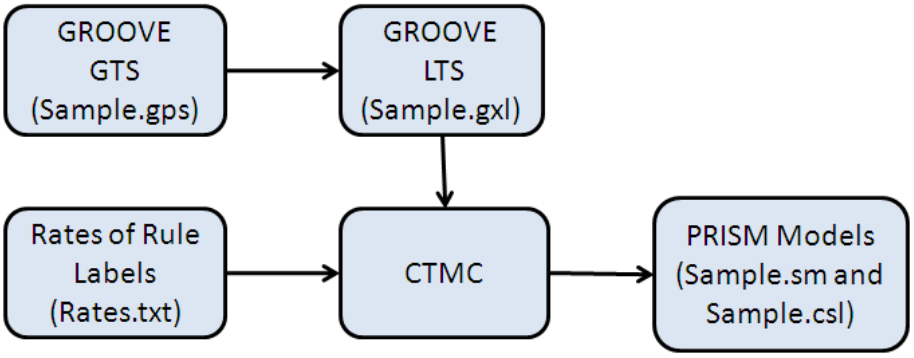


Figure 9.5: The architecture of PRISM model generation.

within which an accident will be removed, etc. When defining properties of a model, one of the most fundamental tasks is to identify particular sets of states of the model. For example, to verify a property such as "the car arrives at its destination with probability 1", it is first necessary to identify the states of the model which correspond to a situation where "the car arrives at its destination". Action labels identify particular sets of states of the system. Followings are the PRISM files of the TIS model.

```
******* An extract of TISGlobalModel.sm ******
stochastic


const double accident = 1;
const double arriveAtDest = 17280;
const double assign = 1440;
const double detour = 17280;
...
```

```
module K
s :  [0..215] init 0;
[accident_1] (s=0) -> accident * 1 :(s'=3);
[accident_3] (s=0) -> accident * 1 :(s'=1);
[moveCar_2] (s=0) -> moveCar * 1 :(s'=5);
[moveCar_1] (s=0) -> moveCar * 1 :(s'=6);
[moveCar_3] (s=0) -> moveCar * 1 :(s'=4);
[accident_2] (s=0) -> accident * 1 :(s'=2);
[accident_2] (s=1) -> accident * 1 :(s'=7);
[moveCar_2] (s=1) -> moveCar * 1 :(s'=9);
[accident_1] (s=1) -> accident * 1 :(s'=8);
[removeAccident_3] (s=1) -> removeAccident * 1 :(s'=0);
[moveCar_1] (s=1) -> moveCar * 1 :(s'=10);
[removeAccident_2] (s=2) -> removeAccident * 1 :(s'=0);
  ...

  endmodule
```

The statement in the above PRISM model "[accident_1] (s=0) $->$ accident * 1 :(s'=3)" shows that there is a transition from $(s = 0)$ to $(s' = 3)$ with rate 1 which is the value of accident rule label. In square bracket [accident_1] it shows the action label for the transition.

```
******* An extract of TISGlobalModel.csl *******
```

```
const double k;


label "accident_1" = (s= 170| s= 433| s= 186| s= 199| s= 105| ...);
label "accident_1" = (s= 212| s= 468| s= 238| s= 252| s= 147| ...);
label "moveCar_1" = (s= 495| s= 228| s= 474| s= 339| s= 91| ...);
label "moveCar_1" = (s= 420| s= 286| s= 394| s= 229| s= 120| ...);
label "removeAccident_1" = (s= 302| s= 67| s= 345| s= 107| ...);
label "removeAccident_1" = (s= 357| s= 102| s= 390| s= 144| ...);
...
P=? [ F<=k "removeAccident_1" ]
P=? [ F<=k "arriveAtDest_1" ]
```

The statement in the above CSL file label "accident_1" = (s= 170| s= 433| s= 186|s= 199| s= 105|...) identifies a set of states of the model which correspond to situation where "accident_1" is an action label of a transition. The statement P=? [F<= k "arriveAtDest_1"] is a formula for analysing a transient property of a system. It asks what is the probability of a car arriving at its destination, where k will have a range of values. We change k by small steps from 0 to a sufficiently large number, here in our case k represents a unit of *times per day*.

Fig. 9.6 shows a snapshot of a PRISM system file (TISGlobalModel.sm). When we compile it we get 2400 *states* and 7904 *transitions*. Fig. 9.7 shows the PRISM CSL file (TISGlobalModel.csl) which depicts a transient properity of arriveAtDest rule.

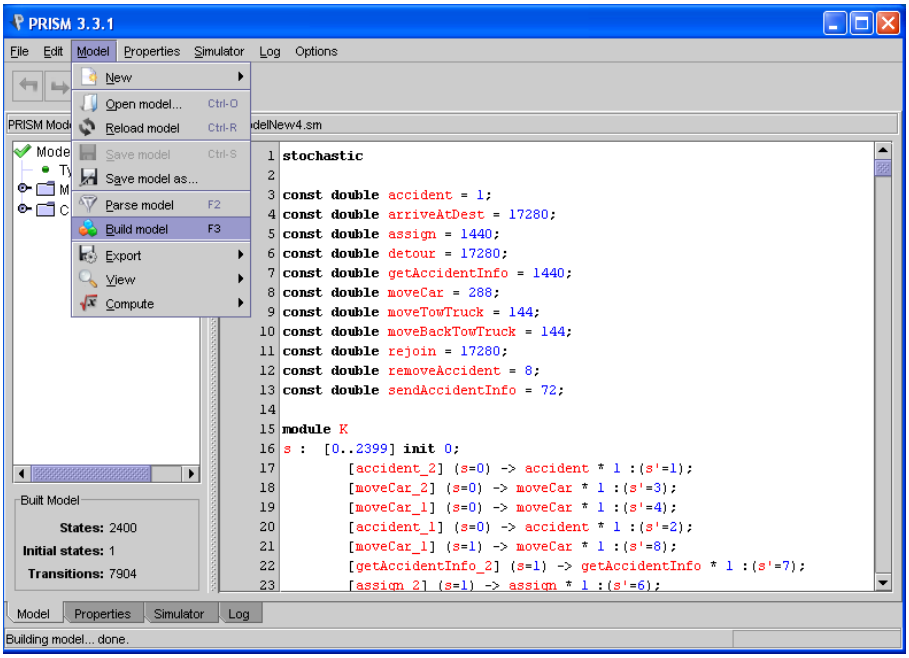


Figure 9.6: PRISM system file (TISGlobalModel.sm) is loaded in editor

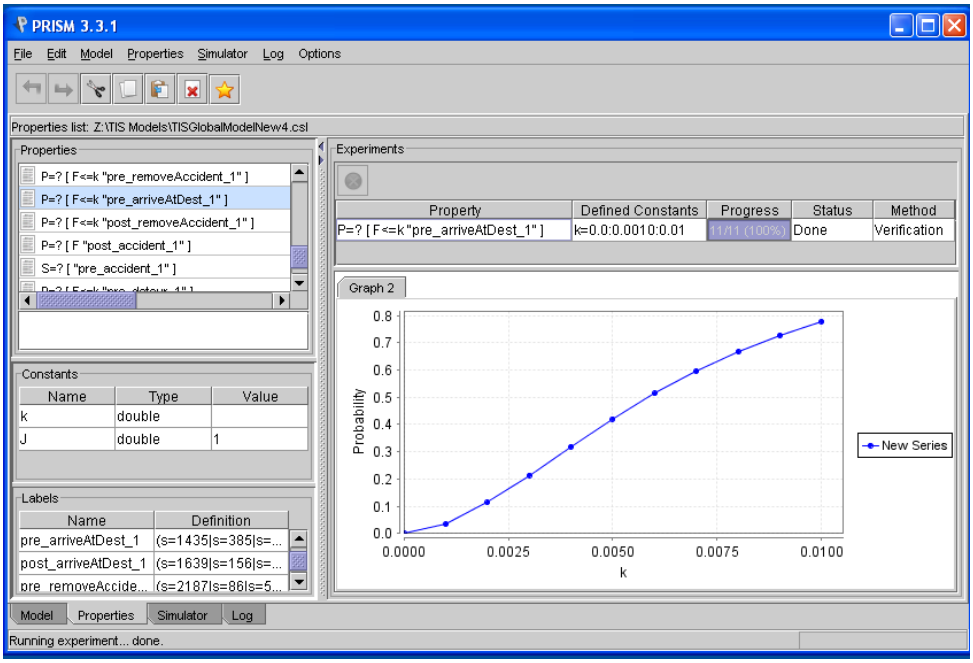


Figure 9.7: PRISM CSL file (TISGlobalModel.csl) is loaded in properties view

9.2 A Tool for Generation of Local Views

The View Generator tool requires a global graph transformation system of GROOVE and the type graph of a view. It performs the projection of global graphs to the view's local type graph. It reduces start graphs and rules to the remaining types, removing all instances of types that are no longer present in the views' type graphs. Following is the algorithm of View Generator tool. In lines 2 to 4 the loop calls *modularize* function for each instance of the global model that includes all rules and start graphs. The function parameters are an instance graph of a global model and a local type graph of a view. The *modularize* function given at line 5 identifies nodes and edges that should be subtracted from the global model, then it deletes all nodes and edges whose types could not survive in the projection.

Algorithm 3 View Generator Algorithm

```
1:  Input: globalInstanceGraphs[] includes rules and start graphs, viewTypeGraph
2:  for (a=0 to number of globalInstanceGraphs) do
3:      modularize(globalInstanceGraphs[a],viewTypeGraph)
4:  end for
5:  function MODULARIZE(instanceGraph,viewTypeGraph)
6:      instanceGraphNodes = getNodes(instanceGraph)
7:      instanceGraphEdges = getEdges(instanceGraph)
8:      viewTypeGraphEdges = getEdges(viewTypeGraph)
9:      for (i=0 to number of instanceGraphEdges) do
10:         isViewEdge = false
```

Algorithm 4 View Generator Algorithm

```
11:     isViewNode = false
12:     edgeSourceNode = instanceGraphEdges[j][0]
13:     instanceEdge = instanceGraphEdges[j][1]
14:     edgeTargetNode = instanceGraphEdges[j][2]
15:     if (instanceEdge == nodeType) then
16:         for (j=0 to number of viewTypeGraphEdges) do
17:             viewEdge = viewTypeGraphEdges[j][1]
18:             if (instanceEdge == viewEdge) then
19:                 isViewNode = true
20:                 break
21:             end if
22:         end for
23:         if (!isViewNode) then
24:             deleteNodes.add(edgeSourceNode)
25:         end if
26:     else if (instanceEdge == dataType) then
27:         AttributeNodes.add(edgeSourceNode)
28:     else if (instanceEdge == NACNode) then
29:         NACNodes.add(edgeSourceNode)
30:     else if (isValidEdge(instanceEdge)) then
31:         for (j=0 to number of viewTypeGraphEdges) do
32:             viewEdge = viewTypeGraphEdges[j][1]
33:             if (instanceEdge == viewEdge) then
34:                 isViewEdge = true
```

Algorithm 5 View Generator Algorithm

```
35:                     break
36:                 end if
37:             end for
38:             if (!isViewEdge) then
39:                 deleteEdges.add(instanceEdge)
40:             end if
41:         end if
42:     end for
43:     deleteNodes(deleteNodes, AttributeNodes, NACNodes)
44:     deleteEdges(deleteEdges)
45: end function
```

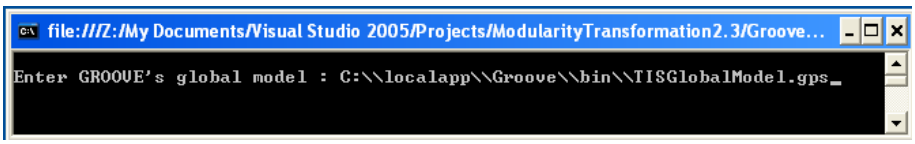
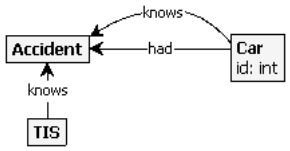
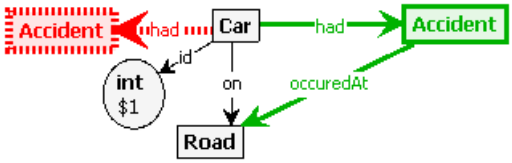


Figure 9.8: The View Generator tool takes a global model as an input.

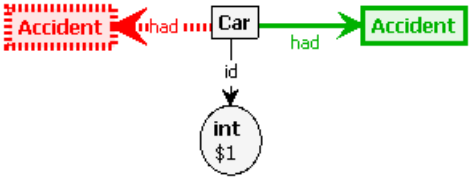
Fig. 9.8 shows the interface of the View Generator tool. It takes a global GTS as input. In global GTS we reduce the global type graph to the view's local type graph, so it is used as another input for the tool. View Generator projects global GTS to the local view based on the view's type graph. In our case it reduce the TIS global model to Car view, Service view and Recovery view as discussed in Chapter 6. It does this by transforming GROOVE's XML representation of rules and start graphs by deleting all nodes and edges whose types do not survive in the projection. It also takes care of dangling edges



(a) Type graph of Service View



(b) *accident(car) rule of the TIS model*



(c) Service View rule accident(car) after projection

Figure 9.9: Service view projections of *accident*(*car*)

and dangling nodes, by deleting all incident edges of deleted nodes and by deleting all NACs and attribute nodes of the deleted nodes.

Fig. 9.9(c) shows the projection of the TIS model's rule *accident(car)* given in Fig. 9.9(b) to Service View model. As the Service View type graph given in Fig. 9.9(a) does not have road node type, so the View Generator tool deletes road node instances and all thier incident edges.

9.3 Summary

In this chapter we have discussed our tool support for CTMC generation and the tool that automatically generates views from a global graph transformation system based on their local type graphs. The CTMC Generator transforms GROOVE LTSs into CTMCs. These CTMCs are further translated into PEPA or PRISM models for synchronization and stochastic analysis. The View Generator transforms the global graph transformation system given as a GROOVE model by projecting GROOVE's XML representation of rules and graphs into projected rules and graphs of the view based on its type graph.

Chapter 10

Evaluation

In this chapter we discuss limitations and achievements of our approach, advantages and shortcomings of modelling through graph transformation and performance analysis. We also suggest that views should be as small as possible, because this will reduce the size of the state space.

10.1 Global vs Synchronized Model

The results of analysing the model obtained by synchronising the projections coincide with those of analysing the global model. What is more interesting is the reduction in the state spaces produced. To witness, Table 10.1 shows the size of the global and local views as generated by GROOVE and the synchronized models composed in PRISM and PEPA. The synchronized model is composed in PEPA for two cars, while in PRISM for two and three Cars. GROOVE is out of memory for three cars in the case of the global model and in the case of local views it is out of memory for five cars.

Table 10.1: Global vs synchronized model

Cars	GROOVE								PRISM		PEPA	
	Global model		Car view		Service view		Recovery view		Synchronized model		Synchronized model	
	states	trans.	states	trans.	states	trans.	states	trans.	states	trans.	states	trans.
2	2400	7904	84	248	16	48	512	1524	2400	7904	2400	7904
3	out of memory		936	4062	216	1188	6600	25434	401436	1956078	out of memory	
4			9984	56652	10000	92000	79793	373162	out of memory			
5			out of memory									

Table 10.2: Global model vs local views state and transition ratio

Cars	Global model		Car view		Service view		Recovery view		States Ratio	States + Trans. Ratio
	states	trans.	states	trans.	states	trans.	states	trans.	(CV+SV+RV)/G	(CV+SV+RV)/G
2	2400	7904	84	248	16	48	512	1524	0.255	0.236
3	401436	1956078	936	4062	216	1188	6600	25434	0.01931	0.01630

It turns out that, in this case, we are unable to generate the LTS from the global system directly, but through modularity and by synchronization of local views over shared labels, we can compose a global system with 401436 states and 1956078 transitions.

10.2 Decomposition of Views

As we are decomposing a system on the basis of local type graphs, while decomposing a system into views we should take care that they have as small type graphs as possible. Projections over smaller type graphs result in smaller subsets of rules. Therefore, the generated state spaces would also be smaller. If some views are very small and some are very large, then their state spaces are unbalanced, resulting in a large overall state space. Therefore, we suggest that there should be some balance between views. In Table 10.2 we give the ratio of global models compared to local views. As for three Cars we can compose a global model from its local views, we also give it in the table. Table 10.2 shows that there is a four times reduction in the state space and number of transitions for two Cars and more than a 98% reduction in the state space and number of transitions for three Cars.

10.3 Performance Analysis

Based on the PEPA process we can extract performance measures such as the *steady-state solution* providing long-term probabilities for all states and the *transition throughput* giving the actual long-term frequencies at which

transitions are executed. Table 10.3 shows rule labels and their rates. We use a unit of *times per day* , e.g., accident = 1 and removeAccident = 8 mean that an Accident can occur once a day and it is removed in 3 hours respectively. Fig 10.4 shows steady-state solution for the *TIS Interface* given in Fig 8.1. It shows that for 79% of the time, the system will be in state *P0*, for nearly 20% in states *P1* or *P2* and for nearly 1% in state *P3*. That means, for most of the time there is no accident, while the probability that one car is involved in an accident at any time is 20% and the probability that two cars are involved at the same time is 1%.

Another interesting property to be analysed in our model is the throughput of rules, showing the long-term frequency of actions. Fig. 10.5 shows the througphput of our model. The throughput of rule *arriveAtDest(car)* shows the long-term frequency at which Cars finish their journey. They are replaced by Cars at the start of their trip in order to yield a non-reducible (i.e., strongly connected) CTMC. It shows that a Car finishes its journey in about 11 minutes. The throughput is therefore a measure of the long-term average time between these arrival transitions. The results reported in Table 10.6 also show the throughput of the *arriveAtDest* and *detour* rules. When we increase the rate of the *removeAccident* rule from 8 to 24, then a Car's arrival at destination is sped up by 6.1% and its detour possibility is reduced by 53.8%. This gives an indication of the potential pay-off in investing in more tow trucks to remove accidents.

Table 10.3: Rule labels and their rates

Rule Labels	Rates
accident	1
arriveAtDest	17280
assign	1440
detour	17280
getAccidentInfo	1440
moveCar	288
moveTowTruck	144
moveBackTowTruck	144
rejoin	17280
removeAccident	8
sendAccidentInfo	72

Table 10.4: Steady-state of the TIS Interface

States	Steady-state
P0	0.790123457
P1	0.098765432
P2	0.098765432
P3	0.012345679

Table 10.5: Rule labels and their throughput

Action type	Throughput
accident(1)	0.874260487
accident(2)	0.874260487
arriveAtDest(1)	131.1622293
arriveAtDest(2)	131.1622293
assign(1)	0.874260487
assign(2)	0.874260487
detour(1)	13.43105839
detour(2)	13.43105839
getAccidentInfo(1)	0.873101931
getAccidentInfo(2)	0.873101931
moveBackTowTruck()	0.843486439
moveCar(1)	248.8934003
moveCar(2)	248.8934003
moveTowTruck(1)	0.42174322
moveTowTruck(2)	0.42174322
rejoin(1)	13.43105839
rejoin(2)	13.43105839
removeAccident(1)	0.874260487
removeAccident(2)	0.874260487
sendAccidentInfo(1)	0.809099415
sendAccidentInfo(2)	0.809099415

Table 10.6: The performance of *arriveAtDest* and *detour* rules are shown, when the rate of *removeAccident* is increased from 8 to 24

Action type	removeAccident (Rate = 8)	removeAccident (Rate = 24)	Percentage
arriveAtDest	131.1622293	139.2083196	6.134457
detour	13.43105839	6.203623247	-53.8114

10.4 Summary

In this chapter we have discussed the limitations of GROOVE, PEPA and PRISM tools and have given the comparative analysis of local views and global models in terms of states and transitions. We have shown how modularity reduces the size of the state space, mitigating the scalability problem. We have also given performance analysis of our models.

Chapter 11

Conclusion

This chapter presents a summary of our contributions and their evaluation in terms of our thesis statement. We also discuss possible future work.

11.1 Summary of Contributions

We have used graph transformation as a language for developing abstract models of systems of dynamic nature. Using the Traffic Information System (TIS) as our case study, we have modelled it by incorporating physical mobility and the interaction and communication with other components of the business domain, introducing modularity to avoid state space explosion. In addition, we have given tool support for performance analysis, where LTSs generated from graph transformation systems are transformed into CTMCs by our CTMC Generator tool. These CTMCs are further translated into PEPA or PRISM to analyze performance properties. Tool support for automated generation of local views from a global graph transformation system

is also given.

We have distinguished three approches monothic, top-down and bottom-up, to system modelling. In the monolithic approach, the system is modelled globally. Graph transformation does not allow to model a system as composition of subsystems, which is a major bottleneck for modelling large systems. In the top-down approach a global graph transformation system is decomposed into its subsystems (views) based on their local type graphs by our View Generator tool, while in the bottom-up approch the system is modelled as a composition of subsystems having a shared interface. We knit these subsystems (views) together in either PEPA or PRISM and synchronize them over shared labels. Through synchronization of subsystems we get a resulting system which is bisimilar to the original global system. We have also given the formalization of the composition and decomposition of systems.

11.2 Evaluation

In our thesis statement, we identified challenges of modelling mobility, state space explosion, and performance analysis. We have given the solution of these problems in our thesis and review them here briefly.

Modelling Mobility We have used graph transformation as a language for developing abstract models of systems of dynamic nature. We have modelled the TIS, which incorporates physical mobility, interaction and communication with other components of a business domain, through graph transformation.

State Space Explosion We have introduced top-down and bottom-up modularity to mitigate state space explosion, which usually occurs while modelling complex/large systems. In the top-down approach, our View Generator tool decomposes a global model into subsystems (views). In the bottom-up approach, we structure our models into different views sharing a common interface.

Synchronization over Shared Labels Subsystems/views are synchronized over shared labels in PEPA and PRISM in order to compose a global system. In our model we have *accident(Car)* and *removeAccident(Car)* as shared labels. Attributes that are used as parameters (just Car in our case) are preserved in the projection. This ensures that the actual parameters in labels are preserved and are used consistently in local views, so that synchronisation over shared labels leads to the correctly composed model in PEPA and PRISM.

Projection of Rules extended to NACs In the top-down approach, projection of rules is extended to NACs. When a rule is decomposed into subrules, so are the NACs. In the composition of a rule from its subrules we get a global rule and its NACs.

Performance Analysis We derive CTMCs from GROOVE LTSs by our CTMC Generator tool for stochastic analysis of graph transformation systems. These CTMCs are further translated into PEPA and PRISM which provide a rich environment for performance checking and properties analysis.

Formalization We have formalised notions of views for typed attributed graph transformation systems for decomposition and composition of systems. We have explained the conditions for correctness, i.e., the equivalence of the resulting synchronized LTS with the one derived directly from a monolithic system.

11.3 Future Work

In modularity approaches, what other conditions could be satisfied for composing more complex systems, e.g., dependencies and conflicts between views. We envisage language support for the bottom up approach to composing systems from views and investigate how the modularity achieved can be extended to the stochastic aspect. Another concern is the limitation of the present theory to the DPO approach, while GROOVE supports the more general SPO-like behaviour. This requires an encoding of rules that delete nodes with an unbounded number of incident edges, typically leading to additional states and transitions.

Bibliography

[1] W. G. Anderson. *Continuous-Time Markov Chains*. Springer, 1991.

[2] N. Arijo and R. Heckel. View-based Modelling and State-Space Generation for Graph Transformation Systems. In *GT-VMT'12, 11th International Workshop on Graph Transformation and Visual Modeling Techniques, March 24-25, 2012, in Tallin, Estonia* .

[3] N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore. Modular Performance Modelling for Mobile Applications. In Samuel Kounev, Vittorio Cortellessa, Raffaela Mirandola, and David J. Lilja, editors, *ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011*. ACM, 2011.

[4] C. Baier, B.R.H.M. Haverkort, H. Hermanns, and J.P. Katoen. Model checking continuous-time markov chains by transient analysis. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 358–372, Berlin, 2000. Springer Verlag.

[5] P. Baldan, H. Ehrig, and B. König. Composition and Decomposition of DPO Transformations with Borrowed Context. In Andrea Corradini,

Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin / Heidelberg, 2006.

[6] D. Berndl and N. Koch. Automotive scenario: Illustrating service specification, 016004. Technical report, Sensoria, August 31, 2007.

[7] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In C. Priami, editor, *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, number 2874 in LNCS, pages 88–150. Springer, 2003.

[8] E. Brinksma and H. Hermanns. Process algebra and markov chains. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 183–231. Springer Berlin / Heidelberg, 2001.

[9] R. De Nicola, G.L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315, 1998.

[10] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. MoSL: A Stochastic Logic for StoKlaim. Technical Report ISTI-06-35, 2006.

[11] R. De Nicola, J.-P. Katoen, D. Latella, and M. Massink. STOKLAIM: A Stochastic Extension of KLAIM. Technical Report 2006-TR-01, ISTI, 2006.

[12] R. De Nicola, J.P. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.

[13] F. Drewes, P. Knirsch, H.-J. Kreowski, and S. Kuske. Graph transformation modules and their composition. In *Proc. of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, AGTIVE '99, pages 15–30, London, UK, UK, 2000. Springer-Verlag.

[14] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[15] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[16] 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, Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

[17] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engeneering*, 7(4):457–477, 1997.

[18] S. Gilmore, J. Hillston, M. Ribaudo, and L. Kloul. PEPA nets: A structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, October 2003.

[19] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[20] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal*, volume 1382 of *LNCS*, pages 138–153. Springer Verlag, 1998.

[21] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Handbook of graph grammars and computing by graph transformation. chapter A view-based approach to system modeling based on open graph transformation systems, pages 639–668. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[22] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. *A view-based approach to system modeling based on open graph transformation systems*, pages 639–668. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[23] R. Heckel, G. Lajios, and S. Menge. Modulare Analyse Stochastischer Graphtransformationssysteme. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Proc. Software Engineering 2005*, volume 64 of *Lecture Notes in Informatics*, pages 141–152. GI, March 2005.

[24] R. Heckel, G. Lajios, and S. Menge. Stochastic graph transformation systems. *Fundamenta Informaticae*, 74, 2006.

[25] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 239–248, Chicago, June 2005. IEEE Computer Society Press.

[26] J. Hillston. Tuning systems: From composition to performance. *The Computer Journal*, 48(4):385–400, May 2005. The Needham Lecture paper.

[27] M. A. Hoff, K. G. Roggia, and P. B. Menezes. Composition of transformations: A framework for systems with dynamic topology. *International Journal of Computing Anticipatory Systems*, 14:259–270, 2004.

[28] N. Koch and D. Berndl. D8.2a: Requirements modelling and analysis of selected scenarios automotive case study, 016004. Technical report, Sensoria, August 31, 2007.

[29] H.-J. Kreowski and S. Kuske. Handbook of graph grammars and computing by graph transformation. chapter Graph transformation units and modules, pages 607–638. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[30] H.-J Kreowski, S. Kuske, and G. Rozenberg. Concurrency, graphs and models. chapter Graph Transformation Units — An Overview, pages 57–75. Springer-Verlag, Berlin, Heidelberg, 2008.

[31] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic model checking in practice: Case studies with PRISM. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):16–21, 2005.

[32] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[33] S. Lack and P. Sobocinski. Adhesive categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer Berlin / Heidelberg, 2004.

[34] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. Wiley, 1993.

[35] J. R. Norris. *Markov Chains*. University of Cambridge, 1997.

[36] B. Plateau and J.M. Fourneau. A methodology for solving Markov models of parallel systems. *Journal of parallel and distributed computing*, 12(4):370–387, 1991.

[37] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and L. Lo Presti, editors, *Proc. 3rd Workshop on Automated Verification of Critical Systems*, Tech. Report DSSE-TR-2003-2, pages 150–160. University of Southhampton, 2003.

[38] A. Rensink. The GROOVE simulator: A tool for state space generation. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[39] A. Rensink. Compositionality in graph transformation. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, editors, *Automata, Languages and Programming (ICALP), Bordeaux, France*, volume 6199 of *Lecture Notes in Computer Science*, pages 309–320, Berlin, July 2010. Springer Verlag.

[40] A. Rensink. Compositionality in Graph Transformation. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul Spirakis, editors, *Automata, Languages and Programming*, volume 6199 of *Lecture Notes in Computer Science*, pages 309–320. Springer Berlin / Heidelberg, 2010.

[41] A. Rensink. A first study of compositionality in graph transformation. Technical Report TR-CTIT-10-08, Centre for Telematics and Information Technology University of Twente, Enschede, February 2010.

[42] M. Tribastone, A. Duguid, and S. Gilmore. The PEPA Eclipse Plug-in. *Performance Evaluation Review*, 36(4):28–33, March 2009.

[43] M. Wirsing, M. Hölzl, L. Acciai, F. Banti, A. Clark, A. Fantechi, S. Gilmore, S. Gnesi, L. Gönczy, N. Koch, A. Lapadula, P. Mayer, F. Mazzanti, R. Pugliese, A. Schroeder, F. Tiezzi, M. Tribastone, and D. Varró. Sensoria patterns: Augmenting service engineering with formal analysis, transformation and dynamicity. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 170–190. Springer, 2008.