Modelling and Analysis using Graph Transformation Systems

by

Zarrin Langari

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

Abstract

Communication protocols, a class of critical systems, play an important role in industry. These protocols are critical because the tolerance for faults in these systems is low and it is highly desirable that these systems work correctly. Therefore, an effective methodology for describing and verifying that these systems behave according to their specifications is vitally important. Model checking is a verification technique in which a mathematically precise model of the system, either concrete or with abstraction, is built and a specification of how the system should behave is given. Then the system is considered correct if its model satisfies its specification. However, due to their size and complexity, critical systems, such as communication systems, are notoriously resistant to formal modelling and verification.

In this thesis, we propose using graph transformation systems (GTSs), a visual semantic modelling approach, to model the behaviour of dynamically evolving communication protocols. Then, we show how a GTS model can facilitate verification of invariant properties of potentially unbounded communication systems. Finally, due to the use of similar isomorphic components in communication systems, we show how to exploit symmetries of these dynamically evolving models described by GTSs, to reduce the size of the model under verification.

We use graph transformation systems to provide an expressive and intuitive visual description of the system state as a graph and for the computations of the system as a finite set of rules that transform the state graphs. Our model is well-suited for describing the behaviour of individual components, error-free communication channels amongst the components, and dynamic component creation and elimination. Thus, the structure of the generated model closely resembles the way in which communication protocols are typically separated into three levels: the first describing local features or components, the second characterizing interactions among components, and the third showing the evolution of the component set. The graph transformation semantics follows this scheme, enabling a clean separation of concerns when describing a protocol. This separation of concerns is a necessity for formal analysis of system behaviour.

We prove that the finite set of graph transformation rules that describe behaviour of the system can be used to perform verification for invariant properties of the system. We show that if a property is preserved by the finite set of transformation rules describing the system model, and if the initial state satisfies the property, then the property is an invariant of the system model. Therefore, our verification method may avoid the explicit analysis of the potentially enormous state space that the transformation rules encode.

In this thesis, we also develop symmetry reduction techniques applicable to dynamically evolving GTS models. The necessity to extend the existing symmetry reduction techniques arises because these techniques are not applicable to dynamic models such as those described by GTSs, and, in addition, these existing techniques may offer only limited reduction to systems that are not fully symmetric. We present an algorithm for generating a symmetry-reduced quotient model directly from a set of graph transformation rules. The generated quotient model is bisimilar to the model under verification and may be exponentially smaller than that model.

Dedication

To:

Hoori and ***Hassan***

Mehran and ***Khorshid***

به:

مادر و پدر

مهران و خورشید

Contents

List of Figures **x**

1 Introduction **1**

1.1 Formal Verification 2

1.2 Problem Statements and Contributions 4

1.2.1 Contributions 5

1.2.2 Previous Work 8

1.3 Thesis Organization 10

2 Background **12**

2.1 Model Checking 12

2.1.1 Software Model Checking 15

2.1.2 Symmetry Reduction 16

2.2 Graph Transformation Systems 21

2.2.1 Directed Labelled Graphs 22

2.2.2 Graph Morphism 22

2.2.3 Graph Transformations 23

2.2.4 Tool and Notations 30

3 Formal modelling using GTS **33**

3.1 Introduction 33

3.2 Distributed Feature Composition Protocol 34

3.2.1 Basic Semantics of DFC Architecture 35
3.3 Visual Semantics Using GTS 39
3.4 Evaluation 44
3.4.1 Case Study 44
3.5 Related Work 48
3.6 Conclusion 50

4 Verification of Dynamic Systems using GTS 65

4.1 Introduction 65
4.2 Motivating Problem: Reversible Features in DFC 66
4.2.1 Invariant-Ordering Property of Reversible Features 67
4.2.2 Example 68
4.3 Verification Through GTS 69
4.3.1 Proving Properties using Structural Induction 75
4.4 Experiments with DFC 78
4.5 Related Work 81
4.6 Conclusion 82

5 Symmetry for the Analysis of Dynamic Systems 83

5.1 Introduction 83
5.2 Symmetry in Dynamic GTS Models 85
5.2.1 Group of Symmetries for Graphs of Different Sizes 86
5.2.2 GTS Symmetry 87
5.3 GTS Bisimulation 89
5.3.1 Generating A Bisimilar Quotient 89
5.4 Vertex Bisimulation 95
5.4.1 Property Preservation 96
5.5 GTS model of an Evolving Toroidal Mesh 100
5.6 Related Work 102
5.7 Conclusion 103

6 Conclusion **108**

6.1 Summary 108

6.2 Future Work 110

Bibliography **131**

List of Figures

2.1 Communication graph of mutual exclusion. 17

2.2 Kripke model for mutual exclusion. 18

2.3 The quotient model. 20

2.4 H is a subgraph of G. 23

2.5 A partial mapping from graph G to H 24

2.6 Applying rule r to graph G and deriving H from G in DPO 27

2.7 Direct derivation in double-pushout 28

2.8 Applying rule P to graph G and deriving H from G in SPO. 28

2.9 Direct derivation in single-pushout 29

2.10 A simple GROOVE transformation rule. 31

2.11 Negative Application Condition (NAC) 31

2.12 A GROOVE transformation rule with a NAC. 32

3.1 A DFC usage with rectangles showing feature boxes and arrows showing the direction of the call request. 37

3.2 Caller is communicating via the channel **ch** to a component at its right, sending its setup signal to the router through **box-out** channel. 38

3.3 Caller Process Finite-State Machine. !: signal transmission, ?: signal reception 41

3.4 Graph of Communicating Finite-State Machines. 42

3.5 Telephony usage for CW features as Src and Trg features. 45

3.6 The graph for Call Waiting usage. 46

3.7 A Graph derivation step representing the third level of computation by the application of the transformation rule *JoinCallee* adding an edge between target CW feature and the callee. 47

3.8 GTS rules for building a usage and its dynamic growth. 52

4.1 A usage with reversible boxes A, B, and D. 66

4.2 Specifications about reversible features 68

4.3 Interaction of three reversible features in a usage. 69

4.4 A proposition graph φ is satisfied by graph s. 72

4.5 Initial state, four GTS rules, and three propositions for a linked list. 76

4.6 Property satisfaction 77

4.7 The GTS rule ActivateCallerReverseFeature. 79

4.8 Propositions P, Q in the property $AG(\phi)$. 80

5.1 a) A ring with three processes. b) A 3×3 toroidal mesh. 84

5.2 Quotient generation algorithm 90

5.3 Graph permutation for rings 91

5.4 LTS of a ring with three processes. 92

5.5 Reduced LTS of a ring. 93

5.6 Two atomic proposition graphs. 98

5.7 Mutual exclusion with single token in toroidal mesh. 100

5.8 Initial state and GTS rules of a 2×2 toroidal mesh. 105

5.9 Transformation rules for addition of a vertical ring. 107

6.1 A FSM graph. 111

6.2 A local transformation rule. 112

6.3 Transformed FSM graph. 113

6.4 A rule to pass messages between components. 113

6.5 InsertDevice and InsertFeature-Src rules with the addition of ports. 114

6.6 A feature with several in- and out- ports. 116

6.7 Source CW feature joins an existing usage. 116

6.8 Applying the rule PassTokenNtoN. 117

Chapter 1

Introduction

Communication protocols and network technologies enable the construction of computer systems with several processors connected together. These protocols may be implemented by hardware, software, or a combination of the two. In these systems, each processor provides part of the functionality of the larger system. Ensuring the correct behaviour of these processors, both in isolation and interconnection with the other components, is important in guaranteeing the correct behaviour of the larger system. However, the complexity of these systems increases the risk of failure in both the hardware and software components.

Currently, there is increasing use of communication protocols in both mission and safety-critical systems. These systems require the highest level of assurance to avoid serious consequences due to system failure. These consequences may include loss of life and severe injuries for safety-critical systems such as medical devices, medical monitoring, automobiles, aviation, etc.; or they may include large-scale environmental damage and considerable economic loss for mission-critical systems such as communication protocols, air-traffic control systems, train interlocking systems, and automotive systems. A typical example of such a system is a health monitoring device in which Internet-based gathering of data from large numbers of patients requires high levels of assurance about the integrity of the communication mechanisms in use.

Therefore, ensuring the reliability and correctness of these systems is a crucial and, indeed, substantially difficult task [LP05]. But how can one ensure the correctness? Here correctness, explicitly is functional correctness of a program. The functional correctness is implicitly defined by a *specification* that describes the desired behaviour of a software program or a hardware system. Thus, a program is correct if it satisfies its specification. Checking whether the specification is met is called the *verification* problem. Verification is done using several methodologies, among which the most widely used is *testing*. In testing, we check whether a number of executions work correctly given a subset of the input parameters. This task is done by comparing the generated output with the expected set of output data. However, for large and complex systems, testing all the program computations is not feasible. Thus, the correctness analysis is not comprehensive.

In order to obtain a high degree of assurance, it is important to perform stringent analysis on all possible behaviours at the system modelling and design stage. *Formal verification* is a methodology designed to verify whether a system is correct with respect to its specification, and it is the main focus of this thesis as applied to communication protocols.

1.1 Formal Verification

Formal verification has been introduced to enable us to demonstrate that a design model behaves as required. This method provides rigorous mathematical proof techniques that use as inputs a formal (mathematical) model of the system and a mathematically precise system specification. These techniques underpin an error-free design and implementation by proving that the code conforms to the system specification. In short, they provide an exhaustive analysis of all behaviours of a software system, in contrast to testing, which considers a subset (often a small subset) of the system executions. As a first step toward formal verification, we need to construct a mathematical model of the system, and to have a precise specification stated in a formal language.

Modelling and specification languages have many varieties. There are two main categories of formal verification techniques: *theorem proving* (*cf.* [GM93]) and *model checking* [CE81, QS82, EMCGP99]. In theorem proving, both the model and the specification are expressed by a logical formula and a proof is constructed to show that the specification is implied by the model. These techniques often require significant input from verification engineers. In contrast, model checking is an automated technique that is based on two concepts: the state that the system is in and the actions that can be taken in a specific state. In model checking, which is the technique we have chosen in this work, the model is usually built as a finite-state machine and the specification is given as a logical formula.

As a specification language, *temporal logic* [Pnu77] is a type of logic that is used for specifying reactive systems in which the concept of temporal ordering is an issue. Reactive systems continuously interact with an environment over which they have little or no control, for example, the air-traffic control system. Temporal logic and its variations specify properties on the behaviour of these systems as they execute over time. The main two categories of properties regarding system correctness are typically *safety* and *liveness* properties [Lam77]. A safety property specifies that "a critical error never occurs in any computations of the program". Commonly, it states that a bad state should not be visited in order to have a safe program. These properties are an invariant of the model under verification. In contrast, liveness properties focus on "something good eventually occurs in every computation of the program". In this work, our focus is on safety properties, which are important in safety-critical and mission-critical systems.

To perform model checking, other than selecting the appropriate modelling technique and the specification language, the main challenge is the *state explosion problem.* This problem is described by the exponential growth of the model under verification in contrast to the size of the program describing the model. This issue presents an obstacle in the application of model-checking techniques to even modestly-sized software programs. To solve this problem, one methodology is to symbolically represent the large state space as a Binary Decision Diagram [Bry86] and to perform the verification on the symbolic model [McM93]. *Symbolic* model checking is more appropriate for hardware systems that have a static model. In contrast to symbolic model checking, *explicit-state* model checking builds an on-the-fly model using an explicit representation of the transition system [Hol97]. This approach is appropriate for dynamic software systems with a small number of processes. For systems with a large number of components, the model needs to be reduced for verification.

Methodologies such as *partial-order reduction* [Pel98] and *symmetry reduction* [ID96, CEFJ96, ES96] are used to reduce the model under verification to a model with fewer states and transitions. Both of these methods can be applied on the fly during model checking, and hence are appropriate for reducing the model in explicit-state model checking. In partial-order reduction, the focus is on reducing the number of transitions by exploiting the commutativity of independent, concurrent transitions. However, for some systems, there is little or no benefit in using partial-order reduction if all actions are dependent and there is a high level of interaction between processes. In symmetry reduction, the focus is on reducing the number of states by exploiting the architectural structure of the states and reducing symmetric states to one of the many states that are equivalent up to permutation. Since communication protocols are often constructed of many similar components, symmetry is often a feature of them and symmetry-reduction techniques are appropriate for reducing their state space. However, existing symmetry-reduction techniques address fixed-size models. In this thesis, we have extended symmetry-reduction techniques to address the reduction of dynamic models.

In the following section, we provide a consolidated version of the above problems in the model checking of dynamic software systems and then present our contributions for solving them.

1.2 Problem Statements and Contributions

Many hardware design flaws can be detected using model checking, and today there are increasing demands for using model-checking techniques in software verification as well. However, many software systems such as communication systems are highly dynamic. Examples of systems with this feature include scalable network architectures such as ring, hypercube, and toroidal mesh in which one process or a set of processes can be added to or deleted from the network; systems for heap allocation in which the memory can be allocated or deallocated at runtime; and IP-telephony protocols in which telephony features can dynamically be assembled into or taken apart from the connection session. Considering the dynamic evolution of software systems, there are difficulties in using existing model checkers for model checking these systems.

Modelling using the existing techniques for fixed-size systems, such as finite-state machines results in a huge number of states and computations for complex, dynamic communication systems. The analysis of these systems becomes difficult, or even infeasible, as the number of processes increases. Thus, we recognize the need for modelling and analysis techniques that enable us to analyze dynamic systems with many processes.

Therefore, the first difficulty in model checking dynamic systems relates to *constructing a formal model* and presenting expressive semantics in modelling the dynamic behaviour of a system. The second difficulty is related to the *verification* of potentially unbounded constructed models and the state explosion in these models. The verification problem can be narrowed down to two different problems: *generating a reduced model* and *formalizing the property*. Generating a reduced model and property specification is complicated because of the dynamic nature of the above systems.

This thesis aims at addressing these difficulties and claims to present utility of Graph transformation Systems for:

- **formal modelling of dynamic software configurations.**
- **facilitating verification of some invariants using structural induction.**
- **enabling symmetry reductions due to symmetries in dynamic software configurations.**

In the next section, we describe the contributions made in addressing this claim and stated problems. Then, in the following section, we recognize the previous work as it relates to each problem. Relevant work is also reported at the end of each chapter.

1.2.1 Contributions

While much of the model-checking work has addressed fixed-size, finite-state systems [EMCGP99, McM93], (*cf.* [Hol97]), there is a need for addressing the analysis of potentially unbounded communication systems with a dynamically evolving topology. The main contribution of this thesis is to address the above challenges facing the formal verification of dynamically evolving communication systems. Thus, the focus of this thesis is on describing a formal model for dynamic evolution of communication systems, verification of a specific class of properties for these systems without explicit analysis of the system model, and development of symmetry reduction techniques applicable to dynamically evolving system models.

Formal Modelling

The greatest benefits of software model-checking methods would be their usage in the verifying critical components of the software, and in places where traditional approaches are not effective. Therefore, a modular, component-based software model in which critical components can be identified, is the most appropriate way to take advantage of model-checking techniques. In addition, graphical representation is an intuitive way of describing the topology in which components are connected together in a communication network [AEH+99, Hec98]. Other than the graphical representation of topology, we need a modelling formalism to be able to describe the topological changes and computations of the system. *Graph transformation systems* (GTSs) [Roz97, EEPT06] provide such a formalism.

GTSs [Roz97, EEPT06], also called *graph grammars*, provide a mathematical basis for formal modelling. They are graph-based formalisms that are a generalization of string grammars. Graph transformations can be used to specify how a model is built initially and how it evolves thereafter. The evolution aspect makes graphs and graph transformations highly suitable for describing communication systems. The fundamental aspect in describing the changes that occur in graphs is that these changes are not arbitrary, but are controlled through a set of transformations called *transformation rules.* Thus, the dynamic changes of a system can be captured by these rules. In addition, visual modelling is a natural way of modelling component-based systems such as communication systems.

In the first part of this thesis, we propose a visual semantic modelling approach using GTS to describe behaviours of dynamically evolving communication protocols. Our visual modelling provides a formalism that follows the natural describtion of communication protocols. We describe states of the system and the current topology of a set of components that participate in the system as graphs and consider the topological reconfiguration of the component set such as the creation and removal of components as well as the connection changes. This modelling was first reported in [LT06].

Verification

Connection-oriented communication protocols require a connection session to be established prior to data transfer [Zav04]. In these protocols, services are described by the component set. Dynamic evolution of a communication system changes the topology of the system as new components are added to or existing ones deleted from the connection session. This dynamic evolution may lead to the violation of inter-component specifications, thereby causing an undesirable behaviour of the component set [CKMRM03, CGL+94].

Therefore, we aim to ensure the correct behaviour of the component set in all computations of the system. Invariant properties (safety properties) express important properties of this type. In addition, in distributed communication protocols, verifying properties of component compositions is problematic due to the state-explosion problem and may not even be decidable [BZ83]. Thus, avoiding explicit analysis of the system state space is desirable.

In the second part of this thesis, we present our method for verifying high-level invariant properties of connection-oriented protocols given as GTS model descriptions. In this work, we address the verification problem for a class of systems with potentially unbounded state spaces. We show that an invariant system property expressed by a temporal modality and atomic propositions modelled as graphs can be verified by the analysis of a finite set of transformation rules describing the GTS system model. Therefore, our verification method avoids the explicit analysis of the behaviours and potentially enormous state space that the transformation rules encode. The results of this work first appeared in [LT09].

Symmetry Reduction in Dynamic Systems

Avoiding explicit analysis of the system model for verification is not always possible, and thus we still need methods to reduce the state space of the model to prepare it for verification. Symmetry reduction is the method of choice for systems with many similar components. Systems that have this characteristic include multi-process systems such as rings, hypercubes, and tori. However, existing symmetry-reduction methods [ID96, CEFJ96, ES96] are not applicable to dynamically evolving multi-process systems such as communication systems. In addition, the advantages that graph-based models provide for the modelling and analysis of dynamically evolving systems cannot be fully exploited, because symmetry-reduction methods have not been presented to reduce graph-based models of the systems.

In the third part of this work, we define a notion of symmetry for dynamically evolving symmetric multi-process systems modelled as GTSs that may grow to a given maximum size. The explicit GTS semantic modelling can directly be exploited for reducing symmetric systems. Our symmetry-reduction technique is based on generating a reduced state space directly from the set of graph transformation rules that define the model under verification. For this purpose, we define the notions of *GTS symmetry*, and *GTS bisimulation* based on graph isomorphism. With GTS bisimulation, we describe an on-the-fly algorithm that builds a symmetry-reduced model using the set of graph transformation rules that describe the full dynamic behaviour of the system.

To improve the reduction for symmetric GTS models, we define *vertex bisimulation*. Vertex bisimulation describes an equivalence relation on state graphs based on their set of vertices and can be used in our algorithm for symmetry reduction resulting in exponential state-space saving. We also show that two vertex-bisimilar GTS models can prove the same reachability properties. We have reported this part of the work in [LT10].

1.2.2 Previous Work

Formal Modelling

Heckel [Hec98] and Taentzer [TKFV99] have both explicitly noted the omission of reactive protocols from the graph-transformations literature. Heckel has used graph transformations for view-based modelling of reactive systems. His approach considers a view for an incomplete specification describing only a certain aspect of the overall system. This approach does not explicitly distinguish between topology and local states.

Taentzer also has used graph transformations for modelling distributed systems. Her work describes a distributed system by global transformations. This implies a limitation on supporting the implementation of distributed systems, because often global views do not exist for a distributed system. Her method uses an algebraic approach, double-pushout [CMR+97, CEH+97], for the description of transformation rules. This method is dependent on the checking of specific conditions, and this checking is necessary for the application of transformation rules. On a distributed system, checking these conditions is not decidable as it crosses the boundaries of locality. This method also does not use attributed graphs that raise the power of specifying local states. However, our modelling is closest to the work of Taentzer.

Verification

To provide methodologies for the verification of formal models of systems described as graph transformations we note the following work. The research on the verification of graph transformations, can be divided into the verification of finite-state GTSs and infinite-state ones. Verification of finite-state GTSs includes two major trends: 1- approaches that encode graph transformations as the input language of an existing model checker. 2- approaches that directly build the state space for verification.

The main advantage of the first trend [Var03, DFRdS03] is its adaptability to other off-the-shelf model checkers, but the disadvantage is loosing the expressiveness of visual modelling by translating graph transformations into a textual description of current model checkers. Furthermore, it creates an overhead on translation from graphs to the language of model checkers.

The second trend considers the development of model-checking approaches for graph transformation models. With this approach, very little of the theory and methodologies for traditional model checkers can be applied directly. Approaches in this trend apply all transformation rules on all possible matchings and explicitly generate the state space. The main work in this trend belongs to Rensink in building the graph transformation tool GROOVE [Ren03, KR06]. This approach works well for small dynamic systems, but not for large dynamic communication systems. A disadvantage of this approach is that the whole state space will be generated and the existing state-space-reduction techniques such as partial-order reduction and symmetry reduction cannot be used to reduce the model.

In another research direction, graph transformation models are approximated by Petri nets via an unfolding construction. This trend considers a graph transition system as an extended Petri net and transfers techniques from that area [BCK01b]. A lack of analysis techniques on GTS on the one hand, coupled with a rich literature on analysis techniques of Petri nets on the other, has resulted in this trend [BCK01b, BCK04, BKR05]. While Petri nets can be used to model the behaviour of finite asynchronous systems and the causal relation between executions, they are not as expressive as graph transformations which are Turing-complete formalisms [BKR05].

Another research direction investigates abstractions of graphs according to local structures of their nodes (e.g. the number of incident edges) and edges; the abstract graph is called a shape graph [Ren04a, Ren04d, RD05b, BRKB07]. Defining a graph transformation system on shape graphs allows an over-approximation of the system behaviour while keeping a finite space. Although this method has been defined and understood, only preliminary ideas have been put forth on how to apply the transformations on shape graphs. In addition, the abstraction is not precise, and concretization of the same abstract graph is very different in shape and structure.

Symmetry Reduction

Existing symmetry-reduction techniques [ID96, CEFJ96, ES96, ET99, TW09] that generate a reduced, bisimilar model that alleviate state explosion in model checking are not applicable to dynamic models such as GTSs.

To our knowledge, in the area of GTS models only the work of Rensink [Ren06, Ren08] has directly addressed symmetry. In [Ren06], a generalized definition of bisimulation is used that does not guarantee isomorphism, whereas our work uses graph isomorphism to define equivalence classes and symmetry in GTS models. Also, it does not give rise to a canonical representation of states, whereas we give an algorithm for generating a GTS-bisimilar quotient. In addition, in our work we specifically address symmetry for GTS models that are not fully symmetric, which means that they are not invariant under all permutations of the process indices, and for dynamically evolving GTS models of multiprocessor systems. These subjects are omitted in the above work.

1.3 Thesis Organization

In Chapter 2, we present the required background for this thesis. In the first section of this chapter, we introduce model checking in general, and software model checking in particular. We describe the basic definitions of model and temporal formulas for property specification. Also, as a method for model reduction, we explain symmetry reduction and related concepts such as bisimulation. In the second section of this chapter, we present graph transformation systems theory with the focus on two well-known algebraic approaches in GTSs and their differences: Double-Pushout (DPO) and Single-Pushout (SPO). The reason for describing both approaches is that SPO was developed after DPO and a large amount of the work in GTS literature is based on DPO; however, due to its restrictions, we have chosen our modelling based on SPO.

In Chapter 3, we briefly introduce AT&T's IP-telephony system, Distributed Feature Composition (DFC) that we have chosen as our case study. We describe our modelling of systems based on graph transformations and show how DFC is modelled in GTS. In Chapter 4, we describe our methodology for facilitating the verification of invariant properties in GTS models without building the full system state space. In this direction, we give definitions of graphs that are used to abstract the components' connectivity patterns and to specify properties using graph-based propositions. We describe a requirement of DFC that motivated us to perform verification. We formulate this requirement as an invariant property and show that the set of GTS rules can be analyzed to verify this property.

In Chapter 5, we describe our results on symmetry reduction of dynamic GTS models. We present our algorithm for generating a GTS-quotient model and prove that the original GTS model and the generated quotient are bisimilar. We also describe vertex-bisimulation as a more efficient approach for verification of GTS models. We prove that two vertex-bisimilar models satisfy the same reachability properties. We note that chapters 3, 4, and 5 each contains a section that explains the related work.

Finally, we conclude in Chapter 6 with a summary of the thesis and an outline of directions for future work.

Chapter 2

Background

In this chapter, we give the needed background knowledge of model checking and graph transformation systems. In the first section, the general approach to model checking and its specialization to software is defined, and symmetry reduction techniques such as abstraction methods for large state space models are introduced. In the second section, graph transformation concepts such as graphs, graph morphism, and graph transformation rules are described.

2.1 Model Checking

Model checking is a way of formally verifying a system and checking if the system meets its specification [CE81, QS82, EMCGP99]. To perform model checking, usually the specification or property of the system to be checked is expressed as a temporal logic formula [Pnu77], the system is modelled in a state-transition structure called a *Kripke structure*, and an exhaustive search is done on the model to check if the property is satisfied. If the specification is met, the answer "true" will be returned; otherwise, the model checker returns a counterexample in the form of an execution path.

Definition 2.1. *Let AP be a set of atomic propositions, a Kripke structure M is a four tuple: $M = \langle S, R, I, L \rangle$. S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow 2^{AP}$ is a labelling function that maps each state with the finite set of atomic propositions true in that state.*

To reason about the system properties expressed in temporal logic formulas,f, we need to build a system model that semantically shows how the system evolves over time. A Kripke structure defines such a model. In this model, each state will be labeled with the set of label(s) of subformulas of f which are true in that state. In a state s, if an atomic proposition p is true in that state, then $p \in L(s)$ and we write $s \models p$ and if $p \notin L(s)$ we write $s \models \neg p$. Using these definitions, we define a computation path or a *run* σ in a model as an ordered set of states chosen from the set of states in the model.

Definition 2.2. *$\sigma = (s_0, s_1, s_2, ...)$ is a path or run in a Kripke structure $M = \langle S, R, I, L \rangle$, if $s_0 \in I$ and $(\forall i, 0 \leq i : \exists t, t \in R$ and $t = (s_i, s_{i+1}))$.*

Temporal logic expresses the way in which the behaviour of the system evolves over time. The temporal characteristic of this logic makes it suitable to describe ordering properties, and thus suitable for specifying concurrent reactive protocols. The two main temporal logics used in model checking are Linear Temporal Logic (LTL) [Pnu77] and Computation Tree Logic (CTL) [CE81]. LTL is used to express the linear- time properties of the system along a computation path, and CTL is used to express the branching-time properties. In this thesis, we use the CTL operators to define system properties, hence, here we briefly outline CTL syntax and semantics.

In CTL, semantic interpretations are given with respect to computation trees rather than paths. Thus, path quantifiers are also used as CTL operators. **A** is a path quantifier that stands for *all paths*, and **E** stands for *there exists a path*. Other operators in both LTL and CTL are temporal operators: $\mathbf{F}p$ (eventually p), $\mathbf{G}p$ (always p), $\mathbf{X}p$ (next time p), and p **U** q (p until q). Path quantifiers in CTL precede temporal operators. The CTL definition is formally given as:

Definition 2.3. *Given a set of atomic propositions, CTL formulas are defined recursively as:*

- *Every atomic proposition is a CTL formula.*
- *If ϕ and ψ are CTL formulas then so are:*
  - $\neg\phi$ • $\phi \vee \psi$ • $\boldsymbol{EG}\phi$ • $\boldsymbol{EX}\phi$ • $\boldsymbol{E}[\phi \boldsymbol{U} \psi]$

  The remaining operators such as $\wedge$, $\boldsymbol{AF}$, $\boldsymbol{EF}$, $\boldsymbol{AG}$, $\boldsymbol{AX}$, $\boldsymbol{A}[\phi \boldsymbol{U} \psi]$, etc. are derived from theses formulas.

The semantics of CTL formulas are defined with respect to a Kripke structure M. The meaning of $M, s \models \varphi$, is that the formula φ is true in state s of the model M.

- $M, s \models p$ iff $p \in L(s)$
- $M, s \models \neg\varphi$ iff $M, s \not\models \varphi$
- $M, s \models \varphi \vee \psi$ iff $M, s \models \varphi$ or $M, s \models \psi$
- $M, s_0 \models\ EGp$ iff there exists a path $(s_0, s_1, ...) \in M$, for all $i \geq 0$, $M, s_i \models p$
- $M, s_0 \models\ EXp$ iff there exists a path $(s_0, s_1, ...) \in M, M, s_1 \models p$
- $M, s_0 \models E[\varphi U \psi]$ iff there exists a path $(s_0, s_1, ...) \in M$, and there exists $i \geq 0$ such that $M, s_i \models \psi$ and for all $0 \leq j < i$, $M, s_j \models \varphi$

Model checking has two main approaches: *symbolic* and *explicit-state*. Symbolic model checking [McM93] models the set of states and the transition relation symbolically as Boolean formulas. This approach uses a Binary Decision Diagram (BDD), or more effectively a canonical representation, Reduced Ordered Binary Decision Diagrams (ROBDD), for the set of states and the state-transition model of the system. The BDD representation enables this approach to handle large state spaces. The problem with symbolic model checking is that in its practical application suits static transition relations and not the dynamic creation of software components. Therefore, this method has traditionally been used in hardware rather than software.

Different research trends seek other variations of modelling the system and formulating the specification property, to make them suitable for model checking software and dynamic evolution of the system. In explicit-state model checking [Hol99], the whole state space is generated as a transition system on-the-fly, and properties are defined in temporal logic. All the states in this approach are explored explicitly. Explicit-state model checking uses a forward-analysis approach and is suitable for model checking the dynamic evolution of software systems; therefore, it is appropriate for identifying errors in communications system models.

2.1.1 Software Model Checking

Software model checking is the process of checking a software system is design or code against the specification of the system. Like hardware model checking, a model of the software system is built and the specification is described by a language. The model is usually an abstract representation of the structural or behavioural properties of the software, and the specification is described in a temporal logic language.

There are two major trends that address software model checking and its state-explosion problem. One trend deals with translating programs to an adequate input language for model checkers. The other trend works on improving model-checking techniques to directly deal with software programs. Both of these trends cover abstraction techniques that transform an infinite-state software system to a finite-state system. The process of abstraction or translation may be done manually, semi-automatically as in the SPIN model checker [Hol97], or fully automatically as in Java PathFinder [Hav99, VHB$^+$03] and Bandera model checkers [CDHR00, VHB$^+$03, BHJM07].

The latter trend allows model checkers to deal with the actual description of software systems. Thus, the goal is to extend model-checking techniques to extract the formal model directly from the actual program description. This trend is mainly used for verification of concurrent systems such as communication protocols [DHR$^+$07]. The main methodology used in this trend is explicit-state model checking. However, still one of the main problems with explicit-state model checking is that for complex systems such as communication protocols, the state space may grow exponentially with the number of components, resulting in state explosion. The main approach to address state explosion in this trend is to generate a reduced model on-the-fly to perform verification.

Partial-order reduction [Pel98] and symmetry reduction [ID96, CEFJ96, ES96] are two techniques for reducing a system model. Partial-order reduction exploits the commutativity of concurrent transitions, while symmetry exploits the structure of states to find equivalence relations among them and to build a model that is reduced based on equivalence classes of states. The symmetry reduction technique has been adopted in this thesis and is described in more detail below.

2.1.2 Symmetry Reduction

To reduce the complexity of verification for concurrent software systems, one method is to exploit the symmetry in the systems that have many similar processes or components. The idea is to reduce the model checking of a specification ϕ over a Kripke structure M to the model checking of ϕ over a smaller bisimilar structure $\overline{M}$ [ID96, CEFJ96, ES96]. The structure $\overline{M}$ may be exponentially smaller than M.

In concurrent software systems, we have to deal with all possible combinations of configurations of concurrently executing identical components. The global configuration of such a system may show a great deal of symmetry. The main perception is that the order in which these components' configurations (processes or objects local states) are stored in a global state does not influence the future behaviour of the system. In particular, under *full symmetry* the system model is invariant under all permutations of the process indices. Therefore, full symmetry is achieved when the behaviour is preserved by arbitrarily rearranging processes, and index permutation can be used to define an equivalence relation on symmetric states of the system model.

To describe the symmetry of two states, basic group theory notions are used. In group theory, a symmetry is defined as an automorphism, which is defined in the following section.

Automorphism and Symmetry

For any indexed object b, such as a state, whose definition depends on a finite set of indices I, we can define the notion of *permutation* π which is a mapping that acts on b by simultaneously replacing each occurrence of index $i \in I$ by $\pi(i)$ in b [ES96]. Index permutation is a one-to-one and onto mapping, $\pi : I \rightarrow I$. Therefore, if *Sym* is the set of all permutations on I, then *Sym* I forms a group with functional composition (o) being the group operation [ES96]. This means that if π' and $\pi'' \in Sym\, I$ then $\pi = \pi'' o\, \pi' \in Sym\, I$; if Id is the identity permutation on I then $Id\ o\ \pi = \pi\ o\ Id = \pi$; and for π^{-1}, the inverse of π, $\pi\ o\ \pi^{-1} = \pi^{-1}\ o\ \pi = Id$.

Automorphism is a way of mapping an object to itself while preserving all of its structure. The set of all automorphisms of an object forms a group. For a set, an automorphism is an arbitrary permutation of the elements of the set. For a Kripke structure

$M = \langle S, R, I, L\rangle$, the set of automorphisms of M is $Aut\ M = \{\pi \in SymI : \pi(M) = M\}$. This means that $\pi \in Aut\ M$ precisely when:

1. the mapping $\pi : S \to S$ is one-to-one and onto.
2. $\pi(R) = R$ that is for all s, $t \in S$, $s \to t \in R$ iff $\pi(s) \to \pi(t) \in R$.
3. $\pi(I) = I$, that is for all s, $s \in I$ iff $\pi(s) \in I$.

Example. Figure 2.1 shows the communication graph of two identical processes which try to access a shared resource. This is known as the mutual exclusion problem, in which the objective is to ensure that a resource is never used by more than one process at a time. Each of these processes can be in the local states Critical (C), Trying (T), or Nontrying (N) which respectively show that the process is using the shared resource, is trying to access the resource, or it does not need to have the resource. The Kripke model of this communication system, depicted in Figure 2.2, provides a solution to the mutual exclusion problem. A global state in this model is labelled with local states of all processes. For instance, the state (C_1, T_2) shows that process 1 is in the critical state, and process 2 tries to enter the critical state by accessing the shared resource.

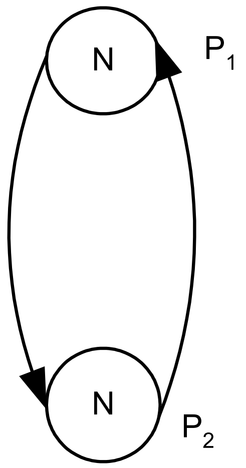


Figure 2.1: Communication graph of mutual exclusion.

In Figure 2.2, $Aut\ M = \{Id, Transpose\}$, where Transpose is a group operation for transposition of indices. For example, for state (C_1, T_2), the $Transpose((C_1, T_2))$ would be (T_1, C_2).

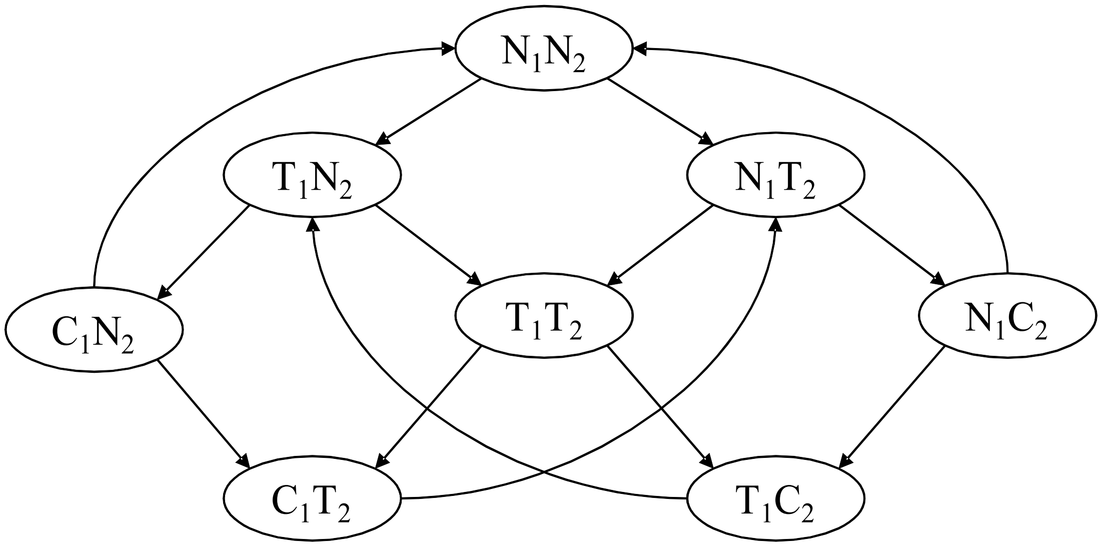


Figure 2.2: Kripke model for mutual exclusion.

For a Kripke model M, *Aut M* forms a subgroup of *Sym* I. Let G be any subgroup of *Aut M*. Two states s and t in S are G-equivalent, $s \equiv_G t$ iff there exists a $\pi \in G$ such that $t = \pi(s)$. Since G is a group, G induces an equivalence relation on M.

To define symmetry, π is extended to the transition relation R so that $\pi((s,t)) = (\pi(s), \pi(t))$.

Definition 2.4 (Symmetry [ES96]). *A permutation π on S is said to be a symmetry of Kripke structure $M = (S, R, L, I)$ if*

1. $\forall s, t \in S, (s,t) \in R \Rightarrow (\pi(s), \pi(t)) \in R$.
2. $L(s) = L(\pi(s))$ *for any* $s \in S$.
3. $\pi(I) = I$.

The symmetries of M form a group under function composition. Model M is said to be symmetric if its symmetry group G is non-trivial.

Quotients

As stated above, symmetry is represented by a group action that acts as an equivalence relation on a structure, thus ideally the reduced state space which is called a quotient of the original model will have only one state representing each symmetry equivalence class. In the Kripke model $M = (S, R, L, I)$, the equivalence relation can be defined on the state labels; i.e. $s \equiv t$ implies $L(s) = L(t)$. The *canonical* quotient structure $\overline{M} = (\bar{S}, \bar{R}, \bar{L}, \bar{I})$ based on the equivalence relation is defined as:

$\bar{S} = \{[s] : s \in S\}$ where $[s]$ is the equivalence class of s with respect to a group G of symmetries,

$\bar{R} = \{([s], [t]) \in \bar{S} \times \bar{S} : \exists s_0 \in [s], t_0 \in [t] : (s_0, t_0) \in R\}$,

$\bar{L}([s]) = L(s)$, and

$\bar{I} = \{[s] : s \in I\}$.

The equivalence class $[s]$ is also called an *orbit* of s. The quotient then is built using the orbits of states. The relation $s \equiv_o t$ defines an equivalence relation on states, given a group G of symmetries on a Kripke model M with respect to permutations on state labels.

Example. Figure 2.3 shows the symmetry-reduced, bisimilar quotient of the model in Figure 2.2.

Bisimulation

The symmetric nature of a model can be exploited to prove interesting properties about the model formulated as temporal formulas. Thus, the properties can be proved over $\overline{M}$, the reduced quotient of the model M, with respect to the relation $\equiv_o$, if $\overline{M}$ is bisimilar to M [ES96, ES97] and bisimilarity is defined as:

Definition 2.5 (Bisimulation)**.** *Let $M = (S, R, L, I)$ and $\overline{M} = (\bar{S}, \bar{R}, \bar{L}, \bar{I})$ be Kripke structures over a set of atomic propositions. A relation $\approx S \times \bar{S}$ is a bisimulation if $s \approx \bar{s}$ implies:*

1. $L(s) = \bar{L}(\bar{s})$,

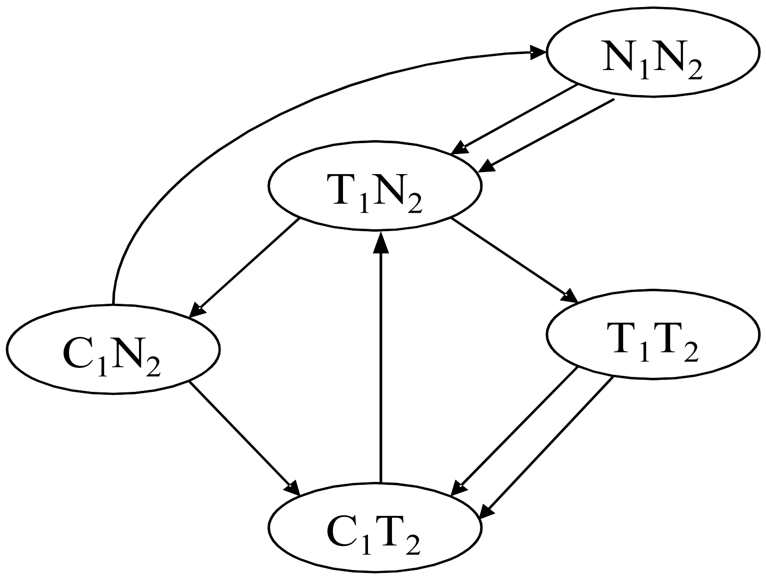


Figure 2.3: The quotient model.

2. *for every $t \in S$ such that $(s,t) \in R$, there exists $\bar{t} \in \bar{S}$ such that $t \approx \bar{t}$ and $(\bar{s},\bar{t}) \in \bar{R}$, and*

3. *for every $\bar{t} \in \bar{S}$ such that $(\bar{s},\bar{t}) \in \bar{R}$, there exists $t \in S$ such that $t \approx \bar{t}$ and $(s,t) \in R$.*

M is then bisimilar to $\overline{M}$, iff for each initial state in I, there is an initial state in $\bar{I}$ such that these states are bisimilar based on the above definition.

Bisimilarity implies that two models satisfy the same CTL properties. Thus, in M and its bisimilar quotient $\overline{M}$, for two states $s \in S, \bar{s} \in \bar{S}$ with $s \approx \bar{s}$ and any CTL formula f whose atomic propositions are invariant under group permutations:

$$M, s \models f \text{ iff } \overline{M}, \bar{s} \models f$$

$\overline{M}$ can be exponentially smaller than M. For example, for full symmetry in n-process systems, all $n!$ permutations of a global state with pairwise distinct local states are orbit-equivalent and can be collapsed into a single abstract state [ES96].

Both the traditional and recent symmetry techniques [CEFJ96, ES96, ET99, TW09] have been defined for Kripke models in which a state is known globally with a set of local states indexed over identical components that build the system. Thus, these techniques are not applicable to systems with states defined as graph models of components. The reason is that, the index permutation does not respect the architecture of the system, whereas each state graph shows the architecture of the system. In addition, these techniques are appropriate for systems with a finite set of states and are not applicable to dynamically evolving models such as graph transformation systems. Thus, with index permutation only one set of symmetries for a fixed-size system are used, whereas in dynamic systems states may contain graphs of different sizes; therefore, we need to define symmetries for different sizes of a system.

2.2 Graph Transformation Systems

Graphs provide a visual demonstration of the structural architecture of a system. In computer science, graphs have been used to represent data structures such as control-flow diagrams, entity-relationship diagrams, Petri nets, visualization of architectural designs, UML and finite automata. *Graph transformations* have been introduced to provide a mechanism by which transformations on graphs can be modelled in a mathematically precise way [Roz97]. Graph transformations can be used to specify how a model is initially built and how it then evolves. The fundamental aspect in describing the changes that happen in graphs is that these changes are not arbitrary, but are controlled through a set of transformations called *production* or *transformation rules.* The basis of a *graph transformation system (GTS)* is this finite set of transformation rules.

The set of transformation rules are comparable to the set of grammar rules for strings in formal language grammars. Thus, similar to the description of a string based on formal language grammars, graph transformation systems present a visual notation that is based on grammar rules. Fundamental elements of the visual notation are graph alphabets and rewriting rules. Therefore, as in the case of string grammars, if a GTS is used to create a graph language, then we have a *graph grammar.* A graph grammar allows us to derive a collection of graphs from an initial graph by applying transformation rules, and to go through all derivable graphs and end with the complete set of derivable graphs. Therefore, the concept of graph transformation in the special case of adding graph elements can be replaced by a graph grammar. Similar to formal language grammars, transformation rules consist of a left hand side and a right hand side. Each transformation rule shows that a graph has been transformed from its left side form to its right side form.

Graphs are quite generic structures which can be encountered in the literature in many variants: directed and undirected, labelled and unlabelled, simple and multiple, etc. [CL96]. The following section defines directed labelled graphs.

2.2.1 Directed Labelled Graphs

Definition 2.6 (Graph [EEPT06, CL96]). *A graph $G = (V, E, Src, Trg, Lab)$ consists of a set V of nodes; a set E of edges; and functions $Src, Trg : E \rightarrow V$, that define the source and the target, respectively, of a graph edge; and the labelling function $Lab : E \rightarrow l$ and $Lab : V \rightarrow l$, where l belongs to a set of labels L.*

If a graph is labelled with attributes or abstract data types such as a number, text, an expression, or a list, it is called an *attributed* graph. In this thesis, we focus on directed attributed labelled graphs.

Definition 2.7 (Subgraph [KKH06]). *Let $G = (V_G, E_G, Src, Trg, Lab)$, one subset of nodes and one subset of edges $(V_X, E_X) \subseteq (V_G, E_G)$ induce a subgraph $X = (V_X, E_X, Src', Trg', Lab')$ with $Src'(e) = Src(e)$, $Trg'(e) = Trg(e)$, and $Lab'(e) = Lab(e)$ for all $e \in E_X$ if and only if there is no edge $e \in E_X$ with $Src(e) \in V - V_X$ or $Trg(e) \in V - V_X$.*

In other words, a subgraph cannot include an edge from the original graph without having the source and target nodes of that edge in the subgraph as well. For example, graph H is a subgraph of graph G in Figure 2.4.

2.2.2 Graph Morphism

Definition 2.8 (Graph Morphism [Roz97]). *Let $G = (V_G, E_G, Src_G, Trg_G, Lab_G)$ and $H = (V_H, E_H, Src_H, Trg_H, Lab_H)$. A graph morphism $f : G \rightarrow H$ maps nodes (V) and edges (E) of graph G to nodes and edges of graph H where $f = (f_v, f_e)$ and $f_v : V_G \rightarrow V_H$ and $f_e : E_G \rightarrow E_H$ are structure-preserving functions. That is, we have for all edges $e \in E_G$,*

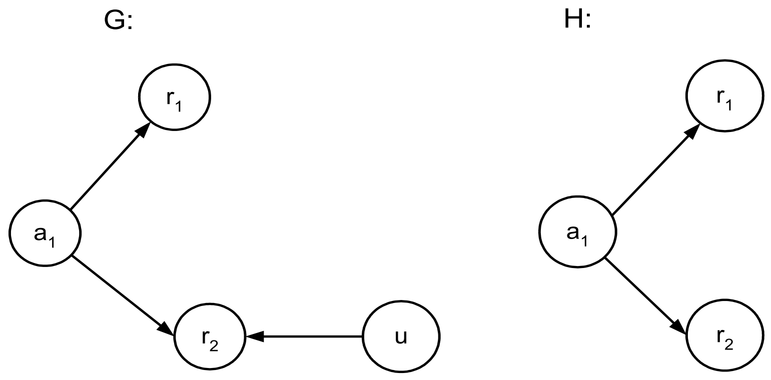


Figure 2.4: H is a subgraph of G.

$f_v(Src_G(e)) = Src_H(f_e(e))$, $f_v(Trg_G(e)) = Trg_H(f_e(e))$, and $Lab_H(f_e(e)) = Lab_G(e)$, $Lab_H(f_v(v)) = Lab_G(v)$. If f_v, f_e are total functions, then we have a total morphism, and if these are partial functions, we have a partial morphism.

For a graph morphism $f : G \to H$, the image of G in H is called a match of G in H, i.e. the match of G with respect to the morphism f is the subgraph $f(G) \subseteq H$ which is induced by (f_v, f_e). Note that f is structure-preserving, and in a structure-preserving mapping, the shape and the edge labelling of the original graph are preserved. If f, respectively f_v and f_e, are bijective functions, then we have an *isomorphism*. We write $G \cong H$ if there exists an isomorphism between the graphs G and H.

If f_v and and f_e map the set of all nodes and edges of graph G respectively, then the morphism is called a *total morphism*. On the other hand, f_v and f_e are *partial morphisms* iff the mapping is not from the whole source graph nodes and edges. An example of a partial morphism has been depicted in Figure 2.5. In this figure, the edge $u \to r_2$ is not mapped to an edge in H, so this is a partial morphism.

2.2.3 Graph Transformations

Transformation Rules

Definition 2.9 (Graph Transformation Rule [Roz97, Ren08])**.** *A transformation rule r is defined as $r : L \xrightarrow{r_1} R$, where L and R are graphs, called the left-side graph and the*

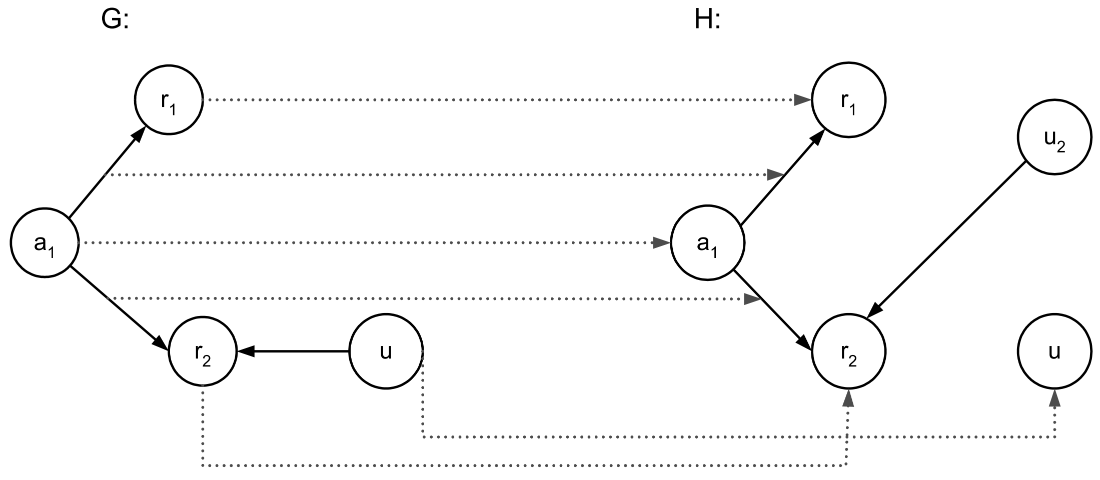


Figure 2.5: A partial mapping from graph G to H

right-side graph of the rule, and there is a partial morphism r_1 between them.

The application of a rule r to a graph G, is based on a total morphism from L to G. We write $G_0 \xrightarrow{r} G_1$ to show that the graph G_0 is transformed to G_1 by the application of rule r. The transformation sequence $G_0 \xrightarrow{r_1} G_1 \xrightarrow{r_2} ... \xrightarrow{r_n} G_n$ is called a *GTS derivation* which is given by a series of direct graph transformations. We write $G_0 \xrightarrow{r^*} G_n$ to denote that such a derivation exists.

The way graphs are embedded and changed in transformations define different methodologies of rule application described in the next section. In general, the application of a rule r to a graph G, replaces the instance of a subgraph L in G with subgraph R. The result of applying a rule r to a graph G is as follows [AEH+99]:

1. Choose an occurrence of the left-hand side in G, by morphism.
2. Remove it from G.
3. Embed R into the remained graph.

A total match between the left-side subgraph of a rule and a subgraph in the source graph is made, then the source subgraph is deleted and replaced by the right-side subgraph R.

As explained in the following section, the embedding step may be explicitly defined or it may be implicit depending on the type of rule application used. In some cases, the application of rules is restricted by certain explicit or implicit conditions. These conditions are called negative application conditions (Definition 2.12). The above procedure is a general framework that captures the properties of different approaches in rule application. It is the responsibility of a rule to define what kind of embedding is allowed. Therefore, with different embeddings we have different transformation-rule application and hence, different types of GTSs. Two types of these rules with examples for each are explained in the next section.

There are two main types of GTSs based on two approaches of rule application: *context-free* GTS [Roz97] and *algebraic* GTS [CMR$^+$97, CEH$^+$97]. A rule is context-free if it has only one node or hyperedge (a sequence of nodes connected by edges) on its left-hand side. This is a similar definition to context-free Chomsky grammars for formal languages. Unfortunately, although context-free graph grammars are straightforward like Chomsky grammars, they are not powerful enough to describe visual languages. This is because in the application of rules in these grammars we have to deal with the connection of the replaced graph and the context of a node or hyperedge. It is also true that while context-free grammars may well define the changes in a graph when graph elements are added, they have problems in describing element deletion, because in context-free grammars rules have only one nonterminal graph element on their left-side, and the rule application replaces this element by a graph on the right-side of the rule. Therefore, describing the shrinkage of graphs is not straightforward by these grammars.

On the other hand, algebraic approaches are considered context-sensitive. In these approaches, to apply a rule to a subgraph, the context graph in which the subgraph is located is also important and appears on the left side of the rule.

The Algebraic Approaches to Graph Transformations

The algebraic approach is based on the concept of gluing graphs. The idea is a generalization of concatenation for strings to gluing for graphs. In this approach, the embedding step of the application of a rule ($r : L \rightarrow R$) to a host graph G is done with the help of a third graph that is common to the replaced and the replacing subgraphs. This common subgraph is used for gluing the surrounding graph of the removed instance of subgraph L in G to the right-side graph R. In this approach, graphs are considered as algebras, and embedding or gluing is defined by an algebraic construction (pushout). Two popular algebraic approaches are *double-pushout* (DPO) [CMR+97] and *single-pushout* (SPO) [CEH+97].

Double-pushout Approach

Double-pushout is an algebraic approach in which a derivation step or a transformation rule r is applied in two gluing constructions or pushouts [CMR+97], $r : L \xleftarrow{l_1} K \xrightarrow{r_1} R$, where r_1 and l_1 are morphisms.

Example. As illustrated in Figure 2.6, consider a graph G to which we would like to apply rule r. In rule r, L is the left side graph which we would like to replace with the right side graph R. K is the common interface subgraph between the left and the right side graphs. The first step is to find the match m for the subgraph L in G based on a total morphism. The next step is to delete the image of L from G up to the interface subgraph K (i.e. the elements of G that has a mapping to L, but not to K) and obtain the context graph D. This step is the first pushout (gluing construction) of the double-pushouts which is actually an inverse gluing. The final step or the second pushout is to glue the image of the right side subgraph R to D up to the interface K and obtain the graph H. In DPO, both the matches l_1 and r_1 are total morphisms.

In general, the total morphism m does not have to be an injective morphism, i.e. in morphism m, elements of L are not identified by different elements of G. This situation may cause some problems. For example, in Figure 2.7, a rule is applied to graph G to delete one of its nodes. Since G contains only one node, this node can be mapped to both nodes in the left side of the rule. Thus the transformation rule specifies both the deletion and preservation of the node, which results in a conflict. There are different ways to solve this problem: preservation of the node in G has priority over deletion, deletion of the node in G has priority over preservation, or the application of the rule in this situation is forbidden. In DPO, the third solution is chosen.

Another problem in the application of a transformation rule is when a node is deleted and it has some connected edges that are not part of the match for the deletion. Again,

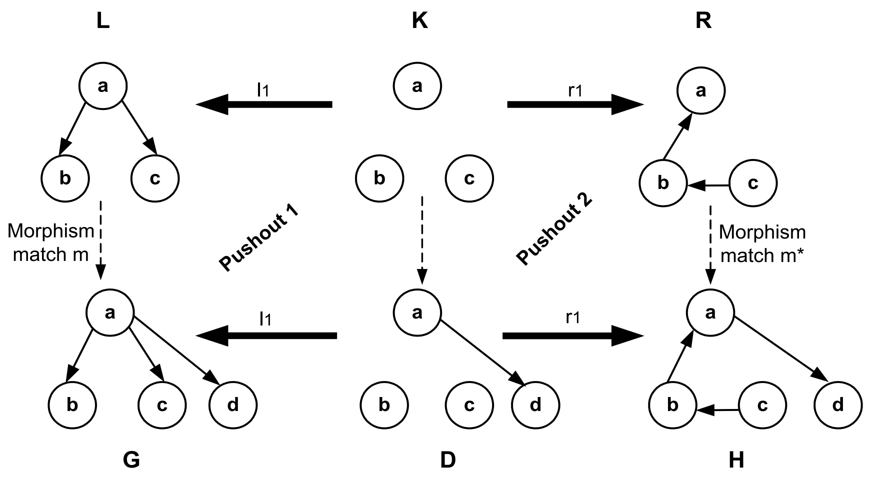


Figure 2.6: Applying rule r to graph G and deriving H from G in DPO

there are two solutions for this problem: deletion of all these edges (which are called dangling edges), or forbidding the application of the transformation rule. In DPO, to ensure that these problematic situations do not occur, the morphism m must satisfy two conditions, which are called *gluing conditions* [CMR+97]:

1. *The dangling condition* requires that if a rule is going to delete a node, it must specify also the deletion of all edges incident to that node.

2. *The identification condition* requires that any node or edge that is being deleted from G by the application of r has only one match in L. This avoids situations such as those encountered in Figure 2.7.

Single-pushout Approach

Definition 2.10 (SPO [Roz97])**.** *In SPO approach, transformation rules are presented as* $r : L \xrightarrow{r_1} R$*, where* r_1 *is a nonempty partial morphism. A matching of a rule* r_1 *with the graph G is a total morphism* $m : L \to G$*. Applying an SPO rule* r *to a graph* G *is done by checking the morphisms* m *and* r_1 *to yield a target graph* H*.*

Example. As illustrated in Figure 2.8, in SPO there is only one gluing step. In this figure, first, a total match m between the graph on the left side of the transformation rule

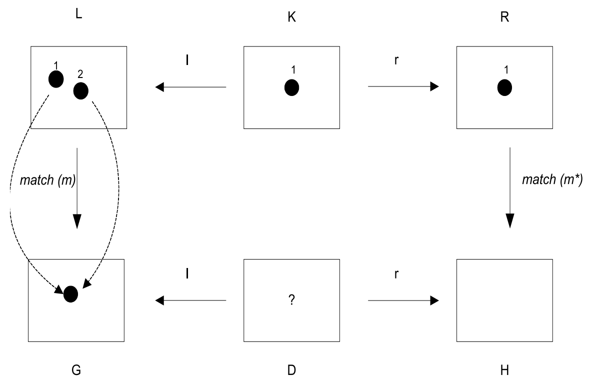


Figure 2.7: Direct derivation in double-pushout

and a subgraph in G is made, and then the subgraph of G is deleted and replaced by the right side graph R, resulting in H. Specifically, everything in L but not in R will be *deleted*, everything in R which is not in L will be *created*, and everything that is in both sides will be *preserved* [CEH+97].

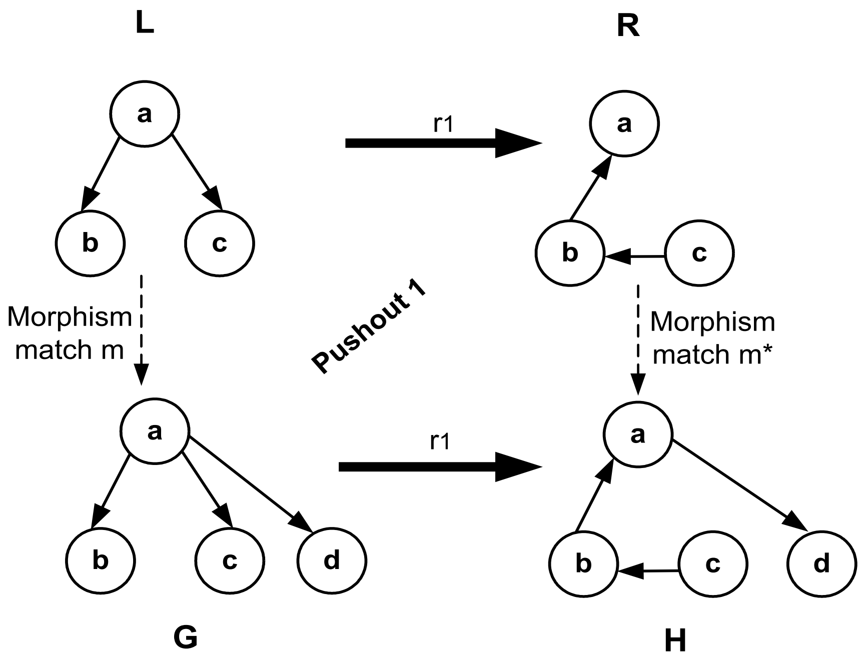


Figure 2.8: Applying rule P to graph G and deriving H from G in SPO.

In comparison to DPO, there is no explicit gluing condition for transformation rules in SPO. For dangling edges, each time a node is deleted, implicitly all incident edges will be deleted as well. To solve the identification problem of Figure 2.7 in DPO, deletion of nodes and edges has priority over preservation in SPO. This solution has been illustrated in Figure 2.9. Because of the priority of deletion over preservation, it can easily be observed

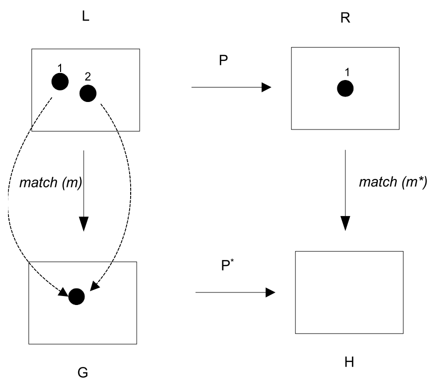


Figure 2.9: Direct derivation in single-pushout

in Figure 2.9 that the morphism m is a total mapping from L to its image in G, but the morphism m^* is a partial mapping from R to the image of R in H. The reason is that elements of R that should have been preserved have been deleted because of the conflict. In other words, there might be elements in R that do not have an image in H.

Because the DPO approach is restricted by those explicit gluing conditions for rules, we gain more expressiveness by using SPO transformation rules. In other words, SPO direct derivations are complete because if there is a match for a transformation rule, there is always a corresponding direct derivation. But for DPO, direct derivations are not complete because even though there might be a match for the application of a rule, it still may not be applicable because of the gluing conditions. However, since it is always possible to restrict the application of SPO rules by some conditions, DPO can be considered as a special case of SPO with gluing conditions as the application conditions.

2.2.4 Tool and Notations

For automatic application of the transformation rules and generation of graphs based on the rules, several tools have been developed [EEKR99]. In this section, we provide some background on the tool that is used in this thesis.

We have conducted our experiments using the GROOVE graph transformation tool [Ren04b]. GROOVE is an open-source software easily accessible and regularly updated with new features. Unlike some other GTS tools that use textual descriptions for transformation rules, GROOVE provides environments to directly define transformation rules using visual notations. It was first developed for software model checking of object-oriented systems. The tool supports the SPO approach and as an attempt for verification purposes creates the full state space of a graph grammar for a finite-state system.

Notation

GROOVE supports directed graphs with multi-labelled edges and nodes, and attributed nodes. In this tool, the distinction between the left and the right side of the rule has been built by different shapes and colors of graph elements. GROOVE uses boxes as graph nodes and arrows as edges between nodes, and attribute are presented as circle or oval nodes. For example, Figure 2.10 illustrates a GROOVE rule in SPO. In this rule, the thin solid elements (black in a colored print-out) are those elements that are preserved in both the left and right sides of the rule. Hence, at first the rule checks if these elements are present in a graph that the rule will be applied to. The thin dashed elements (blue in a colored print-out) are those elements that should be deleted, so they are part of the left side, but not the right side of the rule. The solid fat gray elements (green in a colored print-out) are those elements that are being created, so they are part of the right side, but not the left side of the rule.

For the application of the rule in Figure 2.10 to a graph G, we must first check if there is a subgraph that has node elements A, B, and C, if there is an edge labelled b between A and B, and also if there is an edge labelled c between A and C. Then the rule deletes the edge A-C along with the node C plus the edge A-B, and creates an edge B-A with the label a, a node D, and the edge B-D labelled as d.

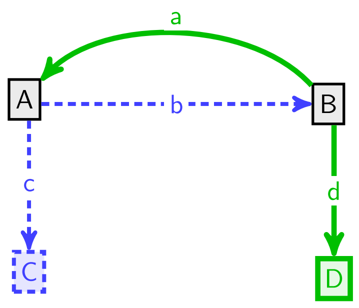


Figure 2.10: A simple GROOVE transformation rule.

Negative Application Condition (NAC)

To apply a rule to a given graph, a matching of the left side of the rule and the given graph is found. This type of matching expresses one type of the conditions that can be expressed by rules [Ren04c]. For example, we cannot express any kind of negative conditions such as the non-existence of an edge. Therefore, in the context of algebraic transformation rules, *negative application conditions* (NACs) are introduced.

Definition 2.11 (NAC [Roz97])**.** *A negative application condition or NAC(n) on L is a graph morphism $n : L \rightarrow N$. A graph morphism $m : L \rightarrow G$ satisfies a negative application condition n iff $\nexists\ q : N \rightarrow G$, such that $q\ o\ n = m$.*

$$\begin{tikzcd} L \arrow[r, "n"] \arrow[d, "m"'] & N \arrow[dl, "q", "/" marking] \\ G & \end{tikzcd}$$

Figure 2.11: Negative Application Condition (NAC)

A rule with a NAC is interpreted as a logical constraint. These constraints are represented in a graphical way and express conditions like non-existence of certain nodes and edges. NACs are applied based on the matching found between the left side of a rule and a given graph, but with the consideration that if NAC matchings exist, the rule is not applicable. In GROOVE, NACs are depicted as dashed fat gray elements (red in a colored print-out). For example, the rule in Figure 2.10 has been changed to the rule in Figure 2.12 with NACs. This rule applies as before with the constraint that there does not exist

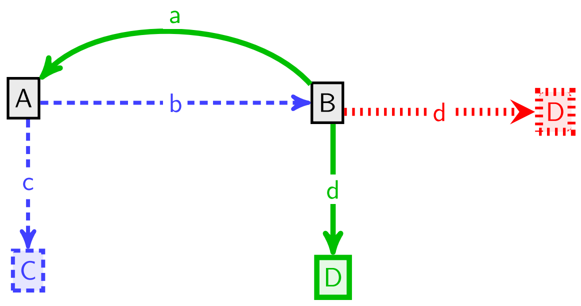


Figure 2.12: A GROOVE transformation rule with a NAC.

already a node D and an edge labelled d between B and D.

Chapter 3

Formal modelling using GTS

3.1 Introduction

Currently, there is intense pressure to rapidly migrate complex communication protocols to the Internet. A communication system evolves dynamically with the addition and deletion of service components. This evolution changes the topology of the system, and may cause violation of inter-component specifications, thereby causing undesirable behaviour. In this context, systems are particularly vulnerable to problems, and an accurate yet usable formal method of describing and analyzing these systems is vitally important [dBG05].

In this chapter, we propose using visual semantics [EH00] to describe the behaviour of dynamically evolving communication protocols as a first step toward such a formal analysis. We show that graph transformations provide a natural and expressive formalism for describing such semantics and we illustrate its use by giving a visual, graph-based semantics to an Internet-based communication protocol. Our graphical description of the semantics has several important advantages, most notably the ability to cleanly separate out those system features of current interest. This separation of concerns is a necessity for formal analysis of system behaviour (*cf.* [AENT03]).

We propose an approach that later allows the designer to formalize the behaviour of each designated component individually. The approach uncouples those components of interest from those that are not currently interacting with them. Our model is well-suited for describing the behaviour of dynamic component creation and elimination. Thus, the structure of the generated model closely resembles the way in which communication protocols are typically described. The graph transformation semantics follows this description, enabling a clean separation of concerns when describing a protocol. Our focus in this chapter is on the dynamic evolution of protocols. Ideas for formalizing the components and interactions are given in the future work section (Section 6.2)

Our approach has two other notable advantages. First, it allows one to give a visual description of system semantics. Second, it allows a designer to describe computations of communication protocols at an appropriately abstract level. In general, the abstraction of code is a difficult task, but abstraction appears more naturally in design artifacts. Therefore, this thesis describes abstraction and model-checking techniques in design levels such as topology changes. In the work presented in this chapter, we model topology changes and global-state changes that are due to the addition and deletion of components. The idea is to use single-pushout (SPO) transformation rules to formally define the communication protocol's dynamic transformations. As a motivating example, we have done a case study and provided the graph-based semantics for significant aspects of AT&T's next-generation IP-telephony architecture.

The remainder of this chapter is organized as follows: Section 3.2 explains an overview of the industrial case that we have chosen to apply our GTS formalism. In Section 3.3, our GTS formalism is presented, and the case study for evaluation of our proposed formalism is described in Section 3.4. We present the related work in Section 3.5 and conclude in Section 3.6 with a summary of our contributions in this chapter.

3.2 Distributed Feature Composition Protocol

Distributed Feature Composition (DFC) is an architecture that AT&T has developed and used as the basis for BoxOs [BCP$^+$04], its next-generation IP-telecommunications protocol. This architecture was introduced by Jackson and Zave in 1998 [JZ98].

In the following section, we provide a brief description for DFC. Later, we use this description to present a formal model for some functionalities of the DFC. We have simplified some of the concepts for the sake of our case study. For example, the decision about how the features are assembled into a call is highly dependent on the interactions between features, but we have made some assumptions about an ordering between features.

3.2.1 Basic Semantics of DFC Architecture

The goal of DFC is to increase component modularity and to structure the way in which components interact. In the most straightforward situation, the DFC protocol provides stand-alone functionalities such as basic phone service. *Features* are used to add incremental functionalities to existing services. Each feature is a component process in the communication system. Examples of features in DFC include Call Forwarding on No Answer, which allows incoming calls to be redirected to another address based on a trigger from the callee; and Call Waiting, which provides to a subscriber the ability to switch to a second incoming call.

Usages

In DFC, a request for telecommunication service is satisfied by a *usage*, which is a dynamically assembled graph of *boxes* and *internal calls.* A *box* is a concurrent process providing either interface functions (an *interface box*) or feature functions (a *feature box*). Each interface box has an *address*, and each feature box has a *box type* that corresponds to the feature that it implements as well as an address. An *internal call* is a featureless, point-to-point connection with a two-way signaling channel and any number of media channels. We use the phrase "internal call" to emphasize that it is not the same as what a user might think of as a "call".

Here we briefly define some basic terminologies used in DFC:

Definition 3.1 (Box Type)**.** *The type of a feature box, corresponding to the feature that it implements.*

Definition 3.2 (Address)**.** *A string used to identify a telecommunication device attached to a network.*

Note that each interface box has an address. Each feature box has a box type and an address, and the feature box is created (instantiated from the box type) on behalf of the address.

Definition 3.3 (Source Subscriptions)**.** *For each address, the sequence of box types that should be assembled into a usage on behalf of an address when the address is the source of a call request.*

An example of a source feature is the Teen-Line feature which restricts outgoing calls at certain times of the day. Source features act transparently if the subscriber is the recipient of the call.

Definition 3.4 (Target Subscriptions)**.** *For each address, the sequence of box types that should be assembled into a usage on behalf of an address when the address is the target of a call request.*

An example of a target feature is the Call Forwarding on No Answer. Target features act transparently if the subscriber is the initiator of the call. Some features behave as a source-subscribed feature in one portion of a call and as a target-subscribed feature in another portion of the call. These features are called *Source/Target* features and allow the subscriber to be involved in multiple calls simultaneously; an example of this is Call Waiting.

A usage describes a connection between two telephones. The party initiating a connection (*caller*) and the party accepting the connection (*callee*) may both subscribe to several features. A usage grows and shrinks dynamically starting at a single initiating phone, then features of the initiator party are added to the usage. Eventually, features of the target telephone are added to the usage, and then the target telephone is added to it. Because a device may be both a caller and a callee, and this matter is decided during a call, all of a party's features must be included in every call, rather than just the callers source features and the callees target features. An example of this situation is when an initiator device A calls B and A is subscribed into some features as source, if another telephone joins the usage via A's Call Waiting feature and reaches A as a callee, then all the A's source features act transparently in the call from the new telephone toward A. So, source features act transparently if the subscriber is the recipient of the call.

There is a partial precedence order among the source-subscribed (target-subscribed) box types of an address. This order is an input to the routing algorithm and is chosen by the designer based on feature priorities to eliminate undesirable feature interactions. Features are not intended to stand alone when they appear in a usage (i.e. Voice-Mail or Call Waiting do not operate without a basic phone service having two communicating phones). A usage is presented visually as a graph. An example of a usage is illustrated in Figure 3.1. This figure shows a straightforward usage formed when device X, which

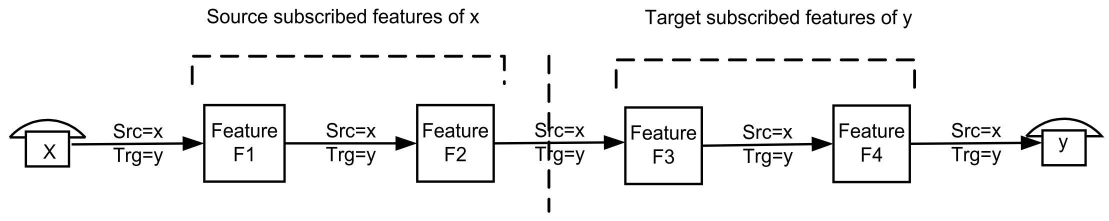


Figure 3.1: A DFC usage with rectangles showing feature boxes and arrows showing the direction of the call request.

has subscribed to features F_1 and F_2, requests a connection to end party Y, which has subscribed to the feature F_3 and F_4.

One characteristic of a feature is its boundedness. A feature is *free* if new instances of that feature are created whenever the feature appears in a usage. When a telephone with instances of its features is involved in a usage, if a second usage requires that telephone, then new instances of the telephone's free features will be created. In contrast, a *bound* feature has only one instantiation, and all calls involving the subscriber's telephone are routed through this instance. Bound features are subscribed both as source and target features. An example of a bound feature is Call Waiting and an example of a free feature is Call Forwarding.

The fundamental idea of DFC is pipe-and-filter modularity [SG96]. Each feature box behaves transparently when its functions are not needed. Each feature box has the autonomy to carry out its functions when they are needed; it can place, receive, or tear down internal calls; it can generate, absorb, or propagate signals travelling on the signaling channels of the internal calls; and it can process or transmit media streams travelling on the media channels of the internal calls. A feature box interacts with other feature boxes mainly through its internal calls, yet it does not know what types of boxes are at the far ends of its internal calls. Thus, each feature box is largely context-independent; the feature boxes that make up a usage can easily be added, deleted, and changed.

Routing

An end party's request for a connection is handled by the *DFC router* [JZ98]. Each internal call is set up with the help of the router, which decides which box to route the request to. Each DFC router implements an algorithm that incrementally constructs a usage and does not rely on stored information about the state of any usage, but rather on static data such as which addresses subscribe to which features. All of the necessary information about the usage state are carried along in the setup signals (the request to establish a call) of the internal calls.

To make a simple phone call with one telephone at each end and some features in between, the router may need to make several internal calls. Figure 3.2 shows that a caller initiates a call by sending a setup signal to the router through the *box_out* channel. Following that, the router sends the setup signal to the other features. If all features agree to setup, they acknowledge that by sending back the *upack* signal to the caller. The caller communicates its signal messages with the other features downstream through the signal channel *ch*.

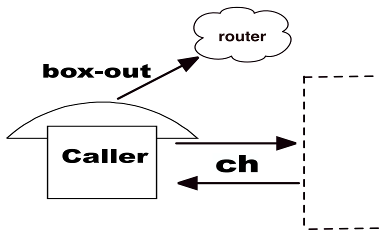


Figure 3.2: Caller is communicating via the channel **ch** to a component at its right, sending its setup signal to the router through **box-out** channel.

This routing algorithm has three basic methods: *new*, *continue*, and *reverse* that are executed on behalf of interface or feature boxes. The method *new* is executed by an interface box to request a connection and creates a setup signal. The method *continue* is applied by a feature box to a received setup signal and results in the resubmission of that signal to the next box closer to the desired end party. The method *reverse* is performed by a feature box to reverse its incoming call, i.e., change the direction of a call, possibly back towards the initiating interface box. Those features that can perform the reverse action are called *reversible* features.

Definition 3.5 (Call Path)**.** *A path through a usage, both of whose endpoints are interface boxes. Two internal calls touching the same box form a segment of a path if and only if they are linked via the box and they are linked via the box if and only if the setup of one was derived from the setup of the other by use of continue or reverse signals. The point on the full path of the usage in which the call continues routing to target-subscribed feature boxes, is called mid-point of the call.*

For routing, the router chooses feature boxes in sequence from a list, called *route*, associated with the current address. The route consists of a sequence of features of the source address and a sequence of features of the target address. If feature boxes use only the *continue* method, then the simplest usage as depicted in Figure 3.1 has instances of all source-subscribed box types of the source address, followed by all target-subscribed box types of the target address.

3.3 Visual Semantics Using GTS

We propose an operational semantics to describe the evolving behaviour of communication protocols using a graph model. The ultimate goal of the work is a 3-level model. This section defines our minimum requirements of the three levels, two of which are established formalisms. But the contribution of the chapter is the third level and the application of GTS to model dynamic hierarchical graph systems.

In the literature [Roz97, BH02], several types of graphs have been defined that are suitable for different system structures and models. Among them we use both hierarchical graphs, where a node may contain a subgraph, and attributed graphs, where nodes and/or edges are labelled with attributes.

At the first level, the functionality of a component is shown as a finite-state machine graph, with each machine describing the behaviour of an individual process. A process may be composed by several modules. This modularity may result in decomposition into additional components, but for the sake of clarity and our interest in component interaction, we consider each process as a single component.

The second level shows a composition of components communicating through channels via internal calls. This composition is shown as a higher level graph. This level represents a Communicating Finite-State Machine (CFSM) architecture [Pac03, BZ83].

The third level shows changes to the global state of the system. The global state of the system may be modified due to a local change of state in any of the components or via a topological change. For example, topology changes show how a component may be added to a communication network or how a component may depart from it. Each node of the transition graph in this level is a graph modelled in level two, and it is a reachable state in the transition graph. In this level, transformation rules result in the transitions between states of this graph. In summary, these are the three levels:

1. Components are modelled by FSMs.

2. Communication is modelled by CFSM.

3. Evolution of the component set is modelled by a GTS.

Having formal semantics to define 3 levels, the main focus of this thesis is on level three which provides semantics for dynamic evolution. In Section 6.2, we have described the way the first-level graphs can be modeled and also showed examples on how to integrate this level and the graphs in the second level.

The graph model at the first level simply uses the standard notation for a Finite-State Machine (FSM) [HU79] graph with the following definition:

Definition 3.6 (Finite-State Machine (Mealy Machine)). *A finite-state machine is a tuple* $(\Sigma, \Delta, Q, q^0, \delta, \lambda)$*, where* Σ *is a nonempty finite set of input symbols;* Δ *is a nonempty finite set of output symbols;* Q *is a finite set of states;* $q^0 \in Q$ *is the initial state;* $\delta : Q \times \Sigma \to Q$ *is the transition relation mapping a state and an input symbol to another state; and* λ :

$Q \times \Sigma \rightarrow \Delta$ is the transition relation mapping a state and an input symbol to an output symbol associated with the transition.

A state of an FSM is a graph node and state transitions are directed edges with suitable labelling for both states and transitions. An example of an FSM graph has been illustrated in Figure 3.3. The FSM in Figure 3.3 describes part of the caller box process [Dom05].

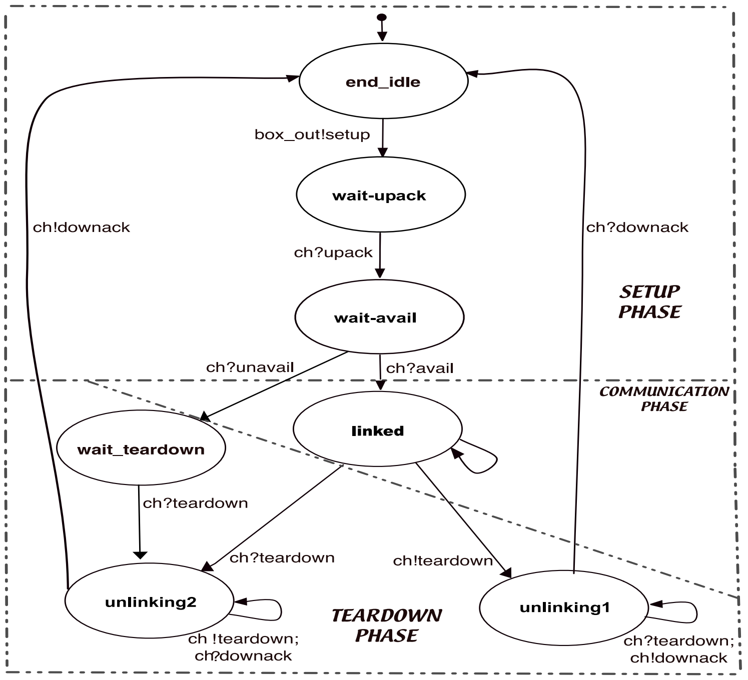


Figure 3.3: Caller Process Finite-State Machine. !: signal transmission, ?: signal reception

After sending the setup, the caller waits for the reception of *upack* (acknowledgement signal that the setup has been received by downstream boxes) and then *avail* or *unavail* signals through the *ch* channel. The communication holds and the line links until a *teardown* is demanded from either party.

For the second level, based on Brand and Zafiropulo [BZ83] we define a CFSM protocol as:

Definition 3.7 (Communicating Finite-State Machine)**.** *Let $I = \{1, 2, 3, ..., n\}$ be a finite set of integers with $n \geq 2$. A CFSM protocol is defined as a pair (P, L), where*

- *P shows a set of n processes and P_i is the i-th process from the set where $i \in I$.*
- *$L \subseteq I \times I$ is a relation identifying the nonempty set of error-free channels C_{ij} such that $(i, j) \in L$. Each channel C_{ij} is a perfect, unbounded FIFO queue that links two processes P_i and P_j.*

A CFSM protocol can be viewed as a directed labelled graph described in Definition 2.6. Nodes of this graph are processes which are defined by FSMs and edges are channels. This graph defines the topology of a communication system. In this communication system, we assume the existence of essentially error-free and unbounded channels over which processes communicate. An example of a CFSM is depicted in Figure 3.4.

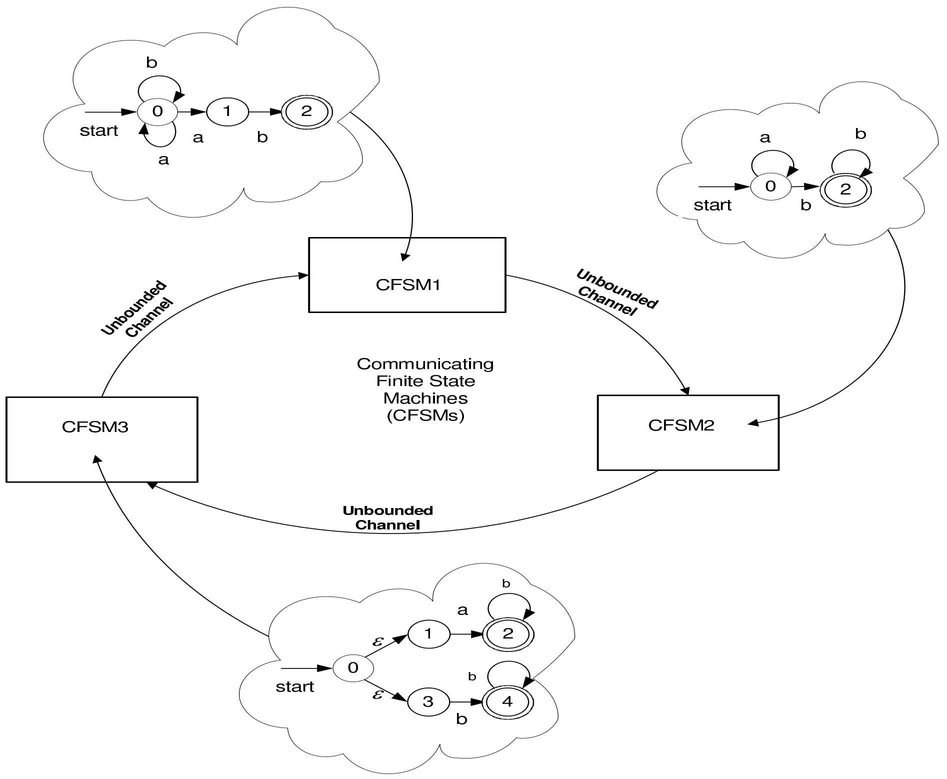


Figure 3.4: Graph of Communicating Finite-State Machines.

For the third level, to describe how the states of a system defined as graphs transform as the transformation rules are applied to them repeatedly starting from the initial state, we give the definition of a graph transition system [Roz97, HPR06], $\mathcal{G} = \langle S, T, I \rangle$. We write G_s to denote the graph of state s.

Definition 3.8 (Graph Transition System)**.** *A graph transition system is defined as: $\mathcal{G} = \langle S, T, I \rangle$, such that:*

1. *S is a set of states, where each state has a graph structure.*
2. *T is a set of transitions : $T \subseteq S \times P \times S$ where P is a set of transformation rules and for all $t \in T$, $t : s_1 \xrightarrow{r} s_2$, there is a graph transformation rule $r \in P$ that transforms G_{s_1} to G_{s_2}.*
3. *I is a set of possible initial state graphs.*

To model the dynamic behaviour of a communication system, we utilize the graph transition system in which nodes represent states of the system at a particular point in time, and transitions show how the system evolves from one state to the next. System evolution, or computation, is thus expressed as sequences of transitions beginning from a source or initial state. Each state of the system is modelled as a graph, and by using the graph transformation system we describe how a system changes from one state to another.

The global operational semantics is a GTS generated by a set of transformation rules for topological changes and operational changes components. The local behaviour of each component can also be specified by a set of local transformation rules. Attributed graphs help to describe the specification of local operational data for each component. The application of a local transformation rule to a state graph does not add or remove any component; therefore, there would be no change in the structure of the state graph.

Note that we do not provide a formal relation between the various level descriptions of a system in this chapter. Since the focus of the thesis is on analysis of changes, our work concentrates on level three. Also, current GTS tools do not support FSM and hierarchical modelling. We propose ways of modelling the finite-state machines in GTS and the relationship between the first and the second level in Section 6.2. Therefore, in this chapter we presented these three levels as an ultimate goal for describing different levels of dynamic systems, and the exact semantic relationship between these levels is left as future work.

The main challenge for the first level is modelling synchronization of the computations of the FSM graphs and also modelling events in these graphs. Possibly, one way would be specifying a synchronization relation between components of a distributed system modelled as a GTS. This may be done by distinguishing a set of local transformation rules for each component, then defining a relation between transformation rules of two components that are going to interact. This relation can be defined based on the existence of common ports in two sets of rules. So, those rules that act on common ports can be synchronized.

In the next section, we present our case study and the GTS rules for this case for the third-level modelling.

3.4 Evaluation

To the best of our knowledge, there is no extant formal model for DFC. Although DFC semantics can be extracted by naively building a single giant finite-state machine (FSM) together with queues as communication channels, this results in state-explosion and does not give a dynamic approach to describe different functionalities of the system. In this section, we evaluate our GTS formalism (the second-level and third-level) by applying it to part of DFC as described in the case study section.

3.4.1 Case Study

In this section, we provide a case study based on DFC that presents an evaluation of our GTS formalism. DFC is an architecture that is being implemented and currently used in AT&T for description of IP-based telecommunication. We have chosen some functionalities of the DFC to be used in our case study for formal modelling and later for verification. Specifically we provide the modelling for:

1. the basic connection in DFC.
2. the dynamic growth of a usage assembled using generic features.
3. the dynamic shrinking of a usage due to the activation of FindMe feature.

4. the dynamic shrinking of a usage when one party participating in a call hangs up.

5. free features.

6. reversible features (in the next chapter).

7. Call Waiting (CW) feature defined as: *A feature that allows a subscriber to receive an incoming call while already engaged in a usage. The subscriber can switch between the new call and the existing call. CW is provided by a bound box and is both a source and a target feature, because the subscriber may be either the caller or the callee in the existing usage.*

8. FindMe feature defined as: *If the call fails to be routed to the target device, e.g. the callee is not available, this feature can be activated to find another device to replace the target. The change of target modifies the routing to a new device with different features.*

At the end of this chapter, the set of graph transformation rules that model the above functionalities has been given.

Figure 3.5 shows a usage with a telephone that subscribes to a Call Waiting feature connecting to a cell phone which subscribes to a Call Waiting feature. The black telephone tries to reach the cell phone and since the Call Waiting feature is a bound box, the second usage joins this feature. The graph of this usage has been built in the GROOVE graph

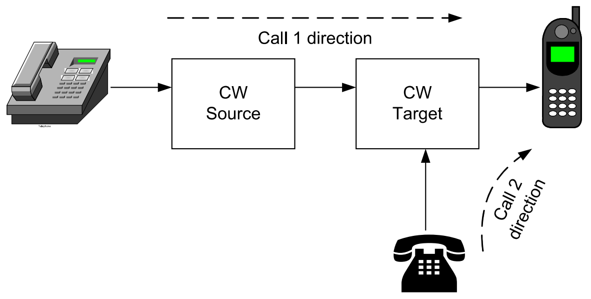


Figure 3.5: Telephony usage for CW features as Src and Trg features.

transformation tool [Ren03] and illustrated in Figure 3.6, which shows a second-level CFSM

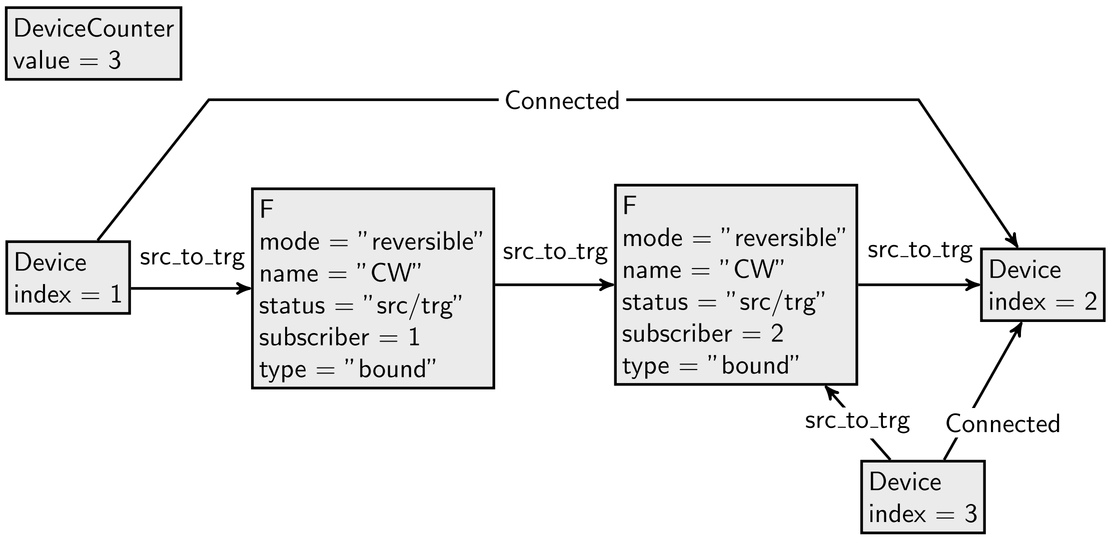


Figure 3.6: The graph for Call Waiting usage.

graph with nodes representing telephones and features. This graph is itself one node of the transition graph at level three. The example in Figure 3.7 shows two nodes of the transition graph at the third level. This is one computation step with a topology change developed by the application of the rule *JoinCallee*, depicted in Figure 3.8, joining the call usage to the callee. The partial usage from the caller towards the callee has evolved in several steps. We omit to show these steps, except the final one that has been illustrated in Figure 3.7.

The set of transformation rules for creating the state graph of Figure 3.6 is illustrated in Figure 3.8. The rules presented in this figure are SPO production rules. Each feature node is labelled with a set of attributes that represent the local operational data of the feature such as the name of the feature; status, which shows if it is a source subscription or target subscription or both; and mode, which shows whether a feature is reversible. In building a usage graph, we insert all the subscribed features of an end party in the order of features subscribed by the source device, and features subscribed by the target device (*cf.* [JZ98]). In addition, there is a type attribute that accepts the values bound and free, and the attribute subscriber that shows who the subscriber of the feature is, the caller or the callee. Most of these rules describe the topological changes of the system.

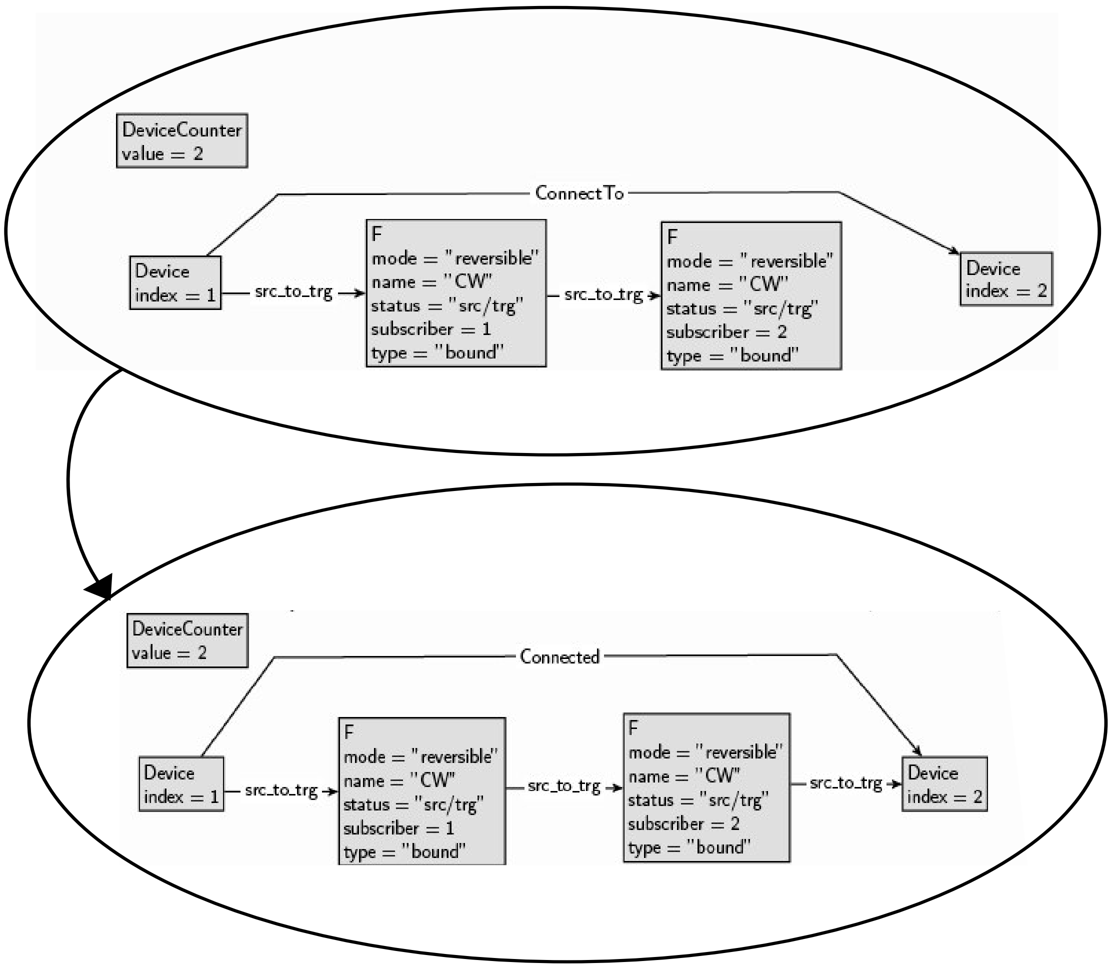


Figure 3.7: A Graph derivation step representing the third level of computation by the application of the transformation rule *JoinCallee* adding an edge between target CW feature and the callee.

The rule *InsertFeature-Src* adds a source-subscribed feature to a usage. In fact, this rule adds the first feature of a caller, and models the *new* method of the routing algorithm in DFC. This rule has a constraint or negative application condition (NAC) that a feature must not exist previously. Rule *AppendFeatureChangeZone* shows that there is a path to the last feature in the source zone, and a new feature that has been subscribed to the callee is going to be added as a target feature. In GROOVE, rules use variables (?x or ?y) as edge and node labels to match with an arbitrary label. We are also allowed to use regular expressions to enhance our rules. For instance, the *AppendFeatureChangeZone* rule expresses that several features are connected through the path connecting the caller and callee. There are a couple of conditions to be checked for this rule: first there is a NAC which ensures that the zone has not been changed before. NAC checks that there is not already a feature box after several addition of source feature boxes. This condition mainly avoids parallel growth usage after the addition of source features. Second, we must ensure that the existing feature is not already subscribed to the callee and hence is not a target feature. The rule *NameFeatureCW* changes the attributes of a feature and converts the feature to a Call Waiting one.

Many processes may exist in a distributed IP-based telephony protocol, but each node of the transition graph is a partial usage that can be analyzed and verified separately, without dealing with the processes not directly involved in the same usage.

3.5 Related Work

Software architectures provide a basis for modelling a software system based on its components, and for describing interconnections of components [Mag95, IW95, SG96]. One of the concerns about architectures is whether they provide a high-level description of components and their connections in terms of graphs. To address this concern, Architecture Description Languages (ADLs) [GP94, SG96, Mag95] were introduced. These languages are used to describe the architecture of a system in terms of components, connectors, and interfaces. Another challenge about software architectures is to ensure that the architecture is compliant with the system specification. In the literature, this challenge is addressed by verifying that the design and composition of a system conforms to a set of architectural constraints, called "architectural style", that restricts the relationships of elements in an architecture.

ADLs have traditionally been used to describe system configurations, but could be improved upon in the domain of capturing and analyzing dynamically evolving and critical software systems. In order to verify correct behaviour of critical components, system configuration must be enriched with behavioural descriptions of the system. For example, there has been work on attaching behavioral specifications to software architecture descriptions

[GP94, Gia99]. Associating behaviour to structure in this fashion is important for understanding how the architectural changes (and related behavioural changes) are consistent and also how they can be synchronized in critical software systems. The above methods mainly focus on specification of interactions among components using a process algebra notation (CSP or FSP). For verification of a behavior, they need to employ recursively the behavior specification of all the components. However, in these methods verification of a one-step architectural change is problematic, because each modification in the architecture may be related to many behavioural changes in the components. Thus, a methodology to manage the evolution through incremental transformation steps, in different layers, is crucial. Our GTS formalism provide this methodology.

Unfortunately, ADLs and architectural styles do not have necessarily a formally defined semantics, and do not support reasoning about the dynamic evolution of a system. This problem can be addressed by using GTSs [Mét98]. GTSs can be seen as a formal basis for extending existing ADLs, by describing the computation of the system and providing a derivation tree that enables analysis of each partially-built configuration of the system. In addition, transformation rules and constraints encoded as graphs, control a well-behaved composition of the components and conformance to style architectures. The modelling provided in this thesis presents a way of reasoning about formal properties of the architecture evolution. An interesting future work would be to use our modelling to verify that each evolution preserves constraints of architectural styles.

To our knowledge, this thesis is the first work to use graph transformation machinery to model the details of the dynamic behaviour of a communication protocol. In fact, Heckel [Hec98] and Taentzer [TKFV99] have both explicitly noted the omission of reactive protocols from the GTS literature. Among other work that detail system semantics we note the "Abstract State Machines (ASM)," [BS03] or "Evolving Algebras," [Gur04]. ASMs present states of a system as algebras and transitions as evolution of the algebras. ASMs are powerful enough to represent step-by-step system semantics. AsmL is an associated programming language for building ASM models. The work presented in this chapter uses a visual ASM style to capture the semantics of distributed reactive protocols.

Other works such as [Kus01, HM00], illustrate how graph transformations are applied to define the dynamic semantics of systems using UML state machines. These systems present operational semantics for local systems, but do not treat the communicating state machines. The grammar rules of these models are context-free and local, with the restriction of that only one component and its neighbourhood can be affected by a transformation rule. Furthermore, context-free grammars may well define the evolution of a system as components are added, but when components are deleted and the graph is shrinking we need to replace a subgraph with a smaller subgraph. In contrast, our proposed model uses context-sensitive graph transformation rules to cover distributed systems semantics; we note that the added power of context-sensitivity seems to be a requirement to deal with models as rich as DFC.

Similar to GTSs are context-free network grammars [SG90] in which terminals are basic processes, and they associate with each process type a program with a set of ports, and a set of rules to combine processes of these types to admissible networks. However, unlike network grammars that are context-free, our GTSs are context-sensitive grammars that are more expressive, because these grammars can describe more complicated structures such as network grids and tori.

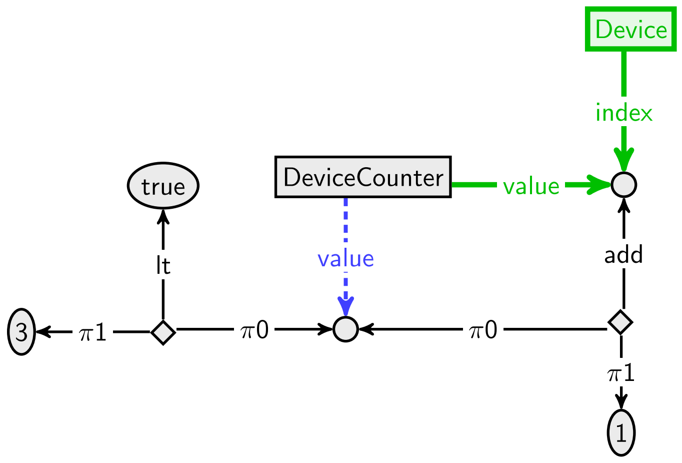
3.6 Conclusion

Graph grammars have primarily been used in areas like pattern recognition, compiler construction, and data-type specification. More recently, their areas of application have been broadened to include concurrent system modelling, software specification and development, database and VLSI.

In this chapter, graph transformation rules have been used to explain the dynamic evolution of a distributed communication protocol. Our work produces a visualization of behaviour in three different levels. We have explicated the semantics of the third level of our model on an IP-based telephony system called Distributed Feature Composition (DFC) using graph transformations. A description of DFC semantics has been presented by a graph transformation system with a hierarchical, attributed graph model and an SPO approach for the transformation rules.

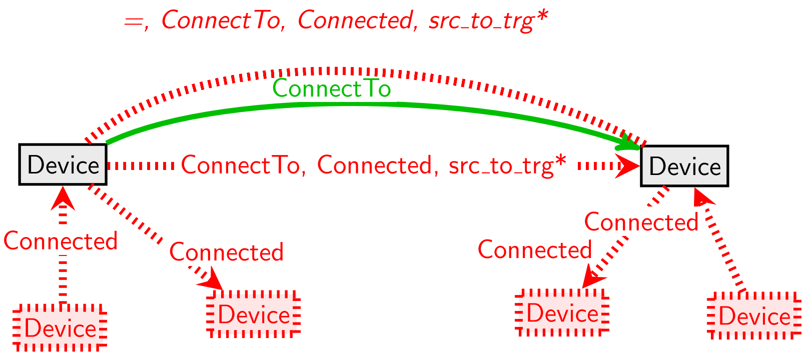
Our model describes the semantics of a telephone or a feature process as a graph, and then details a graph model of communicating features. Then the model describes dynamic evolution of the telephony system via a graph transformation system. At this level we generate a transition graph with nodes representing graphs and edges representing transformations of these graphs. The transition graph provides the ability to focus on a partial connection such as a usage in DFC, without the inclusion of other distributed processes that are not involved in the call. Therefore, each partial usage can be analyzed and verified separately. This is a key advantage over other models because of its visual presentation and ease of use. Another advantage of our model is that it cleanly addresses typical communication protocol layers with the ability to focus on the dynamic evolution of these systems.

insertDevice



Adds a new party (telephone) to a usage, and keeps track of the number of devices added using the DeviceCounter. The diamond in the picture is used for both checking attribute values (e.g if the number of devices is less than or equal to three) and for operation on attribute values (e.g. addition). This rule corresponds to the new signal in DFC routing.

InitiateACall

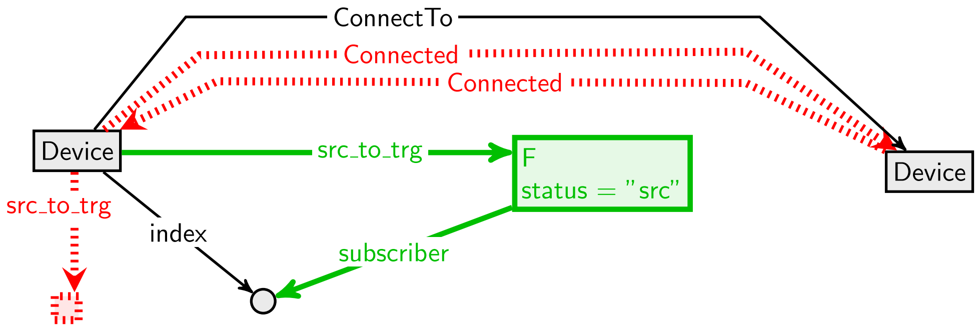


Establishes a new usage. At first it checks if the two devices are not already connected or if there has not been an initiation to connect them and they are not connected to any other device.

Figure 3.8: GTS rules for building a usage and its dynamic growth.

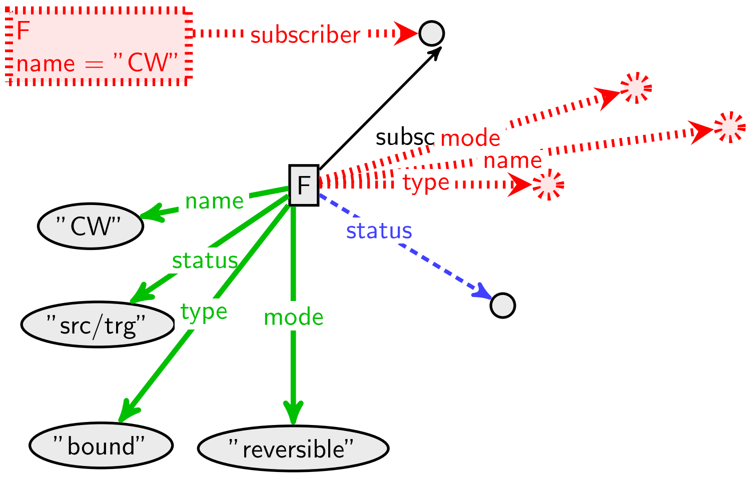
GTS rules for building a usage and its dynamic growth continued ...

InsertFeature-Src



Assembles a new feature into the usage. For the two devices that would like to connect, a new source feature is added to the usage. It is checked that the two devices do not have an established connection and that a feature has not been assembled into the usage before.

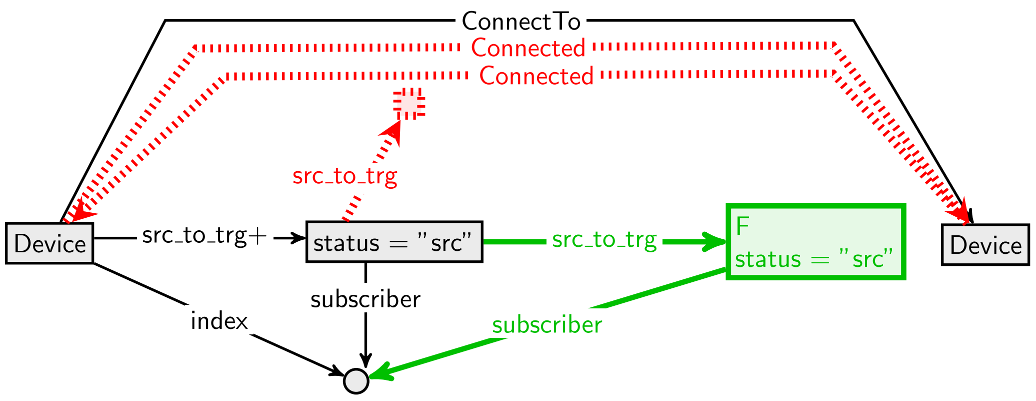
NameFeatureCW



A generic feature is distinguished as a Call Waiting feature. The rule checks if another feature with the same name and the same subscriber does not exist already.

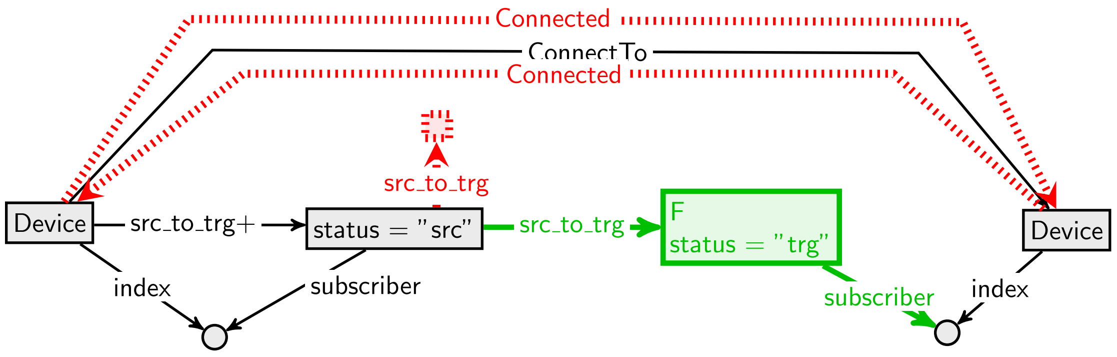
GTS rules for building a usage and its dynamic growth continued ...

AppendFeature-Src



Continues a usage by assembling a new source feature into it. This rule corresponds to the continue signal in DFC routing. The rule checks if assembling the feature is done at the right place after the last one assembled before, and not in parallel to a previously assembled one.

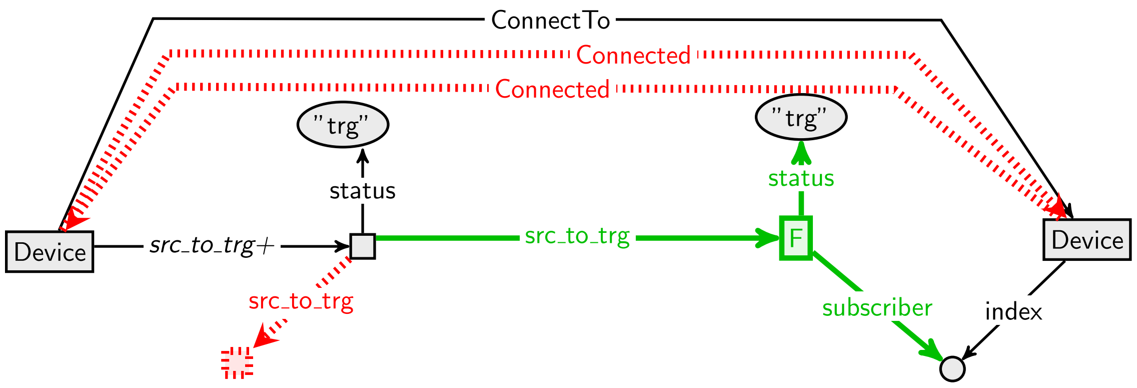
AppendFeatureChangeZone



The usage is continued by assembling the first feature subscribed by the target of the call. The rule checks if assembling the feature is done along a path after the last source feature assembled before, and not in parallel to a previously assembled one.

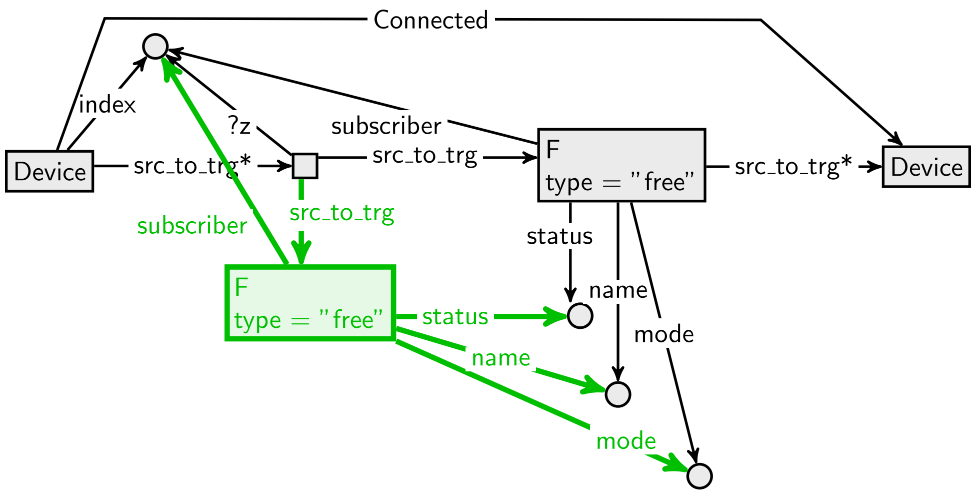
GTS rules for building a usage and its dynamic growth continued ...

AppendFeature-Trg



The usage is continued by assembling features subscribed by the target of the call.

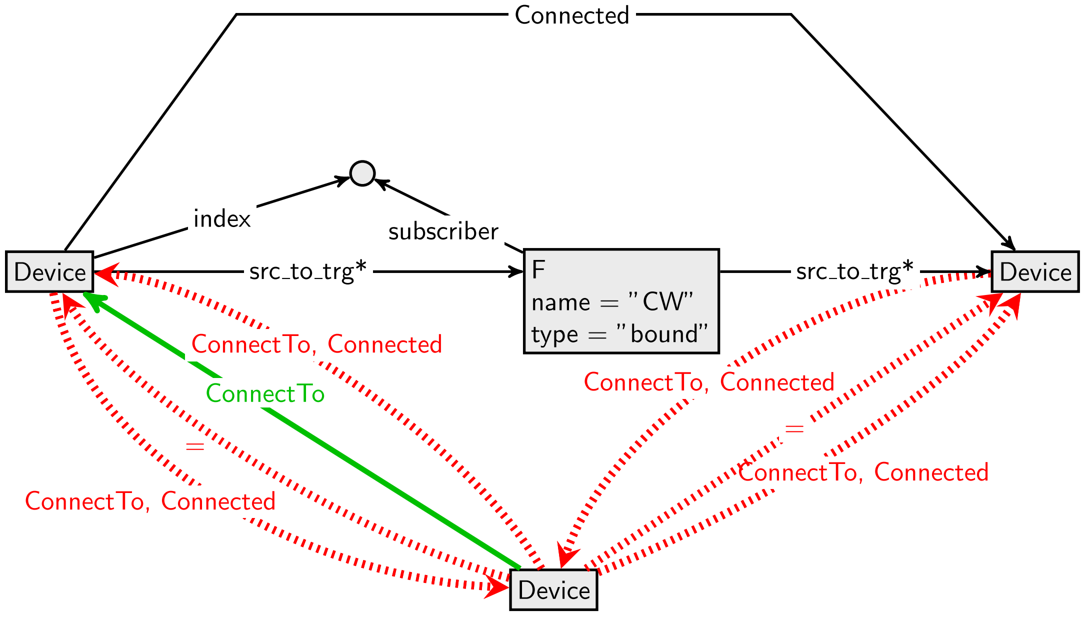
AppendFreeFeature



A new instance of a free source feature is created. The rule checks if the free feature box is already participating in a usage. The rule covers two cases: one is when the free feature is appended right after a device, the second is when a free feature is appended after an existing feature. That is why a non-labeled box is used that is connected to the free feature. This is an abstraction over the box types. A similar rule models the creation of a target free feature.

GTS rules for building a usage and its dynamic growth continued ...

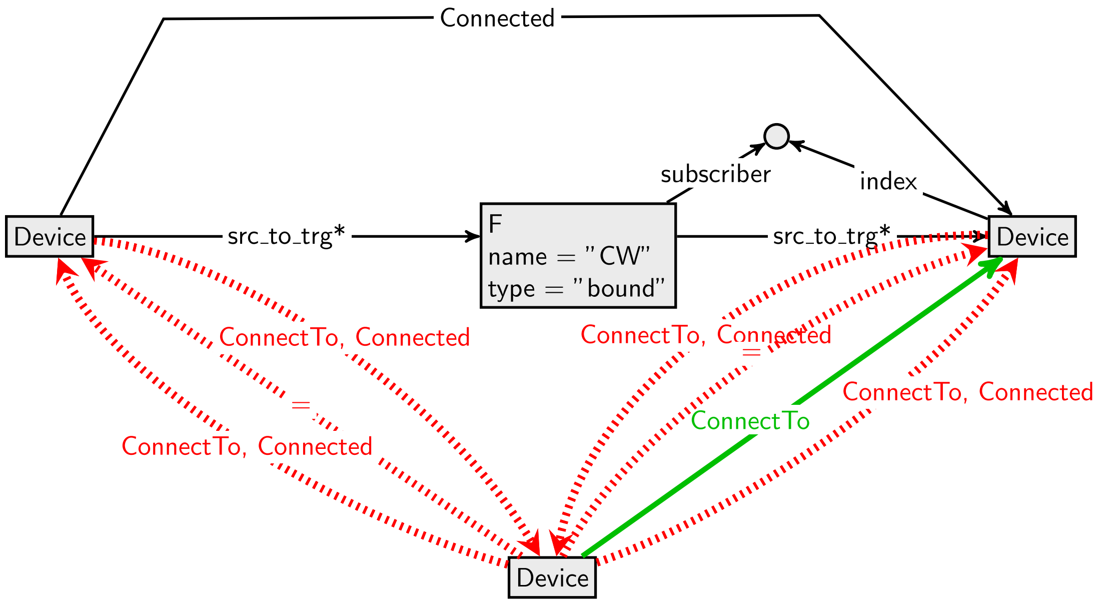
InitiateSrcJoinConnection



A new device is setting up to join an existing usage via a source CW feature. The rule checks that there is not any existing initiation for a connection or that the new device is not already connected to any of the existing devices participating in the usage.

GTS rules for building a usage and its dynamic growth continued ...

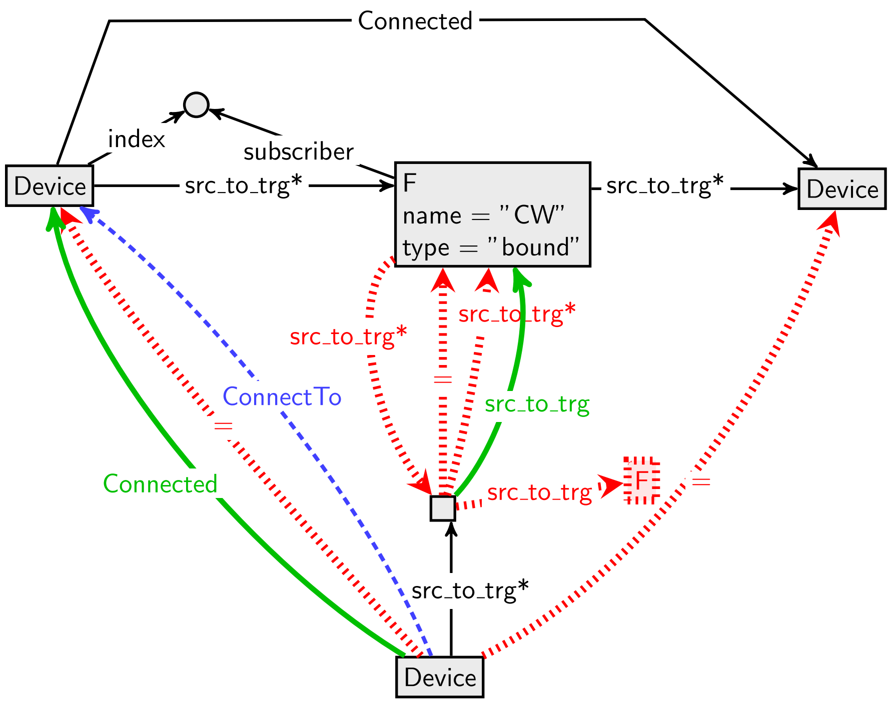
<u>InitiateTrgJoinConnection</u>



A new device is setting up to join an existing usage via a target CW feature. The rule checks that there is not any existing initiation for a connection or that the new device is not already connected to any of the existing devices participating in the usage.

GTS rules for building a usage and its dynamic growth continued ...

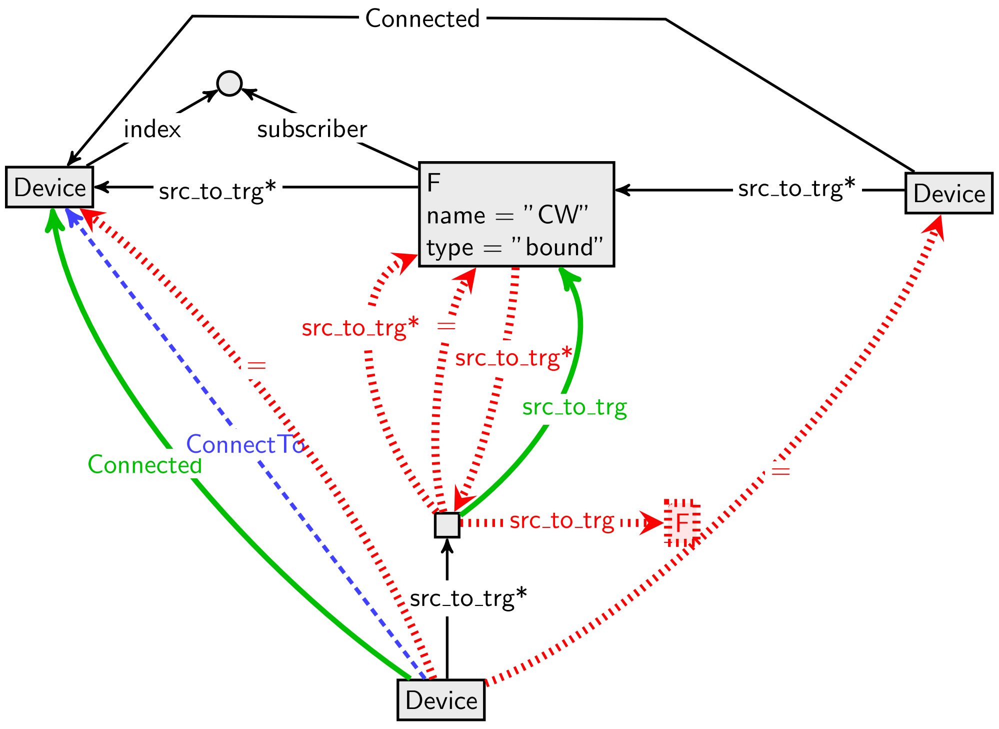
UsageJoinViaSrcFeature



A new usage joins an existing one via a source CW feature. The rule checks that the connection is done through a feature at the end of a created path between the new device and the CW feature. It is also checked that there is not an existing path to the CW feature in any other direction. This check is necessary to avoid reconnection of existing features on the existing usage.

GTS rules for building a usage and its dynamic growth continued ...

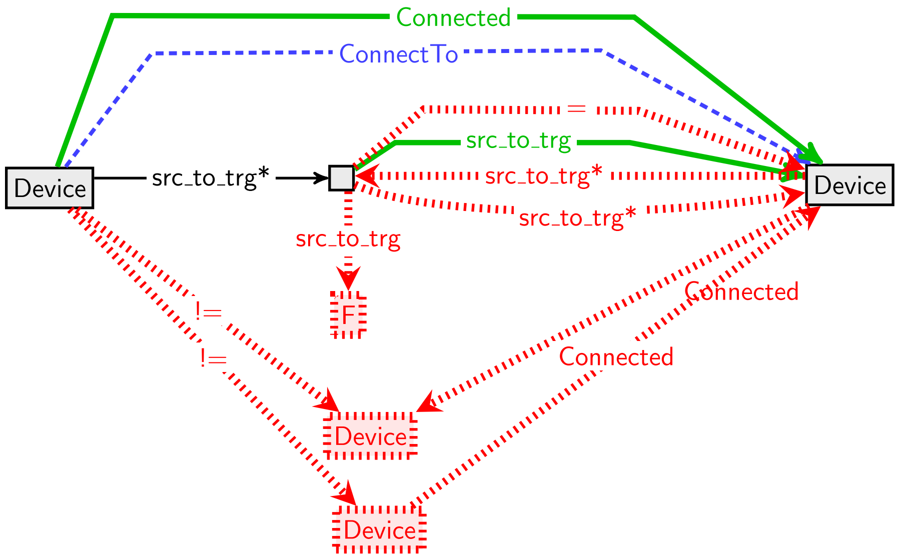
UsageJoinViaTrgFeature



A new usage joins an existing one via a target CW feature. The rule checks that the connection is done through a feature at the end of a created path between the new device and the CW feature. It is also checked that there is not an existing path to the CW feature in any other direction. This check is necessary to avoid reconnection of existing features on the existing usage.

GTS rules for building a usage and its dynamic growth continued ...

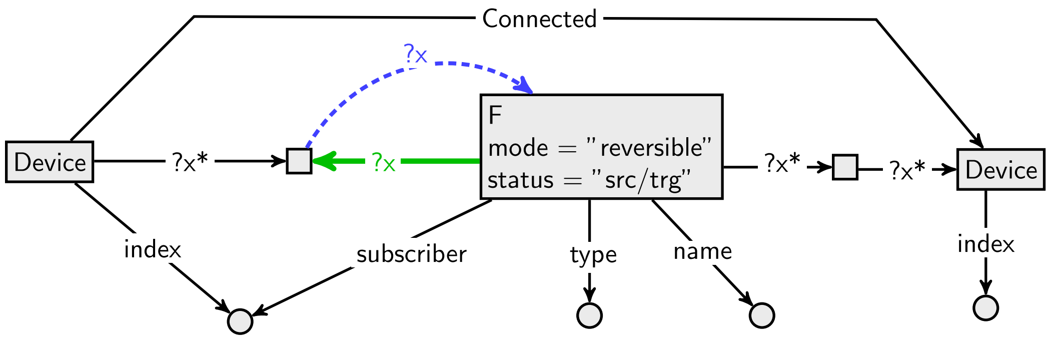
JoinCallee



Usage completes by joining the recipient of the call, that is the creation of a path between two devices is completed. The rule checks whether there is not an existing path between a feature and the end party in any direction. The NAC with "=" checks whether the box that would like to continue the usage is not the same as the device box. The NACs that check the non-existence of devices are there to avoid the application of the rule to partial usages that would like to join an already existing usage via CW, e.g., cases when the UsageJoinViaSrcFeature is applicable. If we did not have these NACs, the rule mistakenly can join a partial usage to an existing device without passing through the CW feature. In other words, it means if there exists another device that is connected to the device on right, then JoinCallee is not the right applicable rule and instead the rule UsageJoinViaSrcFeature or UsageJoinViaTrgFeature should be applied.

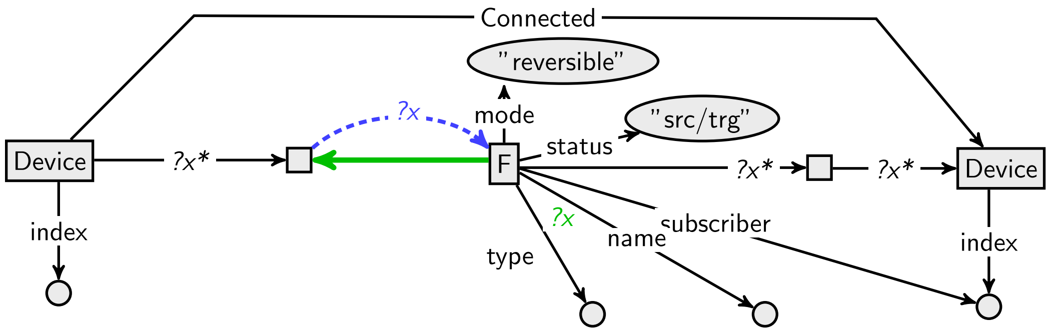
GTS rules for reverse signal operating in a usage.

ActivateCallerReverseFeature



Activating a reverse signal of a source reversible feature.

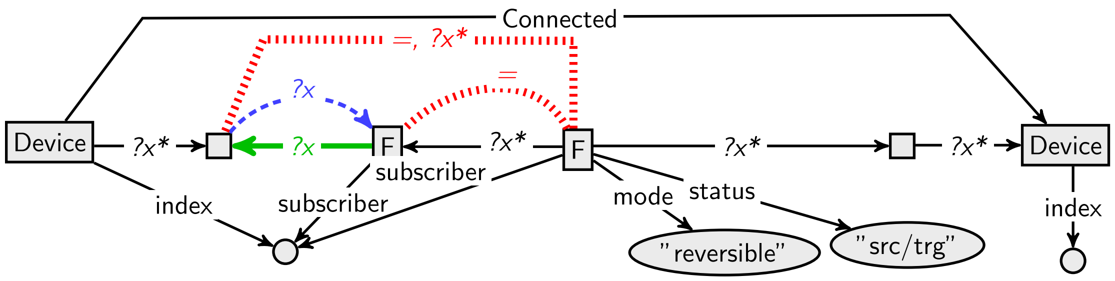
ActivateCalleeReverseFeature



Activating a reverse signal of a target reversible feature.

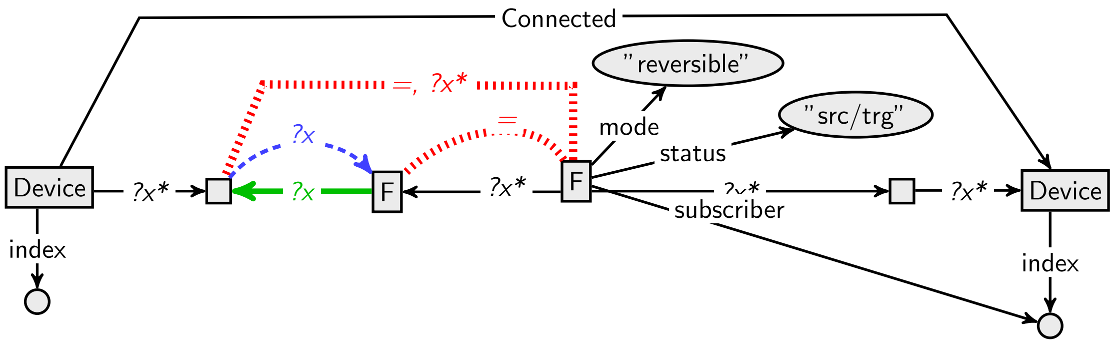
GTS rules for reverse signal operating in a usage.

PropagateReverseActivationSrc



Reverse signal of a source reversible feature is propagated to the caller. It is checked that there is not an existing path between the reversible feature and the one that the reverse signal is propagated to.

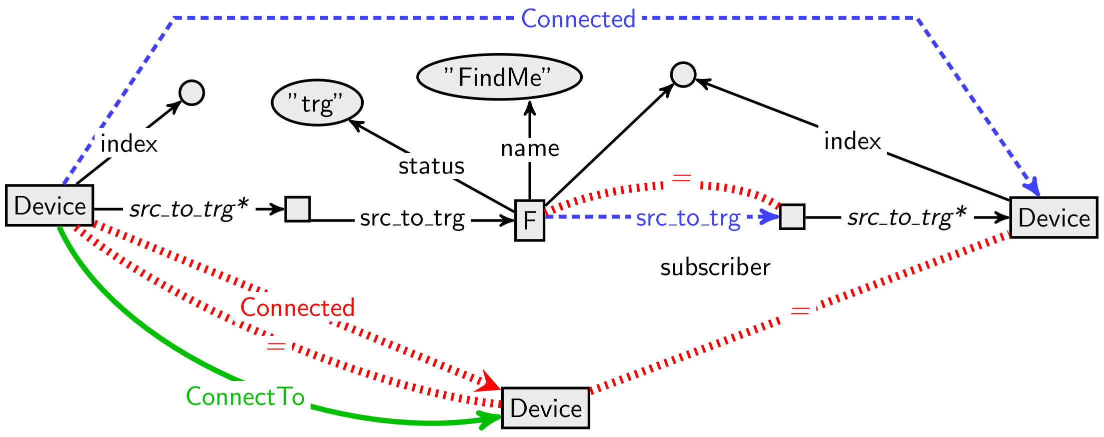
PropagateReverseActivationTrg



Reverse signal of a target reversible feature is propagated to the caller. It is checked that there is not an existing path between the reversible feature and the one that the reverse signal is propagated to.

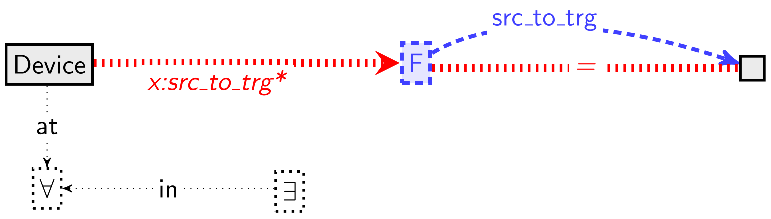
GTS rules for dynamic shrinking of a usage.

ActivateFindMe



This rule activates Find Me feature by disconnecting the usage from the callee and attempting to connect to a new device.

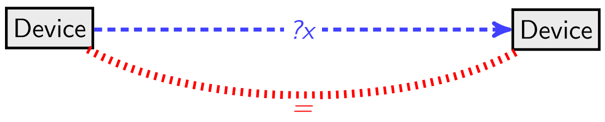
ShrinkTornDownUsage



Usage shrinks by removing disconnected features. Rule checks that every existing device does not have a connection path to the feature that rule is going to remove. In other words, the usage has been torn down before, e.g. via Find Me.

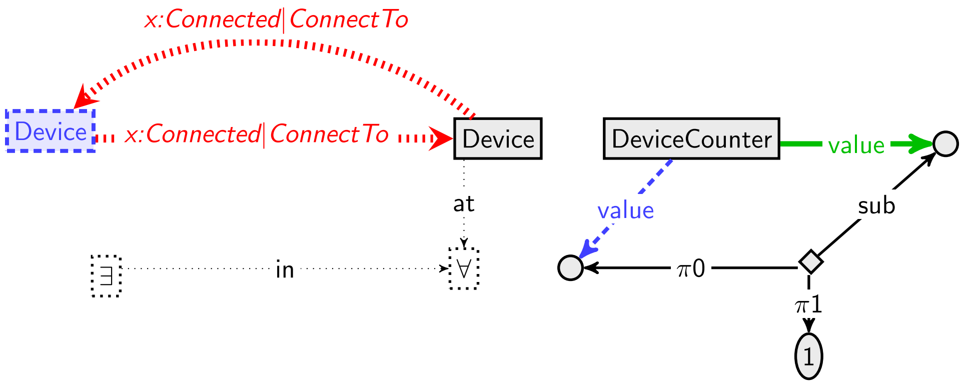
GTS rules for dynamic shrinking of a usage.

TearDownAUsage



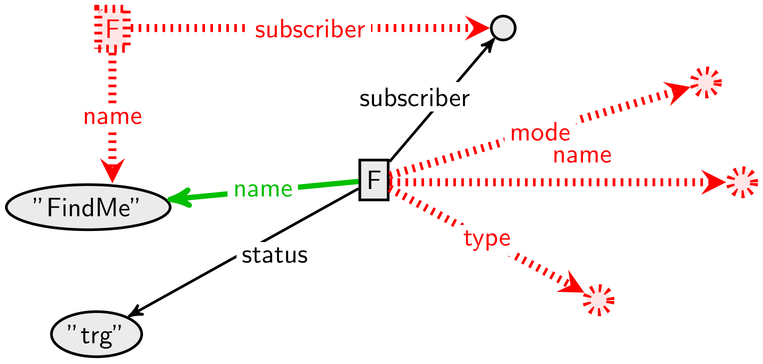
Disconnects a call between two different devices.

RemoveDevice



Removes an orphan device, and decreases the device counter.

NameFeatureFindMe



A target feature is distinguished as a Find Me feature.

Chapter 4

Verification of Dynamic Systems using GTS

4.1 Introduction

A communication system evolves dynamically with the addition and deletion of services. In the previous chapter, a graph transformation system (GTS) was used to model the dynamic behaviour of a telecommunication system [LT06]. In this chapter, we show how GTS formalism can facilitate the verification of invariant properties of potentially infinite-state communication systems. We use this approach to verify an invariant property of telecommunication service components that can act both as the source and the target of a connection.

Verifying an ordering among service components to be invariant is essential to guaranteeing the desirable behaviour of these services. We show how verification can be performed by the analysis of a finite set of transformation rules describing the GTS system model. We prove that invariant properties are preserved in a GTS model if the set of transformation rules describing the model preserves the property. Thus, we show how to perform system verification through analysis of the model description without building the full system state space.

In Section 4.2, the motivating problem and a description of the specifications for the ordering property of reversible features in DFC are presented, and this is followed by a description of the verification problem and its analysis in Section 4.3. In Section 4.4, we present the verification of the ordering property. Related work is presented in Section 4.5, and we conclude in Section 4.6.

4.2 Motivating Problem: Reversible Features in DFC

Reversible features are able to initiate a call on their own. For instance, after a connection has been dropped due to a failure, such a reversible feature will re-establish the connection to the appropriate endpoint. As examples of reversible features in DFC, we can name Call Waiting; Automatic Call Back, which offers the caller an automatic call whenever the callee is idle after the call fails in the first attempt (e.g. busy line); and Mid-Call Move, which moves an endpoint of a connection from one device to another while the caller and the callee are in "talk" mode. Reversible features ensure that messages are not lost and that the connection is long-lasting and continuous. This continuity is characteristic of well-behaved communication protocols.

A DFC usage with activated reversible features may be formed in an unusual way with internal calls in different directions. As an example, consider the usage in Figure 4.1, with features A, B, C, and D subscribed to by address X, in which A, B, and D are reversible (depicted in bold) and feature D is activated. Before the activation of D, all the internal calls between features were in the same direction from left to right.

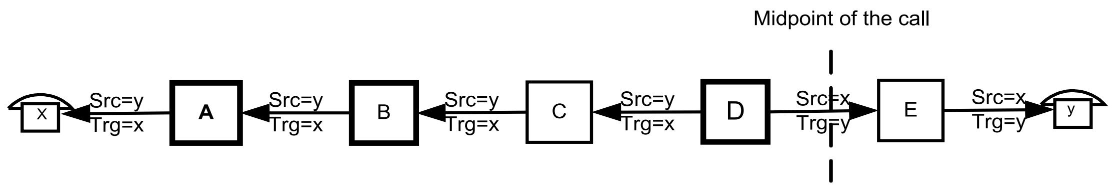


Figure 4.1: A usage with reversible boxes A, B, and D.

A strong motivation is to find a way of managing interactions among reversible features by imposing an invariant-ordering on them that satisfies the system specification and governs desirable interactions. The invariant-ordering property of reversible features is significant in a usage because it guarantees that the interactions of all the reversible features follow a desired pattern even though the usage has changed from its initial configuration [Zav03]. In the following section, we explain this property using DFC specifications.

4.2.1 Invariant-Ordering Property of Reversible Features

Using the informal description of DFC specifications given in the previous chapter, we have extracted three specification statements as first-order logic predicates to describe the invariant-ordering property in relation to orderliness of reversible features in DFC routing. In these statements, a is an address from the set of existing addresses, $Addr$; $Reversible$ is the set of reversible feature boxes subscribed to by an address; $SrcSubscriptions$ and $TrgSubscriptions$, respectively, show a partially-ordered set of source and target subscriptions of an address with respect to a $Precedes(\prec)$ relation ; $Precedes(\prec)$ is a relation between feature boxes that shows the order in which they are assembled in the usage of an address; and R and R', respectively, are the projection of source and target subscriptions onto reversible boxes. Therefore, for address a we have:

$a.R = a.Reversible \cap a.SrcSubscriptions$
$a.R' = a.Reversible \cap a.TrgSubscriptions$

In Figure 4.2, we have the specification statements about reversible features. The first specification in the box states that if an address subscribes to a reversible box, it must subscribe to it both as a source and as a target feature. The second specification states that the precedence order relation ($\prec$) in the set of reversible boxes being subscribed to as source subscriptions of an address is a total order. In this specification, if R is replaced by R' we get the total order for reversible boxes being subscribed to as target subscriptions. The third specification expresses that the subset of source and target reversible features of an address have an opposite precedence order. The third specification is needed to ensure that a usage consisting of multiple features is constructed in an acceptable way, and in particular to ensure that a new usage does not need to be established when the direction of signals is reversed. Using the above specifications, we have this property:

1. $(\forall a \in Addr, \forall f \in a.Reversible) \Rightarrow$
   $(f \in a.SrcSubscriptions \Leftrightarrow f \in a.TrgSubscriptions)$

2. $(\forall a \in Addr, \forall f_1, f_2 \in a.R, f_1 \neq f_2) \Rightarrow$
   $(((f_1 \prec f_2) \vee (f_2 \prec f_1)) \wedge \neg((f_1 \prec f_2) \wedge (f_2 \prec f_1)))$

3. $(\forall a \in Addr, \forall f_1, f_2 \in a.Reversible, f_1 \neq f_2) \Rightarrow$
   $((f_1 \prec f_2, \{f_1, f_2\} \subseteq a.SrcSubscriptions) \Leftrightarrow$
   $(f_2 \prec f_1, \{f_1, f_2\} \subseteq a.TrgSubscriptions))$

Figure 4.2: Specifications about reversible features

Requirement *In a usage associated with an address, if we order the feature boxes from outermost (closest to the endpoint) to innermost (closest to the midpoint), the sequence of reversible feature boxes associated with the address is an invariant sequence regardless of how the usage is initially constructed.*

4.2.2 Example

To justify the invariant ordering, we use an example that shows an undesirable behaviour when reversible features do not follow an invariant ordering. The example in Figure 4.3 shows that a usage has been set up between a source x, who subscribes to reversible features Call Waiting (CW), 3-Way Calling($3WC$), and Automatic Call Back (ACB), and the end party y. For this example, we are not concerned with the subscribed features of end party y. Scenario 1 in Figure 4.3 demonstrates a usage of address x with the source subscription ordering: $CW \prec 3WC \prec ACB$. Automatic Call Back acts as a source feature, and if an outgoing call fails it offers the user x a chance to activate the feature. If x does so, the ACB box disconnects from $3WC$ and waits until the original callee is available; this behaviour is depicted in Scenario 2 of Figure 4.3, though we are not concerned with the details of how ACB knows that the callee is available. Once y becomes available, as depicted in Scenario 3 of Figure 4.3, ACB places a call to x using reverse signal and the target subscription ordering $ACB \prec 3WC \prec CW$. When it is connected to x, it places a (hopefully successful)

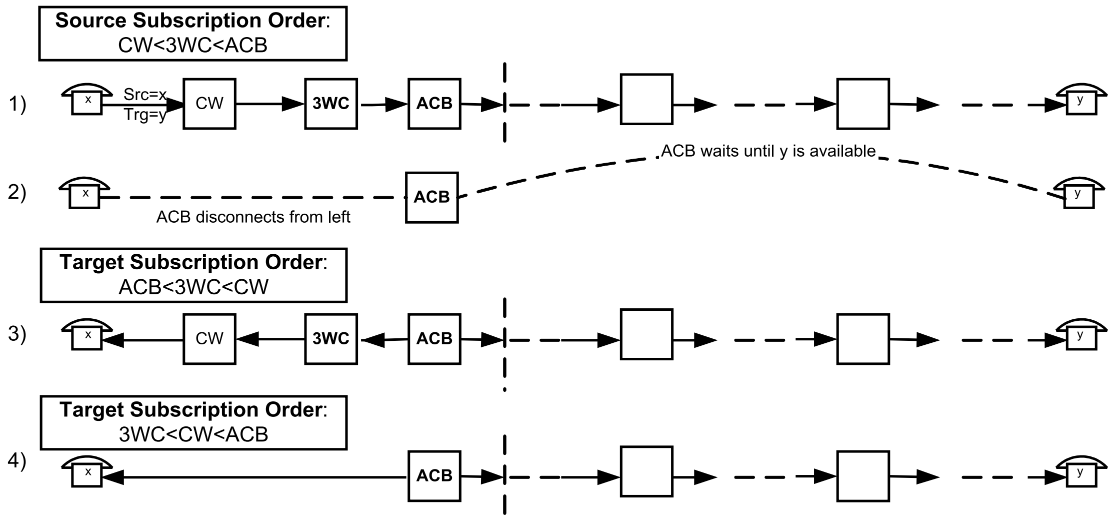

Source Subscription Order:
CW<3WC<ACB
1)
Src=x
Trg=y
x
CW
3WC
ACB
y
ACB waits until y is available
2)
ACB disconnects from left
Target Subscription Order:
ACB<3WC<CW
3)
Target Subscription Order:
3WC<CW<ACB
4)

Figure 4.3: Interaction of three reversible features in a usage.

call to the original callee. The call to the callee is a continue signal rather than a reverse signal.

Now, let us assume there is an error in the target subscription order and we have the order as $3WC \prec CW \prec ACB$. As shown in Scenario 4 of Figure 4.3, because of the incorrect order, ACB is the last feature in the order, and $3WC$ and CW are not routed to. Therefore, if x is actually engaged in another call via CW, it will miss the call from ACB, even though CW would have enabled it to take the call.

We conclude that the reversible sequence associated with an address must have a fixed ordering from left to right. In the following sections, we model this property as a graph and show how to verify it via the GTS formalism.

4.3 Verification Through GTS

We define the verification problem of a GTS and present our method to solve it. In [KR06], Kastenberg and Rensink showed that a labelled-transition system (LTS) created by a graph transition system can be seen as a representation of a Kripke Structure for a system model. Therefore, to verify an invariant property on a LTS, all the reachable states should be verified to check that they satisfy the property. GTSs can generate an infinite-state model; hence, automatic verification is problematic for these systems. At an appropriate level of abstraction, communication protocols are typically modelled as infinite-state systems. Thus, it is reasonable to model them using GTSs.

Here we address the verification of invariant properties that are expressed by the CTL modality AG and atomic propositions [HPR06] modelled as graphs called *proposition graphs*. We consider positive propositions, because in most cases negative constraints can be expressed by negation of positive constraints [EEPT06].

We use the GROOVE [KR06] notation for expressing proposition graphs with labels as regular expressions (e.g. Kleene star labels). We use these labels to compactly express feature connectivity patterns, for instance, to show that between two features of interest there may be an arbitrary length sequence of intervening features. Therefore, we need to extend the definitions of graph and graph morphism to include Kleene star labels. The following definition of *regular expression graph* provides this extension. A regular expression graph is a graph in which edges may be labelled with the Kleene star operator over the set of labels. Using regular expression graphs in the proposition graphs and transformation-rule graphs makes these graphs more expressive.

Definition 4.1 (Regular Expression Graph (REG)). *An REG is a graph G where for a set of labels, L, the labelling function Lab is defined as $Lab : E \to \{l^+ \mid l \in L\} \cup \{l^* \mid l \in L\} \cup L$ where l^* and l^+ represent Kleene closure and the positive Kleene closure of l.*

An REG can be a subgraph of the left-side or the right-side graph in a transformation rule or a subgraph of a proposition graph. Examples of these two cases are depicted in Figures 4.7 and 4.8 respectively. There is an exception that Kleene-star labels are not allowed on newly created graph edges (on the right side of a rule) or on edges to be deleted (on the left side of a rule.) The reason is that Kleene-star represents a multiplicity but it is not known how many, so if we have a Kleene-star label on an edge that has to be created or deleted there should be a mechanism to state what number the star represents.

Definition 4.2 (Path). *In a graph $G = (V, E, Src, Trg, Lab)$, a path p from node v_1 to node v_n is a sequence of nodes connected by edges: $p = \{v_1, v_2, ..., v_n\} \subseteq V$ such that $\{e_1, e_2, ..., e_{n-1}\} \subseteq E$, $v_1 = Src(e_1)$, and for all $1 \leq i \leq n-2$, $v_{i+1} = Trg(e_i) = Src(e_{i+1})$, and $v_n = Trg(e_{n-1})$, and $Lab(e_i) \in Labels^*$.*

In a path, if there are Kleene-star-labelled edges, then G is an REG and the path is defined for an REG. Hence, the sequence s of edge labels in path p, written as $s = (Lab(e_1)...Lab(e_{n-1}))$, specifies a language. For example, for a path p, with consecutively labelled edges a, x^*, y^* and b, the sequence $s = (ax^*y^*b)$ specifies the language of path p, and the string $w = axb$ is a member of that language. To specify the satisfaction of a proposition graph (an REG) by another graph (possibly an REG), we need to define a specific type of morphism to map a path in one REG to another path in the second REG.

Definition 4.3 (Regular Expression Graph Morphism). *$rgm : G \to H$ is an REG morphism between graphs G and H, if either G or H is an REG and for a path $p = \{v_1, ..., v_n\}$ in G there is a path $q = \{u_1, ..., u_n\}$ in H such that:*

- *There is a graph morphism $m : V_G \to V_H$ between the beginning and the end nodes of these two paths, written as $u_1 = m(v_1)$, $u_n = m(v_n)$.*
- *For $2 \leq i \leq n-1$, three cases may occur:*
  1. *If both G and H are REGs, then the language specified by the sequence of corresponding labels over the edges connecting nodes v_i in p is a subset of the language specified by the sequence of labels over the edges connecting nodes u_i in q.*
  2. *If H is an REG, and G is a graph without Kleene-star-labelled elements, then the string specified by the sequence of corresponding labels over the edges connecting nodes v_i in p is a member of the language specified by the sequence of labels over the edges connecting nodes u_i in q.*
  3. *If G is an REG, and H is a graph without Kleene-star-labelled elements, then this is similar to case 2 with respective changes for G and H.*

We have total or partial REG morphisms, if the mappings are respectively total (for the set of non-overlapping paths in G) or partial.

The verification problem is solved using a forward-analysis method that ensures that the invariant property is never violated by the application of transformation rules. Thus, the required invariant is satisfied by the system model. Though the example we used here for the invariant property is an ordering property of DFC, the method can be used for any invariants that are expressed by positive proposition graphs.

To express the verification problem of invariant properties, we define our notion of graph satisfaction:

Definition 4.4 (Graph Satisfaction)**.** *An REG or a graph G satisfies an REG or a graph ϕ, $G \models \phi$, iff there exists an appropriate total morphism (graph or REG morphism) m between ϕ and G written as $m : \phi \to G$. A graph G weakly satisfies a graph ϕ, written as $G \models^w \phi$, iff there exists a non-empty appropriate partial morphism (graph or REG morphism) between ϕ and G, $m^w : \phi \to^w G$.*

Following this definition, then $G \not\models^w \phi$ (G does not weakly satisfy ϕ) means there is no morphism between ϕ and G, and it implies that G does not satisfy ϕ. Note that empty morphisms, where there is no mapping between two graphs, are in fact types of partial morphism, but they are excluded from the definition of $G \models^w \phi$; therefore, empty morphisms are considered to be non-satisfying. Figure 4.4 shows a graph satisfaction

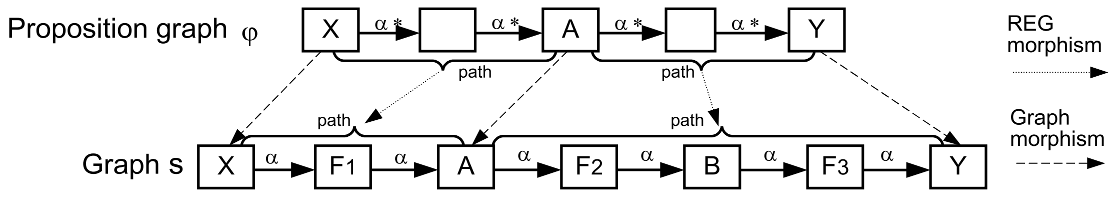


Figure 4.4: A proposition graph φ is satisfied by graph s.

example based on morphisms in Definition 4.3.

Definition 4.5. *Propositional expressions are defined as:*

- *Every atomic proposition is a graph (Definition 2.6) or a REG (Definition 4.1) called proposition graph.*
- *If ϕ and ψ are proposition graphs then so are:*
  - $\neg\phi$ • $\phi \vee \psi$ • $\phi \wedge \psi$

Based on the Definition 4.4, the semantics of graph satisfaction is defined with respect to a graph or REG G, over a set of proposition graphs such as φ and ψ.

- $G \models \varphi$ iff there exists an appropriate total morphism (graph or REG morphism) m between φ and G.
- $G \models \neg\varphi$ iff $G \not\models \varphi$
- $G \models \varphi \vee \psi$ *iff* $G \models \varphi$ *or* $G \models \psi$
- $G \models \varphi \wedge \psi$ *iff* $G \models \varphi$ *and* $G \models \psi$

Using the above definitions, the following theorem states a satisfying condition for a transformation rule to preserve a property ϕ. In general, when the property ϕ is satisfied in a state, the transformation rule preserves the property, if this rule does not transform the state graph in a way that violates the property. That is, given a property ϕ as a proposition graph, a rule r is guaranteed to *preserve* ϕ if its left side graph does not weakly satisfy ϕ or its right side satisfies ϕ.

Theorem 4.1 (Property Preservation)**.** *Given $\mathcal{G} = \langle S, T, I\rangle$, let ϕ be a state property, and $\langle s, r, t\rangle \in T$, where r is a rule, $r : L \to R$, and $s, t \in S$, and suppose $s \models \phi$. If $L \not\models^w \phi$ or $R \models \phi$, then $t \models \phi$.*

Proof. Based on Definition 3.8, when a rule is applied to a state graph s, it reconfigures s resulting in another state graph t. Considering the fact that both s and ϕ are graphs, if s already satisfies the property ϕ, and if r does not reconfigure parts of s that have a mapping to ϕ, then the resulting graph satisfies ϕ.

To make sure that a rule preserves a property in the transformed state, we have to check how the transformation affects the existing mapping of ϕ to s. Thus, the morphisms between the left-side and the right-side graphs of r and ϕ determine if r affects ϕ's mapping to s or not. In other words, based on Definition 4.4, how L and R satisfy ϕ leads us to find out if r's transformation process violates ϕ. Though there might be different types of satisfaction relations between L, R and ϕ resulting in different combinations, all those combinations can be summarized in two cases, which we prove separately:

1) $s \models \phi$ and $R \models \phi \Rightarrow t \models \phi$

2) $s \models \phi$ and $L \not\models^w \phi \Rightarrow t \models \phi$

Case 1 This case states that if the right side of a rule satisfies a property, no matter what the satisfaction situation with its left side is, the rule's application preserves the property. Based on the rule's application process defined in Section 2.2.3, the common elements in R and L are preserved and other elements in R are created in the transformed state. Therefore, if R satisfies ϕ, then ϕ's elements will be mapped to the transformed state, no matter if L satisfies ϕ or not.

Case 2 In this case, when $L \not\models^w \phi$, there is no morphism (partial or total) between L and ϕ. Since $s \models \phi$ and there is a total graph or REG morphism between L and s (Definitions 3.8, 4.1), then whether R satisfies ϕ or no, either r transforms s with the creation of some or all elements of ϕ, or it transforms a part of s that does not map to any elements of ϕ. $\square$

The second case occurs when a rule transforms parts of a big graph that do not interfere with the proposition graph elements; in other words, those parts do not have a mapping to the proposition graph. In all other cases ($L \models \phi$ and $R \not\models^w \phi$, $L \models \phi$ and $R \models^w \phi$, $L \models^w \phi$ and $R \models^w \phi$), the rule may transform the graph in a way that elements mapping to the proposition graph in the left side are deleted in the transformed state. These cases may therefore violate the property. Thus, such transformation rules may or may not preserve the property. As a result, this verification technique is sound but incomplete (may introduce false negatives).

Note that this theorem may not hold for GTSs without node identification [Roz97], where we have similar nodes with the same attributes, or when rules are applied based on non-injective mappings. For instance, consider the non-injective mapping of the left side of a rule to a graph s where there are two sets of mappings from L to a subgraph in s. If both sides of the rule and also s preserve a property (based on the same subgraph mapping), and the rule applies to s with the goal of deleting the mapped subgraph, because of the priority of deletion over preservation in the rule application approach [LT06], the transformed state graph does not satisfy the property. This exception does not occur in our GTS rules built to present DFC behaviour, because we use graph nodes with identification and injective graph morphisms for the rules application in our modelling.

4.3.1 Proving Properties using Structural Induction

With the above theorem and structural induction [Bur69, MNV72], we prove the satisfaction of invariant properties defined as a CTL formula $AG(\phi)$ in a GTS description of a system. The structure we work with in our system model is a graph structure, and our system model is built up from many of these structures. If we show that a primitive object of our structure, which is an initial state graph has a desired property, and also show the act of building up the model preserves the property then we have shown that all states with graph structure must have the property. This problem is specified in the following theorem.

Theorem 4.2. *For the graph transition system $\mathcal{G} = \langle S, T, I \rangle$ and the state property ϕ, $\mathcal{G} \models \phi$, if $I \models \phi$ and for all rules $r \in \mathcal{G}$, r preserves ϕ.*

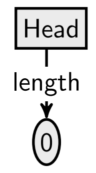
Proof by structural induction:

Basis: $I \models \phi$. For the initial state of the system, the satisfaction of the property can be checked using the Definition 4.4.

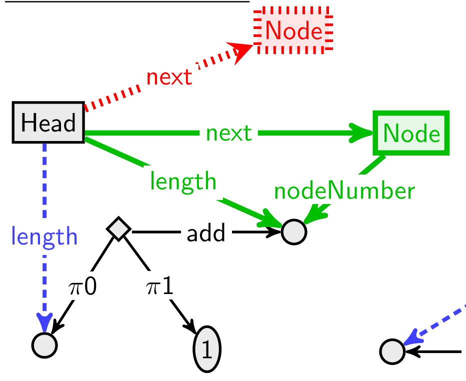
Induction step: A hypothesis is made to assume that the property is preserved by all the rules in $\mathcal{G}$ that are applied to the states up to the kth level in the LTS based the Theorem 4.1.

In the induction step, since we showed that the property is satisfied at the states of the LTS in level k, and because all rules are property preserving, then the given property also holds through application of all rules at level $k+1$ of the LTS. □

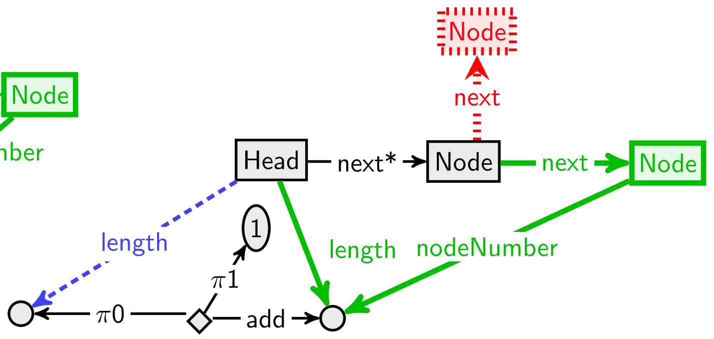
Example: In a linear linked list, we may want to verify that always the length of the list is equal to the number of nodes in the list unless list is empty. The property can be specified as an invariant, $AG(\phi)$, where ϕ is a combination of two propositions expressing either the list is empty or the length is equal to the number of nodes, $\phi = \mathit{IsListEmpty} \vee \mathit{Equal}$. The transformation rules that build the list, and create or delete nodes to or from it have been illustrated in Figure 4.5. The proposition graphs for checking the above invariant



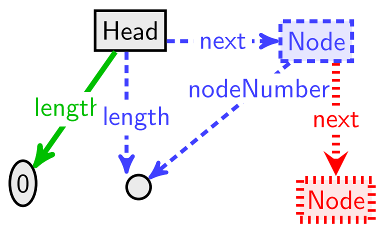
initial state graph



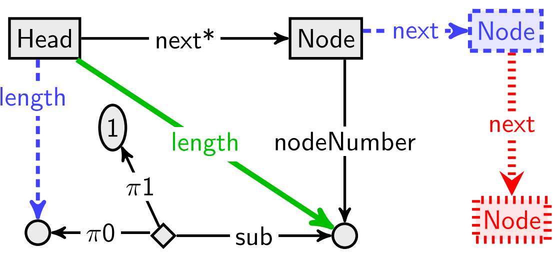
add-first-node rule



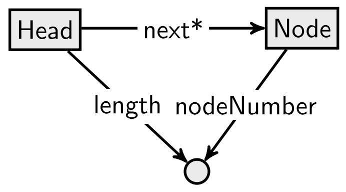
add-node rule



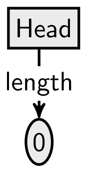
del-first-node rule



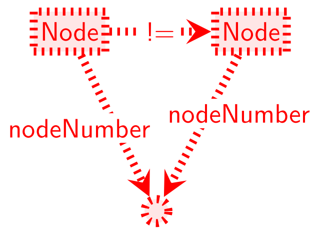
del-node rule



Equal



IsListEmpty



NoDuplicate

Figure 4.5: Initial state, four GTS rules, and three propositions for a linked list.

requirement are also depicted in Figure 4.5. The proposition graph IsListEmpty shows an empty linked list when the length of the list is zero and no other node is connected to the head of the list. Based on the above theorem, this proposition is satisfied in the initial state graph and preserved by the GTS rule del-first-node. The proposition graph Equal describes that the length of the linked list which is shown as an attribute of the head of the list is equal to the number of the nodes in the list. Each node keeps track of the count of the nodes by an attribute nodeNumber. The Equal proposition is satisfied by the other rules of the GTS model. Therefore, this property is satisfied in the initial state and preserved by all the rules. On the other hand, the proposition graph NoDuplicate is a proposition, which expresses that two nodes may not have the same node number. This proposition is a false negative example, where it may be satisfied by the transition graph of the model and also by all the state graphs, but it is not preserved by the transformation rules of the model using the above theorem.

Automation

The above proof can be automated using the following algorithm.

```
boolean IsInvariant
Input: $s_0 \in I$: initial state graph, $P$: set of GTS rules, $\phi$: proposition graph
Output: true if $s_0$ and all $r \in P$ preserve $\phi$, else false
1 $i \leftarrow 1$
2 if $s_0 \not\models \phi$ then
3     return false
4 forall $r \in P$ do
5     if $r.L \models^w \phi$ and $r.R \not\models \phi$ then
6         return false
7 end
8 return true
```

Figure 4.6: Property satisfaction

In this algorithm, the main task is checking the graph satisfaction in lines two and six. As defined before, graph satisfaction needs REG and graph morphism checking, so for these two lines we can use optimal algorithms such as [McK81] for graph morphism checking. Also as part of a future work, an algorithm needs to be developed for checking the REG morphism based on the path matching in graphs.

4.4 Experiments with DFC

Verifying an $AG(\phi)$ Property in a Usage

After the establishment of a connection in a DFC usage, the usage can grow, shrink, split or merge with other usages. The establishment of a connection is based on the DFC routing algorithm, which builds a connection from the ordered feature list associated with an address. Each feature box requests a connection using the "new," "continue," or "reverse" signals (Section 3.2.1).

The DFC routing algorithm is described by two GTS models in the following steps. In the first model, a GTS is used to build a connection for a usage associated with two endpoints. These GTS transformation rules describe the methods "new" and "continue" found in the DFC routing algorithm. The second GTS includes the transformation rules for activation of the caller's and callee's reversible features. From these rules, the caller's reverse activation rule is illustrated in Figure 4.7. The rules for propagation of reverse signal to the source of the connection, and the callee's reverse activation rules are similar, and presented at the end of Chapter 3.

After the establishment of a connection, which we know satisfies the ordering property that is given explicitly in the model description, the second graph transition system, which utilizes the built connection as the initial state, encodes dynamic changes of the usage graph. As an example, Figure 4.7 illustrates the rule for the activation of a caller's subscribed reversible feature. In this picture, edges labelled with $?x*$ (?x is a wildcard showing any label) show that two features are connected through a sequence of intervening features connected by x-labelled edges. Features have several attributes (depicted as circular nodes) such as their name, subscriber, and status, which shows if it is a source subscription or target subscription or both; and mode, which shows whether a feature is reversible. To express that the reversible feature F in Figure 4.7 is connected to a callee through a sequence of non-reversible features, Kleene-star-labelled edges are used (REG). This figure shows the activation of a caller's reversible feature.

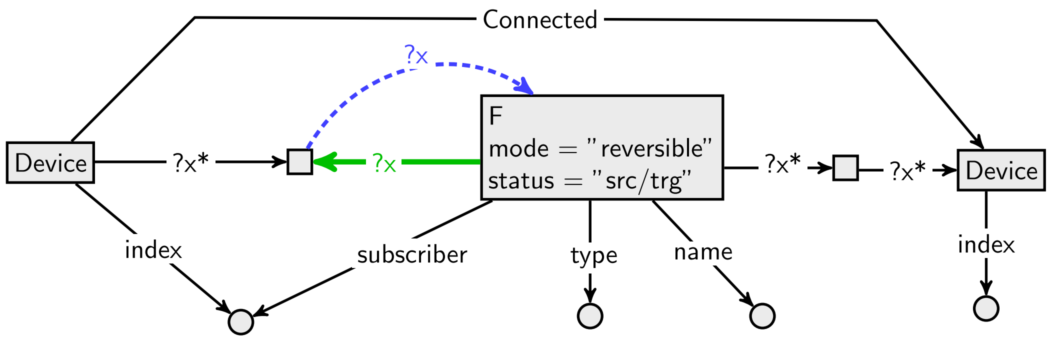


Figure 4.7: The GTS rule ActivateCallerReverseFeature.

We verify the invariant $AG(\phi)$ in DFC usages, where ϕ denotes the correct ordering of the reversible features of an address. We use the requirement in Section 4.2.1 and REG definition to encode property ϕ as proposition graphs connected by operators $\vee$ and $\wedge$. The first case to consider with any usage is when reversible features have been assembled into the usage, but are not active. Therefore, a proposition graph is needed to ensure the ordering of these features as they appear in the established-connection path of an address. The next step is to construct the proposition graphs that ensure the correct ordering of reversible features after they are active. For this step, we use the fact that there is a total precedence order relation among these features, and this ordering for source-subscribed features is the opposite of the ordering for the target-subscribed features of an address. Thus, when any of these features activates, it should be verified that only the direction of signals is changed, because the role of the reversible feature has been changed from source to target or vice versa. So these proposition graphs consider the correct activation of each reversible feature, ensuring that the existing established connection path is used.

The invariant-ordering property of two reversible features, A and B, subscribed to by a caller in an established connection between the caller and a callee in a DFC usage is stated as $AG(\phi)$. Two of the proposition graphs that participate in constructing ϕ are P and Q as depicted in Figure 4.8. In this picture, proposition P shows that there is an established connection between the caller and the callee, ensuring that the reversible features A and B appear in the connection in an order with zero or more non-reversible features in between (shown as Kleene star labelled edges). In this example, we are not concerned with the callee's subscribed features, so these features are abstracted as an F-

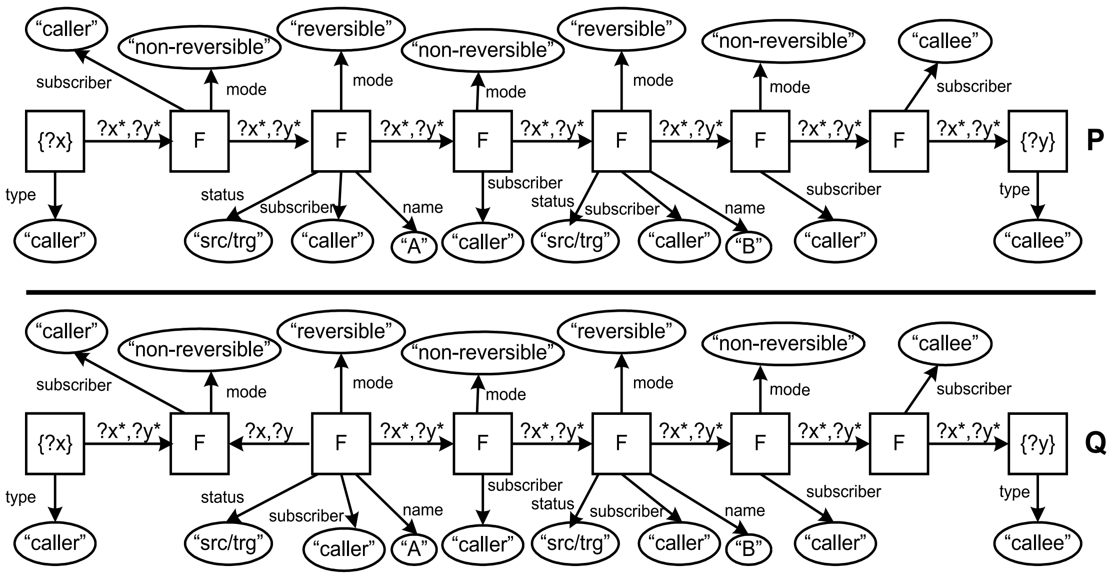


Figure 4.8: Propositions P, Q in the property $AG(\phi)$.

labelled node connected by zero or more $?x$ labelled edges. Proposition Q shows that if the reversible feature A activates, the order of B respective to A does not change in the established connection path from the caller to the callee. Though other propositions are omitted here, they are similar to Q for checking the propagation of reverse signal to the caller and the activation of reversible feature B.

Now we state the verification problem for invariant ordering of reversible features in a DFC usage in Lemma 4.1.

Lemma 4.1. *Let $\mathcal{G} = \langle S, T, I \rangle$ be the DFC transition system, and I be the state of an established connection associated with address A. Let ϕ be the invariant-ordering property of reversible features associated with address A, such that $I \models \phi$. If for all rules r in $\mathcal{G}$, r preserves ϕ, then $\mathcal{G} \models \phi$.*

The Lemma is easily proved using Corollary 4.2 to show that the ordering property in an established connection of a DFC usage can be verified through our GTS model of DFC. The GTS with an established connection as the initial state is given as a transition system $\mathcal{G} = \langle S, T, I \rangle$, where I is an established connection state. We have developed $\mathcal{G}$ with a small but important representative set of transformation rules that describe DFC behaviour for reversible features (presented at the end of Chapter 3 , and all are property preserving based on Theorem 4.1. After the connection has been set up between two endpoints, regardless of how the usage graph evolves, the ordering sequence of reversible features associated with any of the addresses at each end of the connection should remain fixed.

The number of rules that describe the behaviour of DFC are finite. Thus, we are able to verify invariant properties on a potentially infinite-state system.

4.5 Related Work

There are two main types of verification work using GTSs. The first type focuses on the verification of finite-state systems [Var03, dSDR06, KR06]. In the current work, which is a sound though incomplete technique (discussed in Section 4.3), we focus on dynamic systems that are not *a priori* finite state.

Similar to our work, the second type considers verification of infinite-state systems. Interesting works by Baldan, Corradini, and König [BCK01a, BCK04, BKR05] considers an abstraction approach through unfolding of a GTS by means of constructing finite Petri-net structures, but again, Petri nets are less powerful formalisms. These works provide an approximate technique and may introduce false positives [BCK01a, BCK04, BKR05]. In addition, in this approach, properties are not formulated graphically.

The work in [RD05a] also considers an abstraction approach called shaping that partitions elements of a graph based on their similarities. It uses an alternate GTS formalism that does not make use of graph morphisms utilized in the current work. Furthermore, the approach of [RD05a] is not precise, because concretization of the same abstract graph is different in shape and structure.

The work in [HPR06] focuses on assertional reasoning and constructing a weakest pre-condition, and though it is applicable to infinite-state systems, verifying invariants using this approach is difficult. In [HPR06], properties are graph morphisms based on rule application conditions, while in our approach properties are graphs that can be generated in a straightforward manner from user requirements.

The work in [BBG$^+$06] also addresses the verification of changing structure modelled by GTSs. This work is an approximative invariant verification method, and it only considers the conjunction of unsafe situations where the proposition graphs encode forbidden cases. Unlike our work that is a type of forward analysis of rules against the requirement graph, the work in [BBG$^+$06] does the verification using a backward reachability analysis. The work analyzes the actual system states that are unsafe by explicitly applying rules backward to these states. This work uses safe system states to build a requirement, whereas invariant requirements are often given by the user and can be built directly as a proposition graph.

4.6 Conclusion

In [LT06], it is shown that graph transformation is a powerful formalism for modelling the dynamic evolution of communication and telecommunication systems. In this chapter, we show a verification method for invariant checking on the GTS model of a telecommunication system. GTSs generate an LTS as the transition graph. The generated LTS for a communication system may lead to an infinite-state model, and verifying invariants on the model requires analysis of the full state space.

Using the graph satisfaction notion, we address the verification of an invariant using the finite set of transformation rules that describe the system behaviour. Furthermore, we present the conditions under which a transformation rule preserves a property. We then show that if the initial state of the GTS model satisfies the property, and if all transformation rules are also property preserving, then the system satisfies the property. Therefore, by analyzing the transformation rules, we are able to verify a property without explicit exploration of the state space.

This behaviour is true for invariants or properties of the form $AG(\phi)$. The proposition ϕ is described as a graph and we define two forms for transformation rules, and show that if all transformation rules are of one of these two forms, then they preserve ϕ.

Chapter 5

Symmetry for the Analysis of Dynamic Systems

5.1 Introduction

It is not always possible to avoid the explicit analysis of the GTS system model using the theorems 4.1 and 4.2 in Chapter 4. Because, that technique is sound but incomplete and if the conditions stated in Theorem 4.1 are not met for GTS rules, still the system model may satisfy a property. Therefore, in these cases we need to use other methods of abstraction to reduce the system state space for verification purposes. Due to the emergence of multi-core processes in communication systems, these systems use many identical processes and symmetry is often a feature of these systems models. Therefore, symmetry reduction seems to be an abstraction method of choice for reducing models of these systems.

Unfortunately, existing symmetry-reduction techniques [ID96, CEFJ96, ES96] that generate a reduced, bisimilar model for alleviating state explosion in model checking are not applicable to dynamic models such as GTSs. In addition, they may offer only limited reduction to systems that are not fully symmetric.

When systems are composed of several similar components, it is often convenient to identify the various components by their process indices. In a Kripke model of these systems, a state consists of the values of all global variables and the local states of each process. For example, consider a token ring network of three processes, as illustrated in Figure 5.1-a, where each process can communicate to the process on its right and a shared

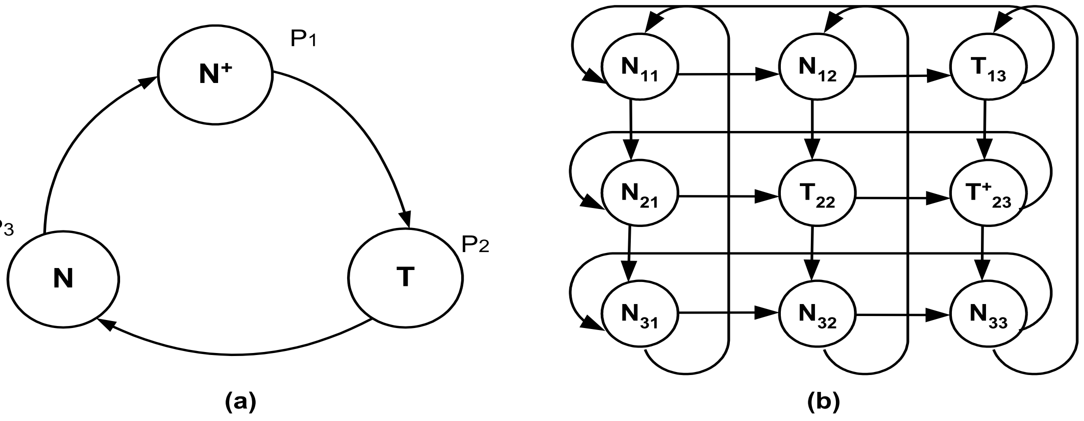


Figure 5.1: a) A ring with three processes. b) A 3 × 3 toroidal mesh.

token is used to show the access of processes to some resource. In this example, the state $N_1^+T_2N_3$ describes that process 2 is trying to access a shared resource and the other two are in their non-trying modes, while process 1 possesses the token (represented by a plus sign). Another example is a 3×3 toroidal mesh network of processes, as in Figure 5.1-b. A toroidal mesh is a grid network with wrap-around links, where each process can communicate to two other processes. In this example, the local state T_{23}^+ describes that the process in row 2, column 3 possesses a token and is trying to access a shared resource. Symmetries in these semantic models are then represented as permutations of the process indices. Symmetry-reduction methods [ID96, CEFJ96, ES96, TW09] use the index permutation to build a symmetry-reduced quotient model that is equivalent, up to permutation, to the behaviour of the original model.

In Kripke models, the labelling of each state does not explicitly show the architecture of the system. On the contrary, in a GTS model of the system, each global state is represented by a graph that explicitly provides the architecture in which processes are connected together [LT06, LT09]. Since index permutations do not respect the architecture of states, they cannot be used directly to represent symmetries of graph semantics and build equivalence classes of state graphs in non-fully symmetric GTS models. Instead, in graph-based semantic models, symmetries are represented as graph isomorphisms that are used to define an equivalence relation on the set of states presented as graphs.

In this chapter, we develop symmetry-reduction techniques applicable to dynamically evolving GTS models and the programs that generate them. Our symmetry-reduction approach is applicable to system architectures such as hypercubes, rings, and tori (used in metropolitan area networks that need high scalability) used for modelling next-generation communication protocols. We also provide an on-the-fly algorithm for generating a symmetry-reduced quotient model directly from the set of graph transformation rules. The generated quotient model is GTS-bisimilar to the model under verification and may be exponentially smaller than that model.

The rest of this chapter is organized as follows: the introduction is followed by a description of GTS symmetry, GTS bisimulation, and the quotient generation algorithm in Sections 5.2 and 5.3. We present vertex bisimulation for symmetric GTSs in Section 5.4 and in Section 5.5 show a GTS implementation of an evolving toroidal mesh. Section 5.6 describes the related research work and Section 5.7 concludes the chapter.

5.2 Symmetry in Dynamic GTS Models

We define symmetry for dynamic GTS models of systems which may not be fully symmetric, but that show some symmetry in their structure. Traditionally, for a fixed-size system, symmetries are represented by a group of index permutations [CEFJ96, ES96]. For GTS systems, we consider states to be symmetric if their associated graphs are isomorphic. Hence, isomorphism is used to define the permutation on state graphs. The set of permutations for each graph differs based on the graph structure and size. For example, for a ring, rotation is a graph permutation. For a toroidal mesh, graph permutations include simultaneous rotation of all rows, simultaneous rotation of all columns, and flipping the toroidal mesh.

In dynamic systems, where the number of components may change, we consider sets of such permutations to define symmetries for different state sizes. In fact, there are different groups of permutations for graphs with different sizes. The state graph permutation implicitly considers the number of nodes in a graph because graph isomorphism is used to define these permutations and isomorphism is based on a bijection on the sets of nodes and edges of the graph. For specific graphs of n nodes, we use the notion π_n to show a permutation on those graphs. For a ring of size n this permutation is a rotation on an n-node ring. For a $k \times k$ toroidal mesh (where $n = k \times k$), a permutation is either the rotation of k horizontal rings, the rotation of k vertical rings, or a mix of these rotations. $k - 1$ horizontal rotations followed by a vertical one or $k - 1$ vertical rotations followed by a horizontal one is actually a flip for the $k \times k$ toroidal mesh, where the flip α is defined as $\alpha(i, j) = (j, i)$. These permutations are automorphisms of a toroidal mesh network.

5.2.1 Group of Symmetries for Graphs of Different Sizes

Proposition 5.1. *Consider S to be the set of graphs with different sizes and the same architecture (topology). If $\mathcal{A}$ and $\mathcal{B}$ are groups of permutations acting on S, then these groups are disjoint (their intersection is empty) if the intersection of the subsets of elements of S that they act on is empty.*

The above proposition induces that $\mathcal{A}$ and $\mathcal{B}$ act on different parts of S. This proposition is clearly observed for graph isomorphisms as graph permutations, since two graphs with the same topology but different numbers of nodes are not isomorphic.

With this proposition, we can define a product on disjoint groups of symmetries. Each group is a set of symmetries for graphs of the same topology and size. Groups are disjoint, because they act on graphs of different sizes, e.g. graphs of rings with one, two, three, and n nodes; or graphs of 1×1, 1×2, 2×2, ..., and $(n + i) \times (n + j)$ toroidal meshes where $0 \leqslant i, j$ and $n + i, n + j \leqslant maxsize$, where $maxsize$ is an upper bound on the number of processes that can be added to the system. If we prove that this product forms a group itself, then this group can be used as a generalized group of symmetries for the topology graphs of different sizes. If $\mathcal{A}$ and $\mathcal{B}$ are groups of symmetries under functional composition for graphs of a specific topology and differing sizes, then their product is defined by:

$$\mathcal{A}\mathcal{B} = \{(\pi, \psi) : \pi \in \mathcal{A} \text{ and } \psi \in \mathcal{B}\}.$$

For example, consider a set that is the disjoint union of toroidal meshes G and H of size 2×2 and 2×3, respectively. If π is a symmetry of graph G and ψ is a symmetry of graph

H, (π, ψ) can be defined such that π is an identity element, mapping G to itself, while ψ is any of the vertical rotation, horizotal rotation, and flip for H.

We show that the product of two groups of symmetries $\mathcal{A}$ and $\mathcal{B}$ on the disjoint union of graphs with the same topology but different sizes satisfies the axioms for a group. Thus, for permutations $e_{\mathcal{A}}, \pi, \pi' \in \mathcal{A}$ and permutations $e_{\mathcal{B}}, \psi, \psi' \in \mathcal{B}$, where $e_{\mathcal{A}}$ is the identity element of group $\mathcal{A}$ and $e_{\mathcal{B}}$ is the identity element of group $\mathcal{B}$, we show these axioms as:

1. The product has the identity element $(e_{\mathcal{A}}, e_{\mathcal{B}})$.
2. For (π, ψ), (π', ψ'), the functional composition of these two is defined as: (π, ψ) o $(\pi', \psi') = (\pi \text{ o } \pi', \psi \text{ o } \psi')$. By defining the composition on the product, the associativity follows from that of the two group products.
3. For any (π, ψ), there is a (π^{-1}, ψ^{-1}) such that (π, ψ) o $(\pi^{-1}, \psi^{-1}) = (\pi^{-1}, \psi^{-1})$ o $(\pi, \psi) = (\pi \text{ o } \pi^{-1}, \psi \text{ o } \psi^{-1}) = (e_{\mathcal{A}}, e_{\mathcal{B}})$

Note that $\mathcal{A}$ is isomorphic to the subgroup of elements $(\pi, e_{\mathcal{B}})$, where $e_{\mathcal{B}}$ is the identity element of $\mathcal{B}$ and $\pi \in \mathcal{A}$. Similarly, $\mathcal{B}$ is isomorphic to the subgroup of elements $(e_{\mathcal{A}}, \psi)$.

Given a finite number of groups $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n$ the elements of product $\mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_n$ are defined as: $\pi_1, ..., \pi_n$ where $\pi_i \in \mathcal{A}_i$ and the product group operation is defined componentwise. We use this group as a generalized group of symmetries for topologies of different sizes.

5.2.2 GTS Symmetry

For a specific topology, consider a set composed of a disjoint union of graphs with different sizes. Suppose we use $\mathcal{A}_i$ to show a group of graph symmetries, where i denotes size of the graph. The number of groups is finite as we work with GTS models with an upper-bound $maxsize$ on the number of graph nodes. Γ is defined as a new generalized group of symmetries built from the product of groups of permutations of graphs with different sizes. Each element of Γ is a tuple $(\pi_1, \pi_2, ..., \pi_n)$ where $\pi_i \in \mathcal{A}_i$. Each π_i can be an identity permutation indicated as e_i, which is a morphism that maps each graph of size i to itself, where $1 \leq i \leq maxsize(\mathcal{G})$. Note that the group $\mathcal{A}_k$ is isomorphic to the subgroup of elements $(e_1, e_2, ..., \pi_k, ..., e_n)$; therefore, for simplicity we indicate $(e_1, e_2, ..., \pi_k, ..., e_n)$ as π_k from now on.

In symmetry for Kripke models, each transition can relate to another transition in the model when the source and target states of these transitions, respectively, are permutations of each other. Unlike symmetry for Kripke models, in GTS symmetry each transition may be mapped to a sequence of transitions defined as a path in the model. The source and target state graphs of the transition and the path, respectively are isomorphic to each other.

Definition 5.1 (GTS Symmetry). *A GTS $\mathcal{G} = \langle S, T, I \rangle$ is symmetric with respect to the set of graph permutations Γ if:*

1. *For all $s_1, s_2 \in S$, where G_{s_1} is an n-node graph and G_{s_2} is an m-node graph, if t is a transition in T such that $t : s_1 \rightarrow s_2$, then for π_n, an n-node symmetry in Γ, there is a path $p \in T$, $p : \pi_n(s_1) \rightsquigarrow \pi_m(s_2) \in T$ where π_m is an m-node symmetry in Γ, and the length of p is at least one. Furthermore, $G_{s_1} \cong \pi_n(G_{s_1})$ and $G_{s_2} \cong \pi_m(G_{s_2})$.*
2. *For all $s_0 \in I$ where G_{s_0} is an n-node graph and for all $\pi_n \in \Gamma$, $G_{s_0} \cong \pi_n(G_{s_0})$ and $\pi_n(G_{s_0}) \in I$.*

GTS symmetry differs from architectural symmetry defined for fixed-size systems in [TW09], because in the case that G_{s_1} and G_{s_2} are of the same size, then $m = n$ and $\pi_n = \pi_m$, which means that we have the same permutation for graphs with the same number of nodes, and in this case the path p would be of length one. In addition, the set of symmetries in architectural symmetry differs from those in GTS symmetry, which are based on graph isomorphisms.

Our methods are applicable when evolution of the system does not change the architecture describing the model structure. For example, the basic building block of a toroidal mesh is a ring, and the toroidal mesh evolves by the addition of these building blocks. Therefore, the dynamic evolution is done by adding a certain number of k nodes to form a new vertical or horizontal ring to keep a balanced toroidal mesh network. Therefore, in toroidal mesh, $m = n$ (when the toroidal mesh is not dynamic), or $m = n + k$ (when k nodes are added), or $m = n - k$ (when k nodes are deleted).

We use graph isomorphism to build a bisimilar quotient of a GTS model. It is notable that graph isomorphism requires that graphs be of the same size and structure. We can use graph isomorphism as an equivalence relation on a GTS model with state graphs of different sizes. Thus, in state-space reduction, we are looking to cut down the number of isomorphic state graphs belonging to the same equivalence class that are represented during verification.

5.3 GTS Bisimulation

Using graph isomorphism, we now define GTS bisimulation, and then give an algorithm to generate a reduced bisimilar quotient of a GTS model. Isomorphism provides a strong equivalence relation for generating the quotient, because the same set of transformation rules are applicable to a state in the quotient and the isomorphic state in the original model.

Definition 5.2 (GTS Bisimulation). *Given two GTSs $\mathcal{G}_1 = \langle S_1, T_1, I_1 \rangle$ and $\mathcal{G}_2 = \langle S_2, T_2, I_2 \rangle$, a relation $\sim \subseteq S_1 \times S_2$ is a GTS bisimulation if $s_1 \sim s_2$ implies:*

1. $G_{s_1} \cong G_{s_2}$.
2. *For every $t_1 \in T_1$, $t_1 : s_1 \rightarrow s_1'$, there is a path $p_2 \in T_2$ of length at least one, such that $p_2 : s_2 \rightsquigarrow s_2'$ and $s_1' \sim s_2'$.*
3. *For every $t_2 \in T_2$, $t_2 : s_2 \rightarrow s_2'$, there is a path $t_1 \in T_1$ of length at least one such that $p_1 : s_1 \rightsquigarrow s_1'$ and $s_2' \sim s_1'$.*

5.3.1 Generating A Bisimilar Quotient

In this section, we present an algorithm for generating a symmetry-reduced GTS model of a dynamically evolving multi-processor system. This algorithm (*cf* [ES96]) provides an on-the-fly generation of the symmetry-reduced model of a GTS-based labelled-transition system. The algorithm may provide an exponential savings in the cost of system analysis

```
GenerateQuotient(state $s_0$, $T$, int n)
Input: $s_0$: initial state-graph labelling, $T$: set of GTS rules, $n$: initial number
       of processes
Output: $E$: equivalence classes of states, $R$: quotient transition relation
$E[1].st \leftarrow s_0$
$CurrentState \leftarrow 1$, $LastState \leftarrow 1$
// loops over E to apply the transformation rules
while $CurrentState \leq LastState$ do
  forall $r \in T$ applicable to $E[CurrentState].st$ do
    $temp \leftarrow Apply(r, E[CurrentState].st)$
    $\bar{s} \leftarrow temp.st$
    $\bar{n} \leftarrow temp.n$
    $stateFound \leftarrow false$
    // checks if the transformed state is a permutation of the
       existing representative states
    for $i \leftarrow 1$ to $LastState$ do
       // finds the equivalence class based on the graph size
      if $E[i].n = \bar{n}$ then
        if $\bar{s} = E[i].st$ or IsAPermutation($E[i].st$, $\bar{s}$, $E[i].n$) then
          $stateFound \leftarrow true$
          $AddTransition(R, CurrentState, i)$
          exit for loop
    end
    // enters the newly found equivalence class in E
    if $stateFound = false$ then
      $LastState \leftarrow LastState + 1$
      $E[LastState].st \leftarrow \bar{s}$
      $E[LastState].n = \bar{n}$
      $AddTransition(R, CurrentState,$
                     $LastState)$
  end
  $CurrentState \leftarrow CurrentState + 1$
end
return $E$, $R$
```

Figure 5.2: Quotient generation algorithm

```
  boolean IsAPermutation(state s, state $\bar{s}$, int n)
  Input: $s, \bar{s}$: state graph labelling, $n$: number of processes
  Output: true if s, $\bar{s}$ are permutations of each other, else false
1 $i \leftarrow 1$, $s_1 \leftarrow s$
2 repeat
3     if $s_1 = \bar{s}$ then
4         return true
5     $s_1 \leftarrow circularPermute(s_1)$
      $i \leftarrow i + 1$
6 until $i = n$
7 return false
```

Figure 5.3: Graph permutation for rings

for fully symmetric GTS models, but for GTS models with some symmetry we get a polynomial-size reduction.

The algorithm GenerateQuotient in Figure 5.2 accepts a set of graph transformation rules, an initial-state graph labelling, and the initial number of processes as input. As output, it generates the set E: the representatives of the equivalence classes of state graphs, and a table R: the quotient transition relation. Each element in E consists of a single representative state graph (st), and the number of processes in that state graph (n). Table R is a two-dimensional table consisting of pointers to table E. There is a transition between each state in $E[i]$ and the state in $E[R[i, j]]$, where j is an index iterating over all transitions of the state in $E[i]$. By keeping track of the number of processes in each representative state: $E[i].n$, our algorithm works correctly for dynamic architectures in which processes can be added or deleted in the execution path.

In line 10, the algorithm checks that two state graphs with the same size are a permutation of each other. Determining if two graphs are permutations of each other needs graph isomorphism checking, which is a hard problem for unlabelled graphs, but it can be shown to have a polynomial complexity for deterministic labelled graphs [Ren06, GJ79]. Also, McKay [McK81] has developed an algorithm for graph isomorphism that works quite well in practice, handling graphs with up to millions of nodes.

The algorithm IsAPermutation in Figure 5.3 has been specialized for ring networks. The algorithm iterates over permutations to find the right permutation, and it can be specialized for different topologies. For example, for ring networks, the permutations are circular ones. For a toroidal mesh, they are appropriate horizontal or vertical rotations, or flips.

As an example, we have implemented the GTS model of mutual exclusion for both a dynamic 2×2 toroidal mesh and a dynamic token ring with three processes. The Labelled Transition System (LTS) generated from the ring GTS has been illustrated in Figure 5.4. Due to space limitations, we have restricted our attention to show an LTS of a fixed-size ring here. Running the above algorithm on our ring example creates a reduced LTS model, illustrated in Figure 5.5. This reduced model has 12 states in comparison to the original model with 36 states. Details about the implementation and these figures are presented in Section 5.5.

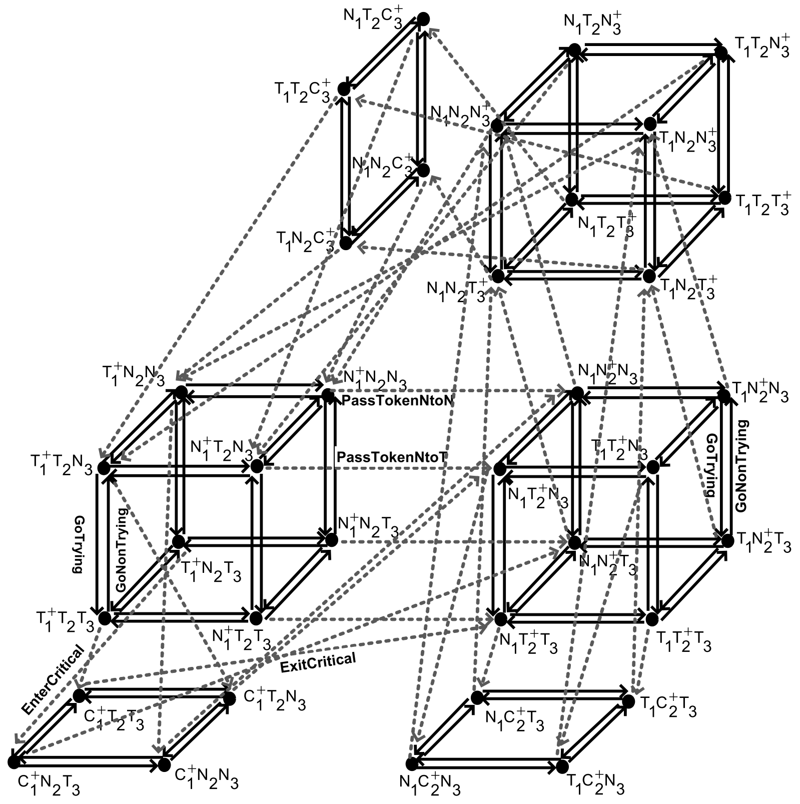


Figure 5.4: LTS of a ring with three processes.

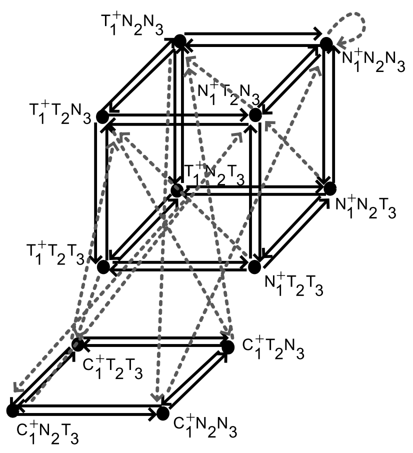


Figure 5.5: Reduced LTS of a ring.

Theorem 5.1. *Let $\mathcal{G} = \langle S, T, I\rangle$ be a GTS and symmetric with respect to the set of graph permutations Γ, and $\bar{\mathcal{G}} = \langle \bar{S}, \bar{T}, \bar{I}\rangle$ be the quotient of $\mathcal{G}$, then $\mathcal{G}$ and $\bar{\mathcal{G}}$ are GTS-bisimilar: $\mathcal{G} \sim \bar{\mathcal{G}}$.*

Proof. Consider π_n, $\pi_m \in \Gamma$ as graph permutations for a set of state graphs with different numbers of nodes. The proof considers two claims: 1) for every graph transformation $\bar{s_0} \rightarrow \bar{s_1} \in \bar{T}$, there is a corresponding path $p = s_0 \rightsquigarrow s_1$ in $\mathcal{G}$, and 2) for every $s_0 \rightarrow s_1 \in T$, there is a corresponding path $\bar{p} = \bar{s_0} \rightsquigarrow \bar{s_1}$ in $\bar{\mathcal{G}}$. We prove the first claim, and the other follows similarly. The proof for each claim is broken into two cases: one for transformations $\bar{s_0} \rightarrow \bar{s_1}$ that do not add or delete components (nodes) to or from the start graph $\bar{s_0}$ of size n. The second case considers a transformation that changes the number of components in the source state graph. For the second case, we only consider the addition of components, as proof for the deletion is similar.

Case 1: Choose an arbitrary reachable state $s_0 \in S$ such that $G_{s_0} \cong G_{\bar{s_0}}$. Using on-the-fly generation of the quotient, we know that there exists a transition $\bar{s_0} \rightarrow \bar{s_1} \in \bar{T}$ such that $\bar{s_0}$ and $\bar{s_1}$ are equivalence classes of state graphs. Thus, there is a graph $u \in S$ that belongs to the equivalence class of $\bar{s_0}$ and there is a graph $v \in S$ that belongs to the equivalence class of $\bar{s_1}$. Therefore, $u \rightarrow v \in T$. Thus, $G_u \cong G_{\bar{s_0}}$ and $G_v \cong G_{\bar{s_1}}$. Since $G_{s_0} \cong G_{\bar{s_0}}$ and $G_u \cong G_{\bar{s_0}}$, by transitivity $G_{s_0} \cong G_u$. Now let G_{s_1} be isomorphic to a permutation of graph v, i.e. $G_{s_1} \cong \pi_n(G_v)$ which implies $G_{s_1} \cong G_v$ and because we had

$G_v \cong G_{\bar{s_1}}$, thus $G_{s_1} \cong G_{\bar{s_1}}$. From $u \to v \in T$, $G_{s_0} \cong G_u$, and $G_{s_1} \cong G_v$ we deduce $\pi_n(u) \to \pi_n(v) = s_0 \xrightarrow{r} s_1 \in T$. Inductively, we can prove for each $\bar{s_i} \to \bar{s_{i+1}} \in \bar{T}$, there is a transformation $s_i \xrightarrow{r} s_{i+1} \in T$.

Case 2: In $\bar{s_0} \to \bar{s_1} \in \bar{T}$, we know that $G_{\bar{s_0}}$ is of size n and $G_{\bar{s_1}}$ is of size m, where $m > n$. Choose an arbitrary state $s_0 \in S$ such that $G_{s_0} \cong G_{\bar{s_0}}$. Since $\bar{s_0} \to \bar{s_1} \in \bar{T}$, then based on the quotient generation algorithm, there is a transformation rule $u \xrightarrow{r} v \in T$ in GTS $\mathcal{G}$ in which rule r applies to the graph G_u of size n and transforms it to a graph G_v with size m. Thus, we have isomorphisms $G_{\bar{s_0}} \cong \pi_n(G_u)$ and $G_{\bar{s_1}} \cong \pi_m(G_v)$. Since $\mathcal{G}$ is GTS-symmetric, then for each transition $u \to v \in T$ there exists $\pi'_n(G_u) \rightsquigarrow \pi'_m(G_v) \in T$, and since permutation is based on isomorphism then every permutation of a graph is isomorphic to it, so $\pi'_n(G_u) \cong \pi_n(G_u)$. From $G_{s_0} \cong G_{\bar{s_0}}$, $G_{\bar{s_0}} \cong \pi_n(G_u)$, and $\pi'_n(G_u) \cong \pi_n(G_u)$ we have $G_{s_0} \cong \pi_n(G_u)$, which means that s_0 is a permutation of graph u and isomorphic to it. Therefore, $s_0 \rightsquigarrow \pi'_m(v)$. Let s_1 be the permutation of graph v; hence, $\pi'_m(G_v) \cong G_{s_1}$. We had $G_{\bar{s_1}} \cong \pi_m(G_v)$, also we know all permutations of a graph are isomorphic with each other, thus $\pi'_m(G_v) \cong \pi_m(G_v)$ and from these isomorphisms we have $G_{\bar{s_1}} \cong G_{s_1}$, and conclude $s_0 \rightsquigarrow s_1$. Inductively, we can prove for each $\bar{s_i} \to \bar{s_{i+1}} \in \bar{T}$ that there is a path in T. □

As a result, we have a theorem about property satisfaction of EFf and ¬EFf (reachability properties), where f is a propositional formula. In GTS-bisimilar models, a transition matches with a path; therefore, neither *next-time* (X) nor *until* (U) operators can be expressed in properties.

Theorem 5.2. *Let ϕ be an EF formula over a set of fully symmetric atomic proposition graphs. For the graph transition system $\mathcal{G}$, and its quotient $\bar{\mathcal{G}}$ and the property ϕ and state graphs $s_1 \in \mathcal{G}$ and $\bar{s_1} \in \bar{\mathcal{G}}$, where $s_1 \sim \bar{s_1}$ we have $\mathcal{G}, s_1 \models \phi$ iff $\bar{\mathcal{G}}, \bar{s_1} \models \phi$.*

Proof idea. This theorem is a direct consequence of exploiting GTS symmetry, and the proof is done using the bisimulation between the GTS $\mathcal{G}$ and its quotient $\bar{\mathcal{G}}$, and it is similar to the proof given in [TW09].

To extend the properties that can be verified on the GTS-bisimilar reduced model, we require that the system be *well-architected*. This requirement is used to show that reachable state graphs with differing number of nodes are reachable from each other.

Definition 5.3 (*cf.* [TW09]). *A GTS $\mathcal{G} = \langle S, T, I \rangle$ is well-architected, if $\mathcal{G}$'s initial states are reachable from each other and also from all other reachable states.*

This characteristic is often present in reactive systems such as communication protocols and IP-telephony. Telephony features as finite-state processes are continuously assembled in a call connection based on the current input, then the features are executed and terminated after the execution by returning to their initial states. Another example of a well-architected system model is the multi-process system model in Figure 2.2. In this figure, the initial state(N) is reachable from any other reachable state. Having a well-architected model, it has been shown by [TW09] that any two reachable states in the model are reachable from each other. With this requirement, we are able to verify EF-CTL formulas. EF-CTL formulas contain all atomic propositions, and they are closed under Boolean connectives ($\neg, \vee, \wedge$) and EF temporal formulas. For example, both EFf and AGEF are EF-CTL formulas.

5.4 Vertex Bisimulation

In this section, we introduce vertex bisimulation to be used for GTSs that are not fully symmetric. Vertex bisimulation enables an exponential reduction with respect to full symmetry group for GTS models. The example below shows the motivation for this type of GTS bisimilarity.

In a 2×2 toroidal mesh, the symmetry reduction based on graph permutations (e.g. horizontal and vertical rotations in ring blocks) reduces the four state graphs with node labelling $N_{11}^{+}N_{12}N_{21}T_{22}$, $N_{11}N_{12}^{+}T_{21}N_{22}$ (a rotation on rows of the torus), $N_{11}T_{12}N_{21}^{+}N_{22}$ (a rotation on columns of the torus), and $T_{11}N_{12}N_{21}N_{22}^{+}$ (a flip on the torus) to one state using rotation and flip permutations. Also the states $N_{11}^{+}N_{12}T_{21}N_{22}$, $N_{11}N_{12}^{+}N_{21}T_{22}$, $T_{11}N_{12}N_{21}^{+}N_{22}$, and $N_{11}T_{12}N_{21}N_{22}^{+}$ will be reduced to another state using the same set of permutations (horizontal and vertical rotations and flip). Observing these two sets of states, though they are different, we still see similarities between them. In all states of the two sets, there is one process in the *Trying* mode, and three others in the *Non-trying* mode. Therefore, if we can exploit this similarity of states, then all of these eight states can be reduced to one.

Definition 5.4 (Vertex Bisimulation)**.** *For GTSs $\mathcal{G}_1 = \langle S_1, T_1, I_1 \rangle$ and $\mathcal{G}_2 = \langle S_2, T_2, I_2 \rangle$ a relation $\sim^v \subseteq S_1 \times S_2$ is a vertex bisimulation if $s_1 \sim^v s_2$ implies:*

1. *G_{s_1} and G_{s_2} have the same set of vertices.*
2. *for every $t_1 \in T_1$, $t_1 : s_1 \to s_1'$, there is a path $p_2 : s_2 \rightsquigarrow s_2' \in T_2$ and $s_1' \sim^v s_2'$.*
3. *for every $t_2 \in T_2$, $t_2 : s_2 \to s_2'$, there is a path $p_1 \in T_1$ such that $p_1 : s_1 \rightsquigarrow s_1'$ and $s_2' \sim^v s_1'$.*

From a GTS-symmetric model with respect to full symmetry group, we derive a vertex-bisimilar quotient. Thus, we can apply all permutations and obtain full symmetry reduction resulting in an exponential reduction. To be able to gain full symmetry reduction without the application of the large set of all permutations, there are techniques that allow the representation of full symmetry-reduced state spaces by a program translation into a symmetry-reduced program text [ET99, BMWK09, TW09]. Vertex bisimulation for GTS models is comparable to safety-bisimulation for Kripke models [TW09], but unlike safety-bisimulation it can be used for dynamic graph models. In the following theorem, we show the vertex-bisimilarity of a model and its quotient. The proof of this theorem is similar to the proof of Theorem 5.1.

Theorem 5.3. *Let $\bar{\mathcal{G}} = \langle \bar{S}, \bar{T}, \bar{I} \rangle$ be the quotient model of a GTS-symmetric system $\mathcal{G} = \langle S, T, I \rangle$, then $\mathcal{G}$ is vertex-bisimilar to $\bar{\mathcal{G}}$.*

5.4.1 Property Preservation

If we prove that the generated quotient model of a GTS-symmetric system is vertex-bisimilar to the original model, then we can use the quotient model to prove interesting properties of the system. The important fact in the vertex-bisimilarity proof is that there is an explicit requirement in GTS models that these models preserve the architecture of the system. For example, if the initial state graph architecture is a ring, then this architecture is preserved in all state graphs of the model and in all the state graphs represented in the symmetry reduced model. Even if the structure dynamically evolves, we require that the evolution of components preserve the overall system structure.

As stated in [ES97, TW09], one of the problems of verifying properties on the quotient models is that the property should have symmetric atomic propositions. Expressing Boolean expressions of atomic propositions as graphs and using graph satisfaction (Definition 4.4) [LT09] provides an abstraction on the process indices that solves this problem. The reason is that when we use an REG (with Kleene-star labels over the edges) in an atomic proposition and a generic node that represents, for example, any of the processes appearing in the proposition, then we do not need to specify each symmetric part of the atomic proposition explicitly. For example, in a model with three processes, an atomic proposition for expressing that at least one of the processes is in the *Critical* state is: $Critical_1 \vee Critical_2 \vee Critical_3$. Figure 5.6-a illustrates such an expression in which there is set of processes with one process being in the *Critical* state and connected to at least one other process. The condition "at least one" has been modelled as an edge labelled with $Connected^+$ between two processes, and a process node is used to abstract any of the existing processes. Thus, we avoided to explicitly show each of the three processes being in the *Critical* state.

As presented in Definition 4.1, we have used the regular-expression graph in which edges may be labelled with a Kleene-star operator over the set of labels. Therefore, all formulas with the existential process quantifier form, $\vee_i$, can be abstractly modelled as a proposition graph with nodes being an abstraction of process indices. Also, the universal process quantifier form, $\wedge_i$, in a graphical notation, is implicitly presented as all the process nodes that participate in the $\wedge_i$ formula connected together. For instance, in the property $\neg \mathrm{EF}(\exists i \neq j : Critical_i \wedge Critical_j)$ in a token ring, the Boolean expression of propositions can be expressed as a graph illustrated in Figure 5.6-b. In this figure, two different processes are presented to be in the *Critical* state and these processes are connected to zero or more other processes in any status (labelled as the regular expression $Connected^*$).

Thus reachability properties and all the properties that can be expressed in terms of EF, such as AG ϕ which is equal to $\neg$ EF $\neg\phi$, are verifiable on the symmetry-reduced GTS model. For these properties, we prove that for a GTS and its quotient that are vertex-bisimilar, they both satisfy the same properties.

Theorem 5.4. *Let $\mathcal{G} = \langle S, T, I \rangle$ be a GTS and $\bar{\mathcal{G}} = \langle \bar{S}, \bar{T}, \bar{I} \rangle$ be the quotient of $\mathcal{G}$ and vertex-bisimilar to it, $\mathcal{G} \sim^v \bar{\mathcal{G}}$. For state graphs $s_1 \in S$ and $\bar{s_1} \in \bar{S}$, where $s_1 \sim^v \bar{s_1}$ we have $\mathcal{G}, s_1 \models \phi$ iff $\bar{\mathcal{G}}, \bar{s_1} \models \phi$ where ϕ is an EF formula.*

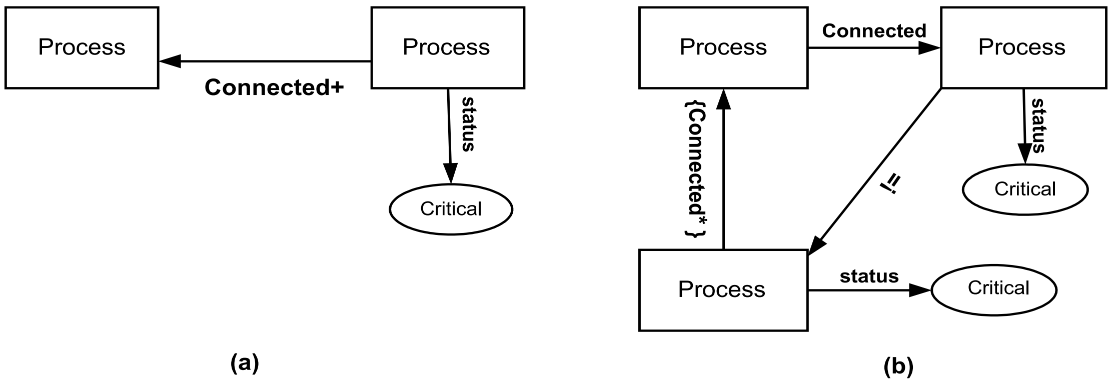


Figure 5.6: Two atomic proposition graphs.

To prove this theorem, first in the lemma below, we show that there is a matching path between two vertex-bisimilar GTSs for GTS-symmetric models.

Lemma 5.1. *Let $\mathcal{G} = \langle S, T, I \rangle$ be a symmetric GTS and $\bar{\mathcal{G}} = \langle \bar{S}, \bar{T}, \bar{I} \rangle$ be its vertex-bisimilar GTS, $\mathcal{G} \sim^v \bar{\mathcal{G}}$. For $s_1 \in S$ and $\bar{s_1} \in \bar{S}$, if $s_1 \sim^v \bar{s_1}$ then for any GTS derivation in $\mathcal{G}$, $s_1 \xrightarrow{r^*} s_m$, there is a derivation in $\bar{\mathcal{G}}$, $\bar{s_1} \xrightarrow{r^*} \bar{s_n}$, and vice versa.*

Proof. It is notable that there may not be a one-to-one correspondence between transformations of these two derivations, which means that the lengths of the two derivations may not be the same. We show the proof for ($\Leftarrow$), and the other direction will follow because $\mathcal{G}$ and $\bar{\mathcal{G}}$ are vertex-bisimilar.

For $\bar{p} : \bar{s_1} \xrightarrow{r^*} \bar{s_n}$ in $\bar{\mathcal{G}}$, we prove that there is a GTS derivation $p : s_1 \xrightarrow{r^*} s_m$ in $\mathcal{G}$ such that $s_m \sim^v \bar{s_n}$. The proof is shown by breaking the derivation $\bar{p}$ into individual transformations and matching each graph transformation in the derivation $\bar{p}$ to a sequence of transformations in p. Later we match the concatenated transformations in $\bar{p}$ to the concatenated sequence of transformations in p.

For the first transition in $\bar{p}$, if the length of the GTS derivation p is zero, then $s_1 = s_m$, and we have a mapping to a path of length zero. If the length of the GTS derivation p is greater than or equal to one, then we have $s_1 \sim^v \bar{s_1}$ and based on Definition 5.4, for one transition $\bar{s_1} \rightarrow \bar{s_2}$ in $\bar{p}$, there is a derivation in p of length at least one, where p is a path from the first state to the i-th state, $p : s_1 \rightsquigarrow s_i$,thus $\bar{s_2} \sim^v s_i$. We proved that for the first transformation in $\bar{p}$, there is a sequence of transformations in $p : s_1 \xrightarrow{r^i} s_i$ where $\bar{s_2} \sim^v s_i$. The same reasoning can be used for the second and subsequent transformations, e.g. $\bar{s_2} \rightarrow \bar{s_3}$ is matched to a path from s_i to s_j in p.

We now use induction. As to the hypothesis, consider for a sequence of k transformations in $\bar{p} : \bar{s_1} \xrightarrow{r^k} \bar{s_k}$, there is a sequence of l transformations in $p : s_1 \xrightarrow{r^l} s_l$, such that $s_1 \sim^v \bar{s_1}$ and $s_l \sim^v \bar{s_k}$. Based on the vertex bisimulation definition, for the transformation $\bar{s_k} \rightarrow \bar{s}_{k+1}$ in $\bar{\mathcal{G}}$, there is a path $s_l \xrightarrow{r^*} v$ in $\mathcal{G}$, where $\bar{s_k} \sim^v s_l$, and $\bar{s}_{l+1} \sim^v v$, let v be s_{l+1}. Therefore, for $\bar{p} : \bar{s_k} \xrightarrow{r} \bar{s}_{k+1}$ in $\bar{\mathcal{G}}$ there is a derivation $p : s_l \xrightarrow{r^*} s_{l+1}$ in $\mathcal{G}$. We consider the application of the first k transformations and the $k+1$th transformation in $\bar{\mathcal{G}}$ as one GTS derivation: $\bar{s_1} \xrightarrow{r^*} \bar{s_{k+1}}$, and also the first l sequences of transformations and the $l+1$th transformation in $\mathcal{G}$ as the derivation $p : s_1 \xrightarrow{r^*} s_{l+1}$. Let $k+1 = n$ and $l+1 = m$. Thus, we have matched the two derivations. $\square$

Proof of Theorem 5.4. To prove this theorem, we use the fact that $\mathcal{G}$ is symmetric, to ensure the preservation of architecture in states of $\mathcal{G}$ and its quotient, even though s_1 and $\bar{s_1}$ only have the same set of vertices. The proof is given for different cases of ϕ. It is sufficient to show the proof for one direction ($\Rightarrow$). The other direction is similar.

Atomic Propositions The propositional formula is defined per Definition 4.5 with an abstraction on process indices. Therefore, without considering specific indices, if the formula is true for s_1, it is symmetrically true for any other communication graph of processes with the same set of local states. Since s_1 and $\bar{s_1}$ are vertex-bisimilar, they have the same node labelling or the same set of possible local states, and both satisfy the same formula.

EF formula From $\mathcal{G}, s_1 \models \text{EF } \varphi$, we deduce that there is a derivation $p : s_1 \xrightarrow{r^*} u$ in $\mathcal{G}$, where u is a state graph that satisfies the proposition graph φ. Since $s_1 \sim^v \bar{s_1}$ and based on Lemma 5.1, we know that for each derivation p in $\mathcal{G}$, there is a matching derivation $\bar{p} : \bar{s_1} \xrightarrow{r^*} v$ in $\bar{\mathcal{G}}$ such that $u \sim^v v$. Therefore, each property that is satisfied in u is satisfied in v as well, $\bar{\mathcal{G}}, v \models \varphi$, and v is a state along the path starting at $\bar{s_1}$. Hence, $\bar{\mathcal{G}}, \bar{s_1} \models \text{EF } \varphi$. $\square$

Based on the above theorem, we can use the vertex-bisimilar reduced GTS model of a system to prove interesting properties of it.

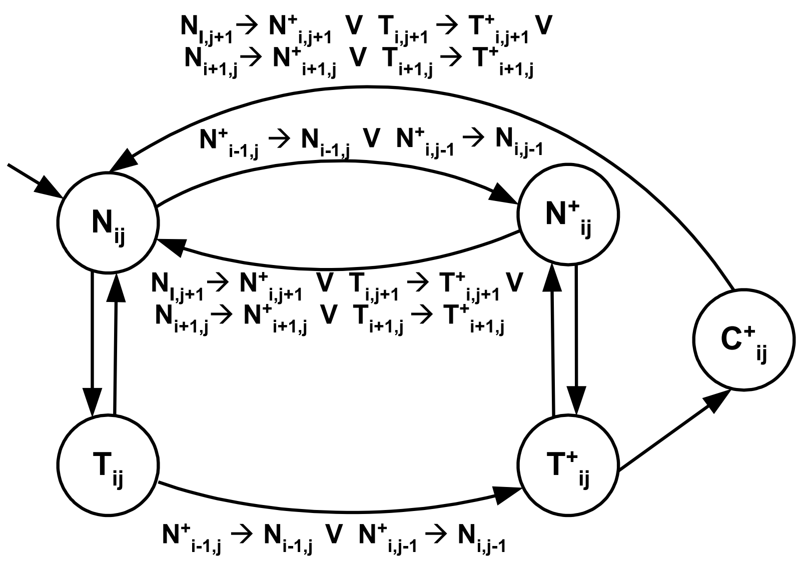

$N_{I,j+1} \rightarrow N^+_{i,j+1}$ V $T_{i,j+1} \rightarrow T^+_{i,j+1}$ V
$N_{i+1,j} \rightarrow N^+_{i+1,j}$ V $T_{i+1,j} \rightarrow T^+_{i+1,j}$
$N^+_{i-1,j} \rightarrow N_{i-1,j}$ V $N^+_{i,j-1} \rightarrow N_{i,j-1}$
N_{ij}
N^+_{ij}
$N_{I,j+1} \rightarrow N^+_{i,j+1}$ V $T_{i,j+1} \rightarrow T^+_{i,j+1}$ V
$N_{i+1,j} \rightarrow N^+_{i+1,j}$ V $T_{i+1,j} \rightarrow T^+_{i+1,j}$
C^+_{ij}
T_{ij}
T^+_{ij}
$N^+_{i-1,j} \rightarrow N_{i-1,j}$ V $N^+_{i,j-1} \rightarrow N_{i,j-1}$

Figure 5.7: Mutual exclusion with single token in toroidal mesh.

5.5 GTS model of an Evolving Toroidal Mesh

In the following, we model mutual exclusion of a toroidal mesh structure as a GTS and show how much reduction we get using GTS symmetry and vertex bisimulation. We consider a toroidal mesh, in which a single token is used to regulate the access to a shared a resource. Figure 5.7 shows the finite-state machine of each process in the toroidal mesh. Each process may be in one of the local states N, N^+, T, T^+, C (the process is currently accessing the shared resource). In the toroidal mesh grid, each process node has two incoming edges and two outgoing edges, thus the token can be passed to two processes. Each process can move freely between local states N and T, and also between N^+ and T^+. A process in the i, j position can access the token, if the token is possessed by the neighbor at $i-1, j$ or $i, j-1$ position, and that neighbor is ready to release the token. A similar situation occurs for releasing the token by a process. These are indicated by simultaneous transitions in Figure 5.7.

The GTS model of a dynamic 2×2 toroidal mesh has been depicted in Figures 5.8 and 5.9. With an appropriate initial state for a token ring network, this implementation can also be used for modelling dynamic token-ring networks. In this model, the local states of processes are represented by attributes. In the communication graph of processes, each local state of a process is presented by four attributes: the status of the process, which takes the values of *Trying, Non-trying*, and *Critical*; the row and column numbers of the process; and a Boolean attribute that determines whether the process owns a single token.

The GoTrying rule applies to a state graph of a toroidal mesh or ring with a process in *Non-trying* mode, and any number of processes in arbitrary modes, and changes the process status to *Trying*; and the GoNonTrying rule does the reverse. The EnterCritical and ExitCritical rules change the status of a process that owns a token to *Critical* and *Non-trying*, respectively. This means that the process accessed a shared resource, or just released it and passed the token to another process. The PassTokenNtoT rule applies to a *Non-trying* process that owns the token and passes the token to a *Trying* process. In a toroidal mesh, there may be two matches for this rule, since a process can pass the token to two other processes, but in a ring the token will be passed only to one process in the right side. The rule first checks to see if the process has an attribute token set as true and then changes it to false. Next, it sets the token attribute of the other process to true. The important point for the application of this rule is that only a *Non-trying* process that has the token can pass it to a *Trying* or *Non-trying* process. A *Trying* process cannot pass the token without having the chance to change the status to *Critical*, and that's why there is no PassTokenTtoN rule. Rule PassTokenNtoN acts similar to the rule PassTokenNtoT.

Rules AddProcess and ConnectRings in Figure 5.9 are used to model the growth of toroidal mesh by incremental addition of vertical rings. Similar rules are used for the addition of horizontal rings. The rule AddProcess adds one node to each horizontal ring, and the rule ConnectRings connects those nodes to generate a vertical ring. At first, it connects each newly added process to a process with the same column number and a higher row number. Then for all new vertical rings, it adds the wrap-around connections. Similar rules that are not illustrated here, are used for deletion of processes.

The LTS generated from this GTS for a ring network with an initial state of 3 processes has been illustrated in Figure 5.4. Due to space limitations, we have restricted our attention to show an LTS of a 3-process ring here. To clearly present the symmetric nature of rings visually, we have omitted to show each state as a graph, and instead each state graph is depicted as a black dot with a label that shows the labelling of nodes in that state graph. In the above LTS, each transition shows the transformation of a state graph to another graph through the application of a GTS rule. Again, to avoid a busy LTS, only some of the transitions are labelled with the transformation rule's name. For example, the application of transformation rule PassTokenNtoN to the state graph $N_1^+N_2N_3$ results in the transformation of the state graph to $N_1N_2^+N_3$, which passes the token from the *Non-trying* process 1 to the *Non-trying* process 2 to its right. The initial state of the model is $N_1^+N_2N_3$, although it can be any other state graph of the LTS as well.

As shown in Figure 5.4, the GTS model of the ring network with three processes results in 36 state graphs in its LTS. We can see that the LTS can be divided into three symmetric parts (each consisting of a cube and a plane). We would like to exploit this symmetry to reduce the system's model. Using isomorphism as the equivalence relation on the set of state graphs S, we extract a canonical quotient with 12 state graphs (Figure 5.5). In the LTS of Figure 5.4, there is no difference between dashed transitions and the regular ones, they are only used for clarity of the picture. For example, using the graph isomorphism, in a GTS model, the equivalence class of the state graph $N_1^+N_2T_3$, consists of the state graphs $N_1^+N_2T_3$, $T_1N_2^+N_3$, and $N_1T_2N_3^+$ (using a rotation as an isomorphism on rings). These labellings are the labellings of three state graphs that are isomorphic to each other.

For the toroidal mesh, the LTS generated from the initial 2×2 toroidal mesh evolving to 2×4 has 3072 states. The toroidal mesh benefits from the rotational symmetries of its rings. Therefore, reducing the model generates an LTS with 504 states based on the rotational symmetries for state graphs with 4 (in 2×2 toroidal mesh), 6 (in 2×3 toroidal mesh), and 8 processes (in 2×4 toroidal mesh). Vertex bisimulation further reduces the model to 54 states based on the symmetries for state graphs with architecture of size 4, 6, and 8.

5.6 Related Work

Ip and Dill [ID96], Emerson and Sistla [ES96], and Clarke et al. [CEFJ96] were the first to explore symmetry reduction for systems with a fixed number of similar processes. These methods offered only polynomial reductions for most non-fully symmetric systems; thus, in [ET99, TW09] the authors have addressed those systems, however, those methods do not apply to graph-based models and, furthermore, are restricted to models with a fixed number of components.

Iosif in [Ios02] has addressed symmetry reduction for dynamic heap objects formalized as non-visual semantics. But his focus is on multithreaded programs that may only create new objects and never delete the existing ones. Our approach is also different than regular model checking [BJNT00], which provides abstraction that generally is not an equivalent representation of the original model.

To address the verification of systems with unbounded number of processes of the same type, parameterized model checking [AK86, GS92, ID99] has been used. In this approach, the system is represented by an infinite family of instances, in which each instance has a different size and is finite. Then the verification method verifies each finite-state instance individually. Unfortunately, in general the verification of parameterized systems is undecidable [AK86]. However, there is another approach in verifying a parameterized system, which is to identify a decidable subclass of the system by choosing a small bound on the number of processes. This approach is sound but incomplete, and unlike symmetry reduction does not rely on an equivalence relation and building a quotient. Many methods [CGB86, EK00] using this approach have restrictions on both the type of the system and the properties that is going to be verified. In addition, in these methods, a fully automated solution that works for all system sizes does not exist.

In the area of GTS models, it is only Rensink's [Ren06, Ren08] work that has directly addressed symmetry in GTS models. In [Ren06], a generalized definition of bisimulation is used that does not guarantee isomorphism, while our work uses isomorphism to define graph symmetry. Also, it does not give rise to canonical representation of states, whereas we give an algorithm for generating a GTS-bisimilar quotient.

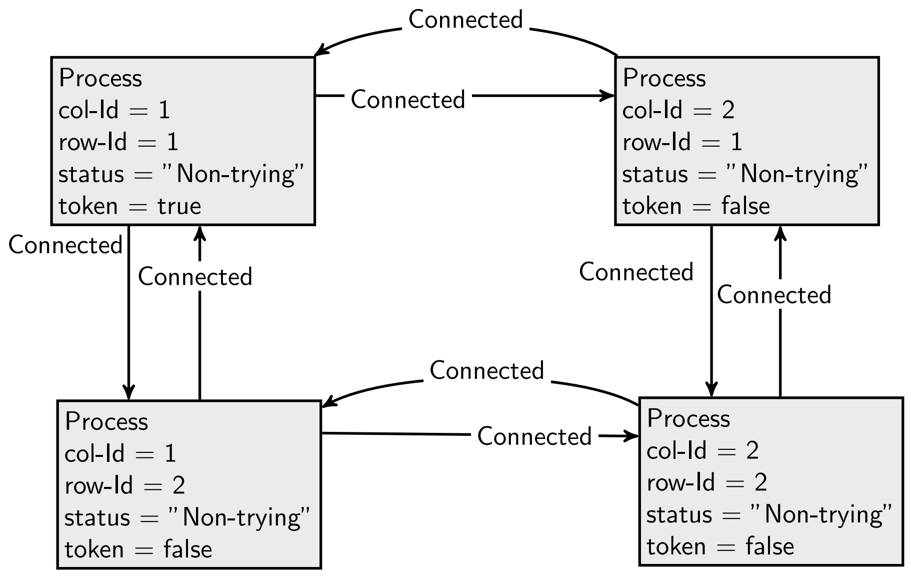
5.7 Conclusion

We have described symmetry-reduction techniques for models that provide explicit visual semantics for dynamic multi-process systems. To generalize notions of symmetry for dynamic GTS models, we defined GTS symmetry and GTS bisimulation. Using these notions, we provided an on-the-fly algorithm for generating a symmetry-reduced GTS model based on graph isomorphism.

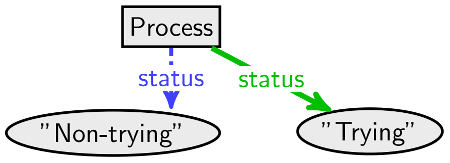
We note that our work requires an upper bound on the number of nodes (components) that can be added to a state, because verification of systems for an arbitrary number of processes is generally undecidable [AK86]. We also have proved that the generated quotient is GTS-bisimilar to the original GTS model, and thus they both satisfy the same set of properties. To achieve better state-space savings for dynamic GTS models that are not fully symmetric, we have defined vertex bisimulation. The vertex-bisimilar GTS model may provide exponential savings over the original model. Vertex bisimulation defines an equivalence relation on state graphs based on their vertices.

We showed that the vertex-bisimilar reduced model can prove an interesting subset of CTL properties satisfied by the original model. This subset includes all the properties expressed with the EF and ¬EF temporal operators. This includes the important class of safety properties that are typically checked in an industrial verification setting. The propositional formula of these properties has been illustrated as a graph. Proposition graphs provide an abstraction on the process indices that take care of the symmetry of propositions. Currently, we are investigating the satisfaction of EF-CTL properties with arbitrary nesting of temporal operators and proposition graphs.

Initial state graph



GoTrying



GoNonTrying

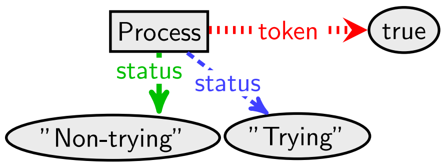
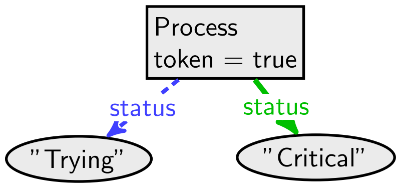


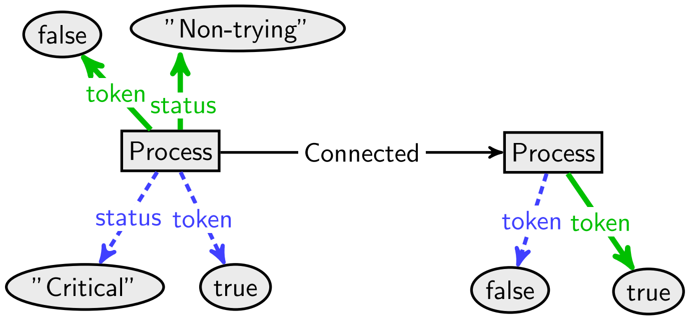
Figure 5.8: Initial state and GTS rules of a 2×2 toroidal mesh.

Transformation rules continued ...

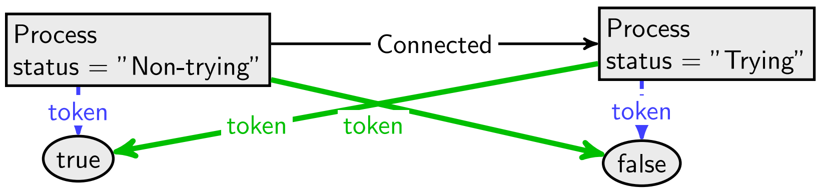
<u>EnterCritical</u>



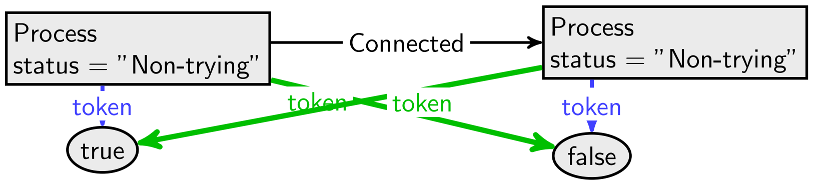
<u>ExitCritical</u>



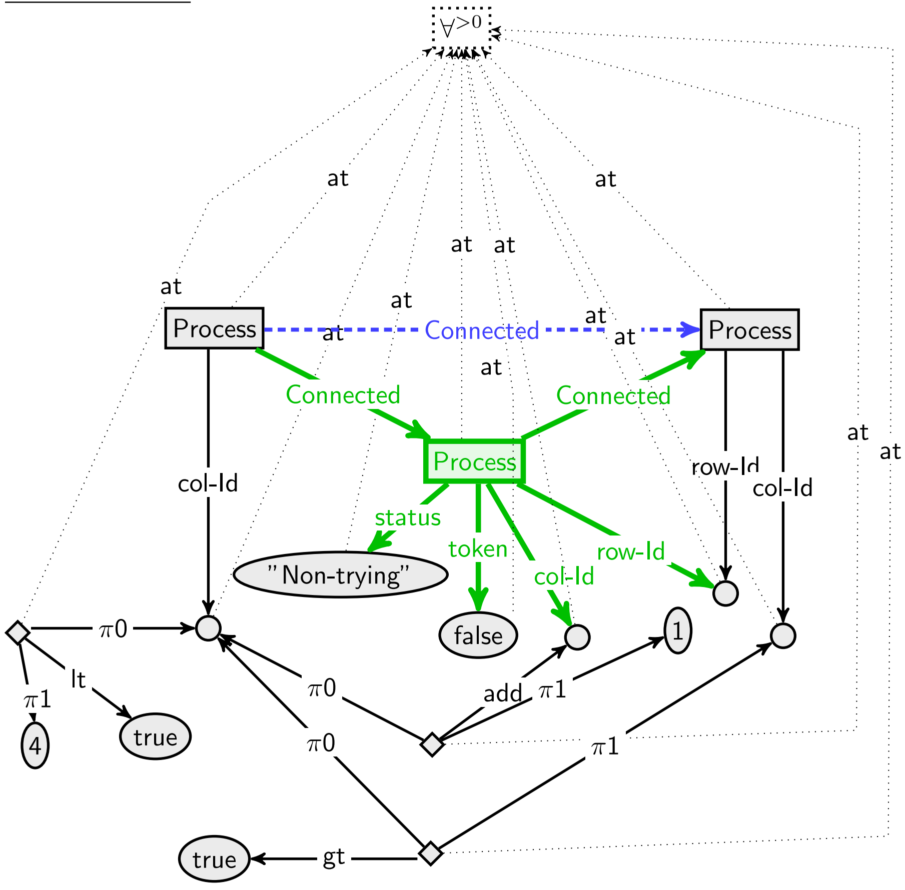
<u>PassTokenNtoT</u>



<u>PassTokenNtoN</u>



AddProcess



ConnectRings

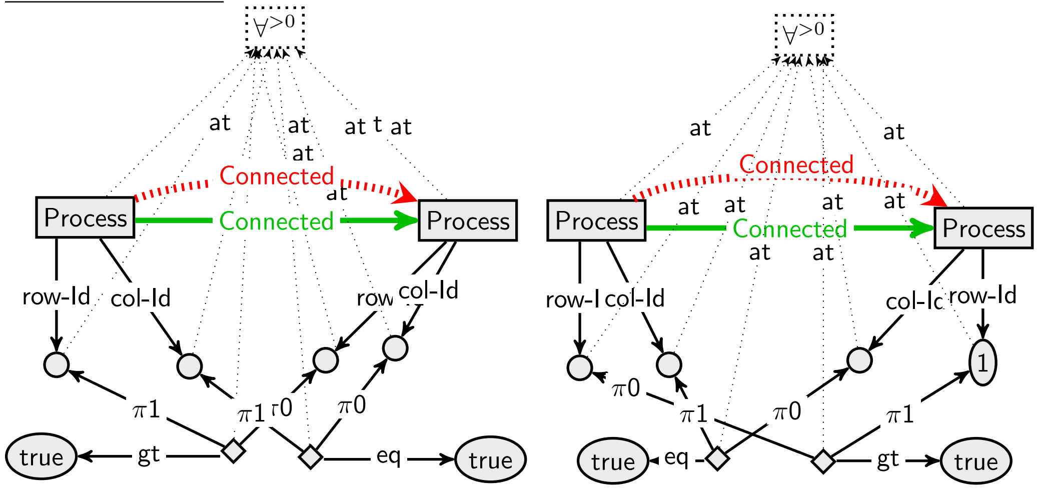


Figure 5.9: Transformation rules for addition of a vertical ring.

Chapter 6

Conclusion

In this work, we have contributed toward providing formal semantics for evolving communication protocols using Graph Transformation Systems (GTSs). We proved that a finite set of graph transformation rules can be analyzed and under certain conditions can be used to verify invariant properties of communication protocols without building the full state space. In the last part of this work, we presented symmetry-reduction techniques for graph transformation models of dynamically evolving systems. In this chapter, we summarize these results and outline possible directions for future work.

6.1 Summary

In Chapter 3, graph transformation rules were used to explain the dynamic behaviour of a distributed communication protocol. Our work produced a visualization of behaviour in three different levels. We explicated semantics of the third level of our model on an IP-based telephony system called Distributed Feature Composition (DFC) using graph transformations. A description of DFC semantics was presented by a graph transformation system with a hierarchical, attributed graph model and an (single-pushout) SPO approach for the transformation rules.

The basis for our model is to treat states of the system as graphs and computations of the system as transformations of these graphs. Similar to the description of a system based on formal-language grammars, our graph transformation system presented a visual notation that is based on graph transformation rules. Fundamental elements of our visual notation are graph alphabets and rewriting rules. Our GTS formalism provided these advantages: 1- presenting visual and modular semantics. 2- describing computations of communication protocols at an appropriately abstract level. 3- addressing typical communication protocol layers. 4- the ability to cleanly separate the concerns and focus on partial connections for analysis.

To show these advantages, we proposed an operational-semantics model using a hierarchical graph model. We modelled states of a communication system as attributed graphs where attributes show operational data for each component. These graphs show a composition of components and processes communicating with each other. At a higher level, we used a GTS that showed changes to the global state of the system. The global state of the system may be modified due to a local change of state in any of the components or via a topological change.

In Chapter 4, we addressed the verification problem for a class of systems with potentially unbounded state spaces. We showed that an invariant system property in which propositional formula is expressed as a graph can be verified against the GTS model by examining the model's finite set of transformation rules, and without resorting to the exploration of the full state space. While the problem is in general undecidable, we showed that the property is satisfied by the GTS model if the set of rules are property preserving. To enable this type of verification, first we defined the notion of graph satisfaction. Then we used this notion to define what it means for a transformation rule to preserve a property. We also defined a specific type of graph, a regular expression graph, which is an abstraction over connected nodes, and enables us to encode properties about path connectivity. We went on to explain an invariant property of reversible features in DFC. This property states that the sequence of reversible features in an existing call associated with an address is an invariant of the call. Maintaining this property is important in order to avoid feature interaction problems due to the dynamic evolution of the system. We then showed how to encode the propositional part of this property using graphs and how to perform the verification.

In Chapter 5, we defined a notion of symmetry for dynamically evolving symmetric multi-process systems modelled as GTS that may grow to a given maximum size. The explicit GTS semantic modelling can directly be exploited to reduce symmetric systems. Our symmetry-reduction technique was based on generating a reduced state space directly from the set of graph transformation rules. For this purpose, we defined the notions of GTS symmetry, and GTS bisimulation based on graph isomorphism. With GTS bisimulation, we described an on-the-fly algorithm that builds a symmetry-reduced model using the set of graph transformation rules that describe the full dynamic behaviour of the system.

To improve the reduction for symmetric GTS models, we defined vertex bisimulation. Vertex bisimulation describes an equivalence relation on state graphs based on their set of vertices and can be used in our algorithm for symmetry reduction resulting in an exponential state space saving. We also showed that two vertex-bisimilar GTS models could prove the same reachability properties given by a subset of CTL. In our method, we used proposition graphs to indicate Boolean expressions of atomic propositions as graphs.

6.2 Future Work

Modelling FSMs using GTS

Hierarchical modelling in GTS is a solution to the design and analysis of distributed communication systems. We have already provided a high-level modelling using GTSs that covers the dynamic topology changes, and changes of local operational data in the system. Here, we present ideas on how to model underlying finite-state machine of each component and also the relationship between level one and two. We have used examples to explain our ideas, and some of these examples relate to our previously used case study in Section 3.4.1. As a future work, these ideas can be implemented as an extension to a graph transformation tool.

Each FSM is a graph with several nodes representing states of the FSM and edges between these states, representing existing transitions between states. An example of a finite-state machine graph is depicted in Figure 6.1. In this figure, two extra edges are used, which point to the start state and the current state of the FSM. There is one node in the graph which is a representation of what component the FSM refers to. This node is labelled the same as the node that represents a component in the second-level graphs, e.g.

it can be the same as a Device or a Feature node in GTS rules describing DFC (depicted in Figure 3.8). There are two other nodes in the graph that represent two ports (in and out ports) of a component for communication. Each port has three attributes, the type of the port which is either *in* or *out*; the status of the port which is either *linked* or *idle*; and the message that is passed to the current in-port from other components or is going to be passed to another component through the out-port. In all the FSM graphs, ports are *linked*, because the FSM will operate when it is already assembled into a communication connection. The linking, which is the transformation of a port's status from *idle* to *linked* is done via a GTS rule in a higher level, e.g., like the rules we saw in Chapter 3. The FSM graph in Figure 6.1 resembles the FSM of a Caller in Figure 3.3.

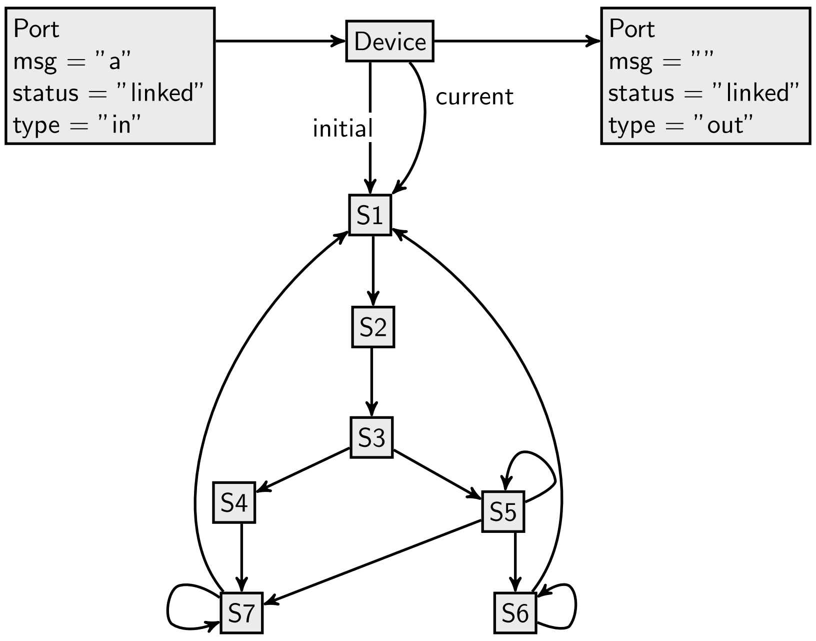


Figure 6.1: A FSM graph.

A set of transformation rules are used to describe the operational semantics of each FSM. Each FSM can either accept or transmit a sequence of messages via its in- or out-port. Therefore, based on each input message and the transition associated with that message in the FSM, there is one corresponding local transformation rule. Input messages belong to the finite set of input alphabet of the FSM. Rule *DeviceGetsMsg-a* in Figure 6.2

is an example of a local rule that checks if a transition exists between two specific states *S1*

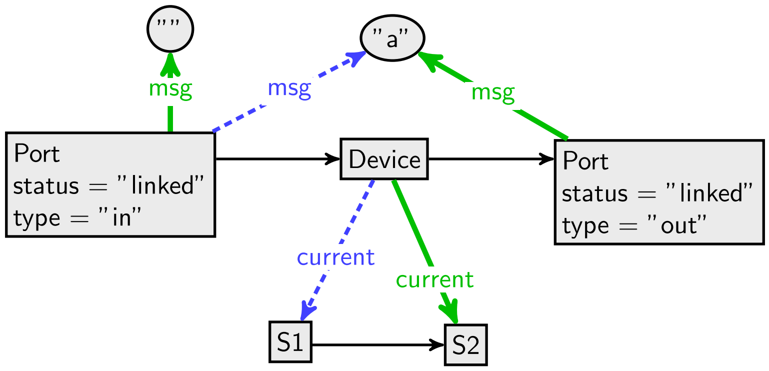


Figure 6.2: A local transformation rule.

and *S2* and whether the FSM has received message "*a*" on its in-port, then the message is passed to the out-port of the FSM and the current state will be changed to the state *S2*. For every transition and associated input in the FSM, there is one corresponding local transformation rule in the grammar. The application of this rule to the FSM graph in Figure 6.1 results in the transformation of the FSM graph to the graph in Figure 6.3.

On a higher level, messages will be passed between ports of components by a rule that first checks if the two ports are *linked*, which means that two components are connected in the architecture. Second, if the message in an out-port of a component is different than the message in the in-port of another component at the right side, then the passing of message is done and the message appears at the in-port of the other component. This rule is illustrated in Figure 6.4. Whenever a GTS rule of an FSM is applied to the FSM graph and changes the FSM's state accordingly and passes the message from in-port of a component to its out-port, it may trigger this higher-level rule that checks if a message at an out-port has been changed. This message passing then may trigger the application of a local transformation rule of another FSM graph.

To relate FSM graphs and the graphs that show the communication of components in the second level, the GTS rules in this level should be changed to reflect the connection through in- and out-ports. For example, two of the DFC rules that appeared previously in Figure 3.8 are remodelled using ports in Figure 6.5.

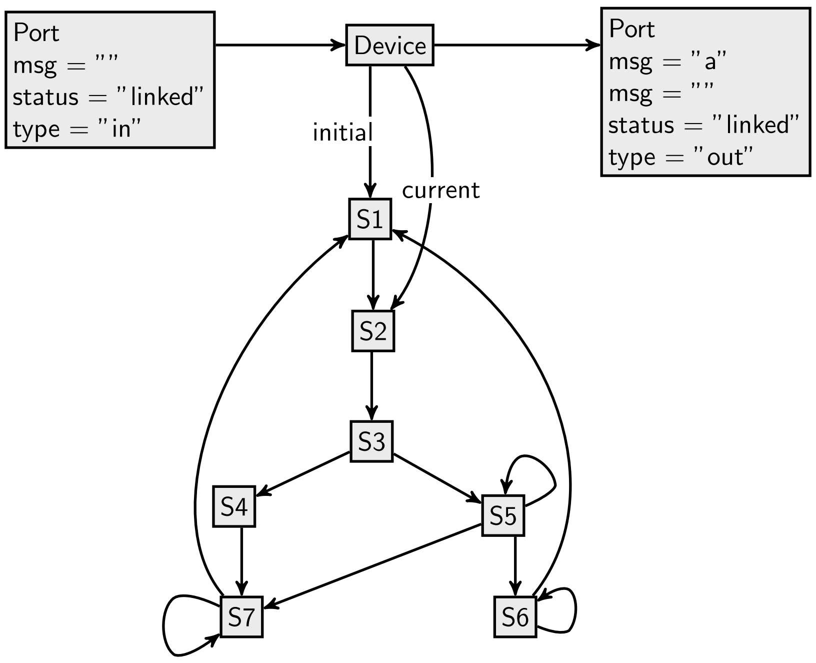


Figure 6.3: Transformed FSM graph.

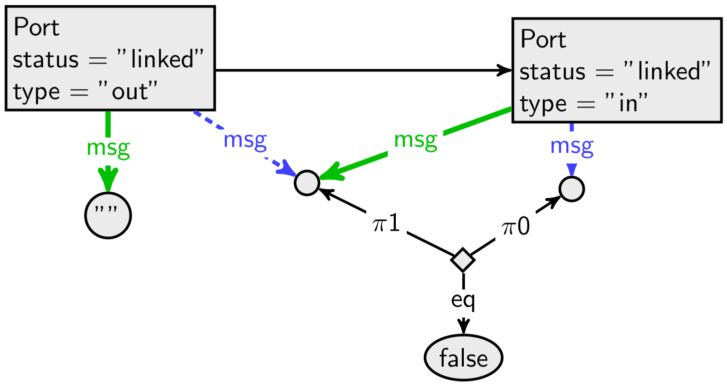
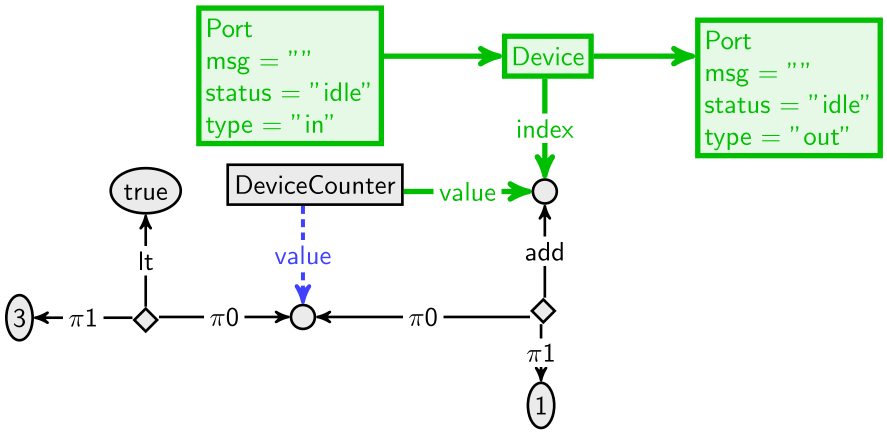


Figure 6.4: A rule to pass messages between components.

All the other rules describing our case study in Chapter 3 may be changed the same way. Then in all these rules, the simple edges between components will be changed to a connection with out-port and in-port nodes. As soon as a component is created to connect to another component, its in-port status will be *linked*, but if one end of the connection is

insertDevice



InsertFeature-Src

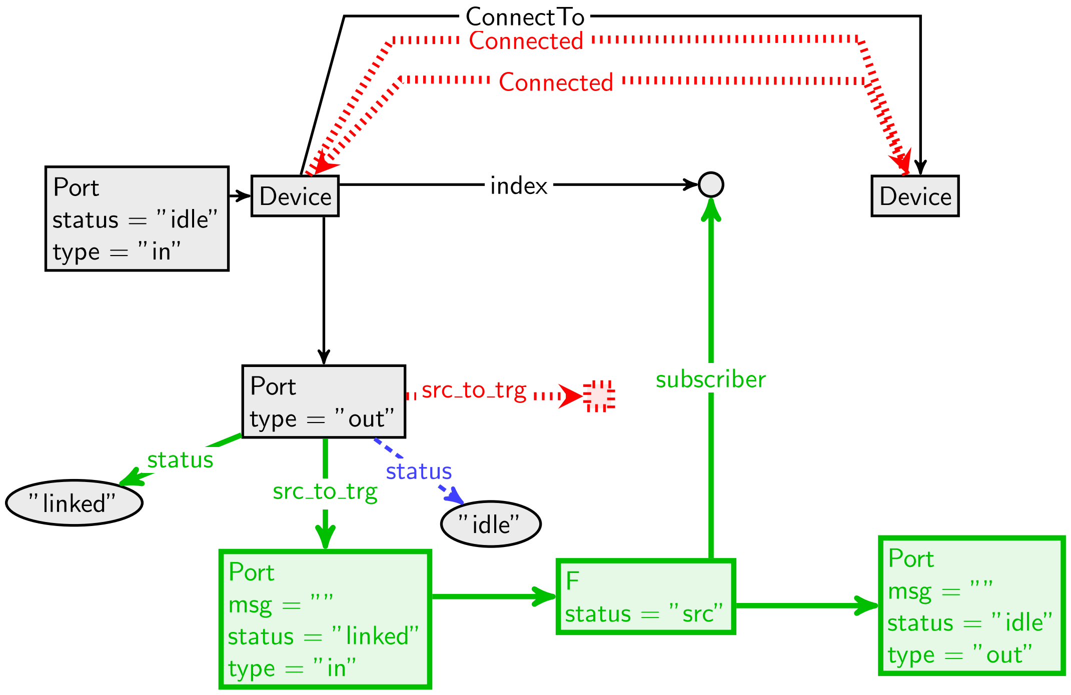


Figure 6.5: InsertDevice and InsertFeature-Src rules with the addition of ports.

still open then the status of an out-port node will be *idle*.

To give an idea of how transformation rules in different levels of DFC example may work together, if any change in an FSM graph of a component happens due to an input message, then the out-port of that FSM will be changed which triggers the rule in Figure 6.4. After the application of this rule then the message will appear at the in-port of another feature and then a local rule associated with the FSM of that feature will be matched and applied to show a computation of that feature, and finally the feature passes the message via its out-port to another feature downstream using the rule in Figure 6.4, and this process continues until the message reaches to the other end of the usage.

In Figure 6.5, each component has one in-port and one out-port, but realistically a component may have more ports, e.g., another in-port and out-port in the other direction. Some components such as a CW feature may have an additional port to enable another usage joining an existing usage. In this case, each second-level GTS rule may implement extra ports by creation of more port nodes when creating a feature by an *InsertFeature-Src* rule or by any of the *Append* rules. In this case, two different sets of ports may be distinguished by another attribute, e.g., to identify them as *upstream* or *downstream* ports, which means they are created for directing upstream signals or downstream signals. Hence, a feature may look like the one in Figure 6.6. Therefore, there may be some changes in the local FSM rules that pass a message between ports of a component reflecting the type of ports used. For the higher-level rule in Figure 6.4, there is no change, because a message is always passed from an in-port to an out-port no matter which in-port receives the message, i.e., no matter whether the signal travels upstream or downstream. A usage with a source CW feature connected via ports may then look like the usage in Figure 6.7. For the CW feature, the local FSM rule should be changed to include additional ports and the way the rule changes a current state based on the message passed between different sets of ports.

Program Text Reduction

Program text reduction is another way of system abstraction in which, instead of building the quotient model, program P is directly reduced to program $\bar{P}$ using the description of program P. Thus, the transition system model driven from $\bar{P}$ can be used for verification without the requirement to build the reduced model of program P. It has been shown

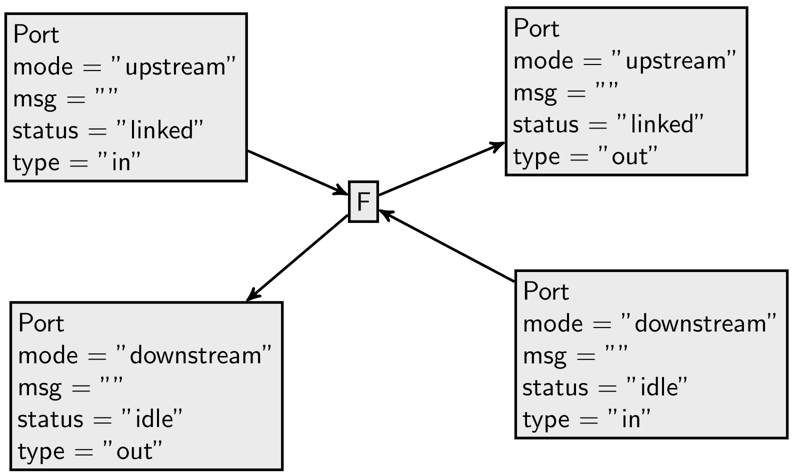


Figure 6.6: A feature with several in- and out- ports.

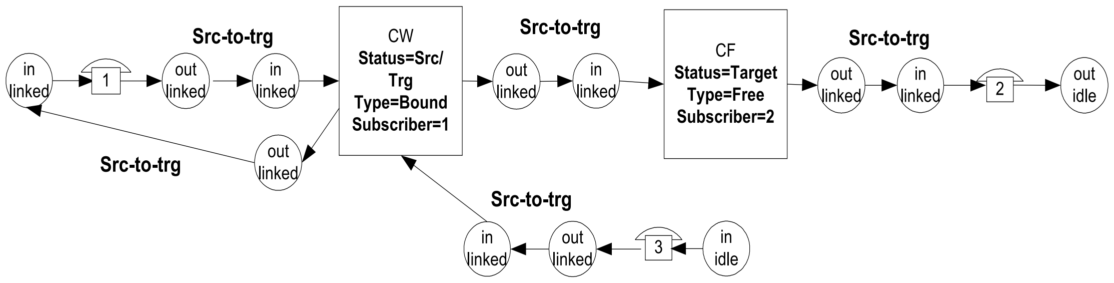


Figure 6.7: Source CW feature joins an existing usage.

that with the assumption of full symmetry this method efficiently reduces the programs of multi-process systems [ET99, BMWD09]. In a GTS formalism, a system is described through its set of transformation rules; thus, this set is a representation of the program text.

As a future work, we show that we can use vertex bisimulation to introduce a specific type of program-text reduction on symmetric systems where the program text has been described through a set of GTS rules. By investigating the transformation rules, we would like to find out which rules generate a vertex-bisimilar state graph when applied to a state. Without a doubt, these rules are not adding or deleting any processes to or from the communication graph of processes; they are simply reconfiguring the topology, keeping the same set of local states before and after the application of the transformation rule. Since we have exploited vertex bisimulation as a way of symmetry reduction for GTS models, we can therefore perform the program-text reduction by eliminating these rules from the set of graph transformation rules that describe the system. This elimination is an abstraction over two vertex-bisimilar state graphs.

As an example, in Figure 6.8, the application of the GTS rule PassTokenNtoN to a ring host graph with node labelling N^+NT is depicted. The application of this rule passes a token from one Non-trying process to another Non-trying process. This GTS rule transforms the graph with node labelling N^+NT to the graph with the node labelling NN^+T. These two state graphs are vertex-bisimilar. But using program-text reduction, in

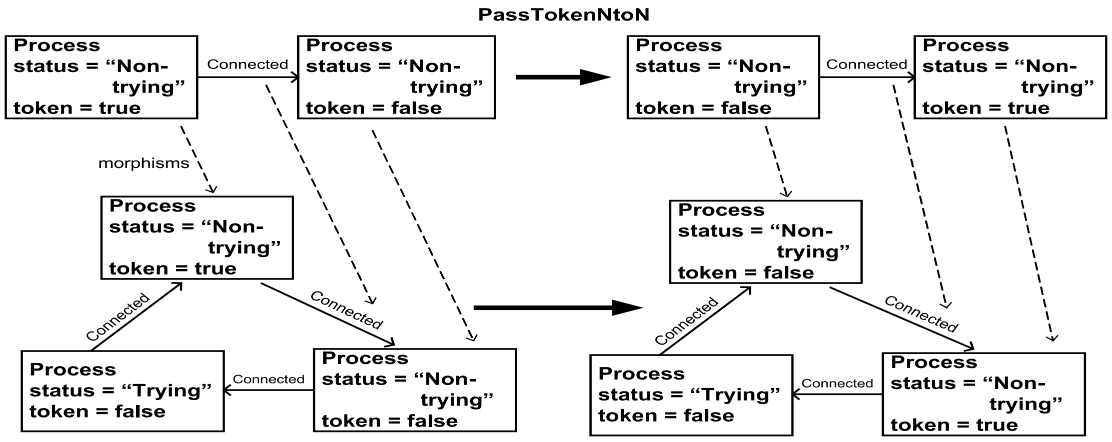


Figure 6.8: Applying the rule PassTokenNtoN.

combination with the algorithm GENERATEQUOTIENT, results in an exponentially reduced model. Comparing this combination method with the method of generating the quotient using vertex bisimulation, although the state-space savings is the same, there is a savings in the running time of the algorithm and also in the number of transitions generated in the quotient model when we combine these methods.

GTS Partial Order Reduction

Explicit-state model checking is known as an effective verification technique for software systems. The main problem with this technique, however, is the state explosion issue. To deal with this problem, some reduction techniques such as symmetry reduction and partial order reduction are used. We have presented symmetry reduction methods for GTS models of multi-process systems. In partial order reduction, the focus is on reducing the number of transitions by exploiting the commutativity of independent, concurrent transitions. We realized there may not be a benefit in using partial order reduction if all actions are dependent. However, for systems with many independent actions there is a clear benefit in using equivalence classes on transitions instead of states. This is especially important in reduction of graph-based models, because detecting symmetries of graphs is usually an expensive undertaking.

The idea is to use the set of transformation rules to find the set of independent actions. In a GTS, an action is enabled by the application of a transformation rule in a state defined as a graph. Rules are applied if there are morphisms between their left-side graph and the state graph. Studying the morphisms and detecting conflicts between left-sides of two different rules helps to define dependency between these two rules. Thus, the application of a set of independent rules to a state graph may be reordered and different orderings can be defined as an equivalent class of rule applications. This requires study of confluency in two GTS derivations. Existing partial order reduction techniques cannot directly be applied to GTS model of the systems; thus, future work in this area is beneficial to reduction of GTS-based modelling.

Supporting Tool and Implementation

This thesis is a contribution to the use of graph transformation systems in the modelling and verification of communication systems. Unfortunately, there are currently no tools to fully support these ideas. Specifically, the practical use of methodologies for software systems require a tool that implements the three level semantics, and the symmetry reduction algorithm. The main challenge for a supporting tool is modelling synchronization of the computations of the first-level graphs (FSMs) and also modelling events in these graphs. In addition, another difficult task is to use GTS to model the interactions between these FSMs. A challenging future work is then the implementation of these ideas in a software tool. The tool must support the modelling of graphs as finite-state machines and provide hierarchical levels in the GTS formalism.

While the focus of this thesis has not been implementing or enhancing GTS tools, but rather providing the theoretical basis for verification using an expressive, Turing-complete formalism, here we show some of the challenges of implementing our ideas. For implementing the approach for invariant checking in GTS models, the main challenge of automation is checking the satisfiability of graphs, hence, checking REG morphisms between graphs in rule and proposition graphs. In addition, the ability to fully support the temporal operators of the properties in a graphical format is another challenge in automation of GTS verification.

In general, automation of symmetry reduction techniques proved to be feasible, since some of the explicit-state model checkers such as MURφ have incorporated symmetry reduction techniques. Unfortunately, MURφ is limited to verifying invariant properties in fully-symmetric models. For the automation purposes, considering full symmetry makes the quotient generation easier, but the disadvantage of the method in MURφ is that the symmetry should be considered when the user models the system. In contrary, symmetry reduction based on graph isomorphism considers dynamic, non-fully symmetric systems as they are designed. Thus, automation in case of solving the isomorphism checking is rather straightforward as it is shown in GENERATEQUOTIENT algorithm 5.2.

It means that if the transition system was a Kripke structure, in which permutations are simply index permutations, the implementation was rather a straightforward task, but for systems modelled as graphs, the automation is more complex for determining if two graphs are permutation of each other. Therefore, the main challenge is more related to the GTS tool and how it can solve the isomorphism checking problem. Graph isomorphism checking, although is not known to be in P or NP class of problems, has been shown to have an automatic solution for some of the system sizes in practice [McK81].

Automata-based Model Checking using GTS

Automata-based model checking has been used frequently for the verification of distributed systems. This methodology converts the negation of a property to an automaton. This automaton accepts all the behaviours that violates the original property; the system will then be modelled as an automaton as well. The intersection of these two automata will either result in an empty set which means that the property is satisfied by the model or it will show all the violating behaviours. This methodology presents its own challenges in building automata specifically for dynamic systems.

Usually the behaviours of automata can be expressed in terms of sequences of communication messages between processes. These sequences are defined by a language, which is called the language of automaton. This methodology can also be seen as language containment, such that if the language (sequence of words) accepted by the system automaton is a subset of the language accepted by the property automaton then the property is satisfied. This view of language containment can be leveraged to GTS formalisms. The reason for this idea is that graph transformation rules that describe the behaviour of a system can be viewed as a graph grammar, and a graph-based language can be derived from the grammar. In the same way, a property can be defined as a GTS and its language will be derived from its grammar. Finally, we find out if the language accepted by GTS of the system is a subset of the language derived from the graph grammar of the property, which shows the property satisfaction. Therefore, it is enough to show that the GTS of the property accepts the graph-based language derived from the GTS of the system. We suggest building a graph-automata for recognition of a graph-based language. A graph-automata can be seen as a graph transducer that accepts graphs as input and generates a trace of transformation rules that transforms that graph to another one. However, unlike words building graph transducers is a challenging task.

Verification of Liveness Properties

In the literature related to model checking, verification of liveness properties is considered more difficult than verification of safety properties. In contrast to safety properties that make sure undesirable behaviours do not occur in the system's model, liveness properties ensure something desirable eventually occurs in the system's behaviour. These properties are important in a communication system, because they provide a way of proving that the system eventually reaches a good state, e.g. that a message eventually arrives at its destination, which demonstrates the absence of infinite loops or failure. Therefore, as an area for future research it would be interesting to extend our GTS verification method to provide the verification of (some) liveness properties described as graphs. In addition, it would be interesting to determine what conditions should hold in GTS rules for the satisfaction of liveness properties.

Bibliography

[AEH$^+$99] M. Andries, G. Engles, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999. 6, 24

[AENT03] N. Amla, E. A. Emerson, K. Namjoshi, and R. Trefler. Abstract patterns of compositional reasoning. In *CONCUR '03*, pages 423–438, September 2003. 33

[AK86] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986. 103, 104

[BBG$^+$06] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE '06*, pages 72–81, 2006. 82

[BCK01a] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. *LNCS*, 2154:381–395, 2001. 81

[BCK01b] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. *Lecture Notes in Computer Science*, 2154:381–389, 2001. 9

[BCK04] P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170. 9, 81

[BCP$^+$04] G.W. Bond, E. Cheung, K. Hal Purdy, P. Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Trans. Inter. Tech.*, 4(1):83–123, 2004. 34

[BH02] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *ICGT '02*, LNCS 2505, pages 402–429, 2002. 39

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007. 15

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *CAV*, pages 403–418, 2000. 103

[BKR05] P. Baldan, B. König, and A. Rensink. Summary 2: Graph grammar verification through abstraction. In *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*, volume 04241, 2005. 9, 81

[BMWD09] G. Basler, M. Mazzucchi, T. Wahl, and Kroening D. Symbolic counter abstraction for concurrent software. In *CAV*, 2009. 116

[BMWK09] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, pages 64–78, 2009. 96

[BRKB07] I. B. Boneva, A. Rensink, M. E. Kurban, and J. Bauer. Graph abstraction and abstract graph transformation. Technical report, Centre for Telematics and Information Technology, University of Twente, July 2007. 10

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 3

[BS03] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003. 49

[Bur69] R.M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12:41–48, 1969. 75

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. 7, 40, 41

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *ICSE*, pages 762–765, 2000. 15

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1981. 2, 12, 13

[CEFJ96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods in Sys. Des.*, 9(1-2):77–104, 1996. 3, 7, 10, 15, 16, 20, 83, 84, 85, 102

[CEH$^+$97] A. Corradini, H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, and A. Wagner. Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, pages 247–312. World Scientific, 1997. 8, 25, 26, 28

[CGB86] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 1986. ACM. 103

[CGL$^+$94] J. Cameron, N. D. Griffeth, Y. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In *FIW*, pages 1–23, 1994. 6

[CKMRM03] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1):115–141, 2003. 6

[CL96] G. Chartrand and L. Lesniak. *Graphs & Digraphs, 3rd ed.* Chapman & Hall, 1996. 22

[CMR$^+$97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation. part i: basic concepts and double pushout approach. pages 163–245, 1997. 8, 25, 26, 27

[dBG05] L. du Bousquet and O. Gaudoin. Telephony feature validation against eventuality properties and interaction detection based on a statistical analysis of the time to service. In *ICFI '05: Eight International Conference on Feature Interactions in Telecommunications and Software Systems*, pages 78–95, 2005. 33

[DFRdS03] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi dos Santos. Verification of distributed object-based systems. In *FMOODS*, pages 261–275, 2003. 9

[DHR$^+$07] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136. IEEE Computer Society, 2007. 15

[Dom05] Alma L. Juarez Dominguez. Verification of DFC call protocol correctness criteria. Master's thesis, University of Waterloo, Waterloo, Canada, 2005. 41

[dSDR06] O. M. dos Santos, F. L. Dotti, and L. Ribeiro. Verifying object-based graph grammars. *Software and System Modeling*, (3):289–311, 2006. 81

[EEKR99] H. Ehrig, G. Engels, H-J Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Applications, Languages, and Tools.* World Scientific, 1999. 30

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer-Verlag New York, Inc., 2006. 6, 22, 70

[EH00] G. Engels and R. Heckel. From trees to graphs: Defining the semantics of diagram languages with graph transformation. In *ICALP Satellite Workshops*, pages 373–382, 2000. 33

[EK00] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE*, pages 236–254, 2000. 103

[EMCGP99] Jr. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999. 2, 5, 12

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9(1/2):105–131, 1996. 3, 7, 10, 15, 16, 18, 19, 20, 83, 84, 85, 89, 102

[ES97] E. Allen Emerson and A. Prasad Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997. 19, 97

[ET99] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods*, 1999. 10, 20, 96, 102, 116

[Gia99] Dimitra Giannakopoulou. *Model Checking for Concurrent Software Architectures.* PhD thesis, Dept. of Computing, Imperial College, London, 1999. 49

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co Ltd, 1979. 91

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, New York, NY, USA, 1993. 2

[GP94] David Garlan and Dewayne E. Perry. Software architecture: Practice, potential, and pitfalls. In *ICSE*, pages 363–364, 1994. 48, 49

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. 103

[Gur04] Yuri Gurevich. Abstract state machines: An overview of the project. In *FOIKS*, LNCS 2942, pages 6–13. Springer-Verlag, 2004. 49

[Hav99] Klaus Havelund. Java pathfinder, a translator from java to promela. In *SPIN*, 1999. 15

[Hec98] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *FASE*, 1998. 6, 8, 49

[HM00] B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In *ICALP Satellite Workshops*, pages 443–450, 2000. 49

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997. 3, 5, 15

[Hol99] Gerard J. Holzmann. Software model checking. In *FORTE*, pages 481–497, 1999. 14

[HPR06] A. Habel, K. Pennemann, and A. Rensink. Weakest preconditions for high-level programs. LNCS 4178, 2006. 43, 70, 81

[HU79] JE Hopcroft and JD Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979. 40

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996. 3, 7, 10, 15, 16, 83, 84, 102

[ID99] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3), May 1999. 103

[Ios02] Radu Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, pages 22–41, 2002. 103

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Software Eng.*, 21(4):373–386, 1995. 48

[JZ98] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *Software Engineering*, 24(10):831–847, 1998. 34, 38, 46

[KKH06] H.-J. Kreowski and S. Klempien-Hinrichs, R.and Kuske. Some essentials of graph transformation. In Z. Esik, C. Martin-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 229–254. Springer, 2006. 22

[KR06] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In A. Valmari, editor, *Model Checking Software (SPIN)*, LNCS 3925, pages 299–305. Springer-Verlag, 2006. 9, 69, 70, 81

[Kus01] S. Kuske. A formal semantics of uml state machines based on structured graph transformation. In Gogolla and Kobryn, editors, *UML*, volume 2185 of *LNCS*, pages 241–256. Springer-Verlag, 2001. 49

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. 3

[LP05] Z. Langari and A. B. Pidduck. Quality, cleanroom and formal methods. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005. 1

[LT06] Z. Langari and R. Trefler. Formal modeling of communication protocols by graph transformation. In *FM 06*, LNCS 4085, pages 348–363, 2006. 6, 65, 74, 82, 84

[LT09] Z. Langari and R. Trefler. Application of graph transformation in verification of dynamic systems. In *IFM 09*, LNCS 5423, pages 261–276, 2009. 7, 84, 97

[LT10] Z. Langari and R. Trefler. Symmetry for the analysis of dynamic systems. 2010. (submitted for publication). 8

[Mag95] 5th european software engineering conference, sitges, spain, september 25-28, 1995, proceedings. In Wilhelm Schäfer and Pere Botella, editors, *ESEC*, volume 989 of *Lecture Notes in Computer Science*. Springer, 1995. 48

[McK81] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. 77, 91, 119

[McM93] K. L. McMillan. *Symbolic Model checking*. Kluwer Academic Publishers, Boston, MA, USA, 1993. 3, 5, 14

[Mét98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998. 49

[MNV72] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. In *Proceedings of ACM conference on Proving assertions about programs*, pages 27–50, New York, NY, USA, 1972. ACM. 75

[Pac03] Jan K. Pachl. Reachability problems for communicating finite state machines. *CORR*, cs. LO/0306121, 2003. 40

[Pel98] Doron Peled. Ten years of partial order reduction. In *CAV*, pages 17–28, 1998. 3, 15

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977. 3, 12, 13

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982. 2, 12

[RD05a] A. Rensink and D. Distefano. Abstract graph transformation. In *International Workshop on Software Verification and Validation (SVV)*, 2005. 81

[RD05b] Arend Rensink and Dino Distefano. Abstract graph transformation. Ctit technical report, Department of Computer Science, University of Twente, Jan 2005. 10

[Ren03] A. Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at `http://www.cs.utwente.nl/~groove`, 2003. 9, 45

[Ren04a] Arend Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 2004. 10

[Ren04b] Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, 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. Springer-Verlag, 2004. 30

[Ren04c] Arend Rensink. Representing first-order logic using graphs. In *ICGT*, pages 319–335, 2004. 31

[Ren04d] Arend Rensink. State space abstraction using shape graphs. In *Automatic Verification of Infinite-State Systems (AVIS)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. 10

[Ren06] Arend Rensink. Isomorphism checking in groove. *ECEASST*, vol. 1, 2006. 10, 91, 103

[Ren08] Arend Rensink. Explicit state model checking for graph grammars. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 114–132, 2008. 10, 23, 103

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997. 6, 21, 22, 23, 25, 27, 31, 39, 43, 74

[SG90] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 151–165, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 50

[SG96] M. Shaw and D. Garlan. *Software architecture.* Prentice-Hall, 1996. 37, 48

[TKFV99] G. Taentzer, M. Koch, I. Fischer, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In H. Ehrig,

U. Montanari, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. III: Concurrency, Parallelism, and Distribution*, pages 269 – 341. World Scientific, 1999. 8, 49

[TW09] Richard J. Trefler and Thomas Wahl. Extending symmetry reduction by exploiting system architecture. In *VMCAI*, 2009. 10, 20, 84, 88, 94, 95, 96, 97, 102

[Var03] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 2003. 9, 81

[VHB$^+$03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003. 15

[Zav03] P. Zave. Ideal connection paths in DFC. Technical report, AT&T Research, November 2003. 67

[Zav04] P. Zave. Distributed feature composition: Middleware for connection services, 2004. AT&T Technicl Report. 6