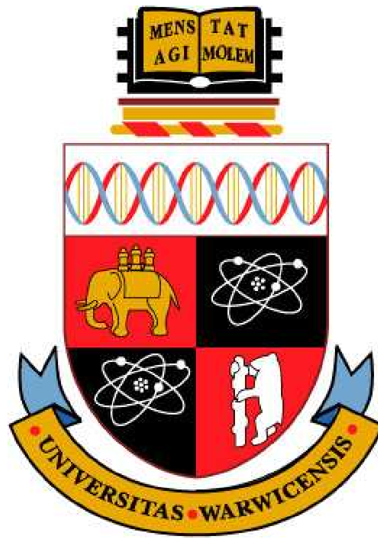

Theoretical and Practical Tools for Validating Discrete and Real-time Systems

by

Hongyang Qu

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

October 2005

THE UNIVERSITY OF
WARWICK

Contents

List of Figures vii

Acknowledgments x

Declaration xii

Abstract xiv

Chapter 1 Introduction 1

1.1 Motivations and related work 2

1.1.1 Dealing with state space explosion 2

1.1.2 Managing concurrent behavior 3

1.1.3 Path condition under partial order relations 4

1.1.4 Probability of real-time traces 5

1.2 Contribution 8

1.3 Thesis Outline 12

Chapter 2 Background 17

2.1 Existing models 17

2.1.1 Flow charts 17

2.1.2 Timed automata 19
2.1.3 Timed transition diagrams 21
2.2 Explicit state model checking 22
2.3 Partial order 25
2.4 Path precondition 28
2.5 Symbolic execution 29
2.6 Probability theory 30
2.7 Related software tools 32
2.7.1 SPIN and PROMELA 32
2.7.2 The Omega library 33
2.7.3 DOT 33
2.7.4 Pascal 34
2.7.5 The C Language 34
2.7.6 Tcl/Tk 35
2.7.7 Lex & Yacc 36
2.7.8 SML/NJ 36
2.7.9 HOL90 37
2.7.10 Mathematica 38
2.8 The untimed version of PET 38
2.8.1 The functions of PET 38
2.8.2 The structure of PET 40

Chapter 3 Safe annotation 42

3.1 Verification of Annotated Code 42
3.1.1 Programming Constructs for Annotations 44

3.1.2 Examples 49
3.1.3 Preserving the Viability of Counterexamples 53
3.2 Experiments 55
3.2.1 Implementation 55
3.2.2 An experiment 57
3.3 Summary 67

Chapter 4 Enforcing execution of a partial order 69
4.1 Transforming shared-variable programs 70
4.1.1 Data structure of transformation 70
4.1.2 Discussion 73
4.1.3 An example 75
4.2 Preserving the Checked Property 80
4.3 The Distributed Program Model 83
4.4 Extending the Framework to Infinite Traces 86
4.5 Summary 89

Chapter 5 Modeling real-time systems 91
5.1 Transition systems 91
5.2 Extended timed automata 95
5.2.1 The definition 95
5.2.2 The execution 96
5.2.3 The product 97
5.3 Translating a TS into ETAs 98
5.4 Modeling shared variables 103
5.5 Modeling communication transitions 108

5.6 Generation of the DAG 110
5.7 An example 111
5.8 Summary 117

Chapter 6 Calculating the precondition of a partial order 121

6.1 Calculating untimed path condition in timed systems 122
6.2 Untimed precondition of a DAG 124
6.3 Date structure to represent time constraints 125
6.3.1 The definition 125
6.3.2 Operations on DBMs 127
6.4 Calculating the timed precondition of a DAG 130
6.5 A running example 131
6.6 Time unbalanced partial order 136
6.6.1 The definition 138
6.6.2 A remedy to the problem 142
6.6.3 An application of the remedy method 145
6.7 Summary 147

Chapter 7 Calculating the probability of a partial order 150

7.1 The model 151
7.1.1 Probabilistic transition systems 151
7.1.2 Calculating participating transitions 154
7.1.3 An example system 155
7.2 The probability of a path 156
7.2.1 Timing relations along a path 156
7.2.2 Computing the probability of a path 157

7.2.3 The complexity of the computation 161
7.2.4 Simplification for exponential distribution 162
7.3 The probability of a partial order 165
7.3.1 The synchronization behind partial order 167
7.3.2 The general algorithm 170
7.3.3 Optimized algorithm for communication transitions . 175
7.4 Summary 179

Chapter 8 Real-time Path Exploration Tool **181**

8.1 The existing work 181
8.2 The implementation of code transformation 185
8.3 The real-time extension of PET 186
8.3.1 The real-time features 186
8.3.2 The structure of the extension 188
8.3.3 The implementation details 193
8.4 Summary 201

Chapter 9 Conclusion **204**

Appendix A BNF grammar of the annotation **212**

Appendix B Real-time PET user manual **217**

List of Figures

2.1 Flow chart nodes 19

2.2 Translation of *branch* statements 19

2.3 An untimed transition 21

2.4 A timed transition 22

3.1 The flow chart and the automaton for the left process 63

3.2 Our implementation results 65

3.3 The SPIN results 65

4.1 Dekker's mutual exclusion solution 76

4.2 Dekker's mutual exclusion solution 77

4.3 The order between occurrences in the execution ρ_2 78

4.4 The transformed Dekker's algorithm 79

5.1 The framework of generating a DAG 92

5.2 The edge 93

5.3 The translation of an *assignment* node 94

5.4 The translation of a *branch* node 94

5.5 Another translation of a *branch* node 95

5.6 A neighborhood of two transitions 101

5.7 An extended timed automaton for a guarded transition 102
5.8 Two sequential transitions 102
5.9 The translation of two sequential transitions 103
5.10 The pair of communication transitions 109
5.11 The product of communication transitions 109
5.12 An example system 112
5.13 Flow charts for Program P1 (left) and P2 (right) 113
5.14 Processes for programs P1 (left) and P2 (right) 114
5.15 ETAs for programs P1 (left) and P2 (right) 115
5.16 The product 116
5.17 Partial order 1 117
5.18 The DAG for the partial order 1 118
5.19 Partial order 2 and the corresponding DAG 119

6.1 A path 122
6.2 An example 129
6.3 A simple concurrent real-time system 132
6.4 Program 1 and 2 133
6.5 Partial order 1 and 2 134
6.6 The DAG for Partial order 1 135
6.7 An example partial order 139
6.8 A negative example 144
6.9 Two unextendable partial orders 144
6.10 Partial order 3 (left) and 4 (right) 146

7.1 The example system 155

7.2 Two partial order examples 167
7.3 Synchronization (left) and non-synchronization (right) scenarios 168
7.4 d'_1 is disabled by b_1 (left) and d'_1 is disabled by β (right) 169
7.5 Two system examples 170

8.1 The PET structure 189
8.2 The source code windows 190
8.3 The flow chart windows 191
8.4 The transition system windows 192
8.5 The partial order window and the path window 193
8.6 The main window 194

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Prof. Doron Peled, for his generous support and incisive guidance, which have been the unfailing source of inspiration during the research and will be a prolonged influence upon my future career.

I would like to thank Prof. Marta Kwiatkowska and Dr. Sara Kalvala for giving me precise comments on the final version of this thesis.

Many thanks go to Dr. Marcin Jurdziński for his invaluable advice on my research, and Nick Papanikolaou and Ashutosh Trivedi for proof reading my thesis and making many beneficial suggestions.

Additionally, I would like to thank my colleagues in Department of Computer Science: Dr. Rajagopal Nagarajan, Dr. Ranko Lazic, Dr. Allan Wong, Dr. Li Wang, Chien-An Chen, Huibiao Zhu, Lei Zhao, and Jiang Chen for their precious discussions and comments in a variety of circumstances.

Special thanks are due to my wife Lingling Wang because of her tremendous sacrifice and continuous encouragement in helping me to pull through the exacting study.

Last but not least, I would like to thank all my family and friends for

uncountable happiness they brought to me throughout these memorable years.

Declaration

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated.

In collaboration with Doron Peled, the approach to limit state space searching by annotation in Chapter 3 was published in the proceeding of *the* 23rd *IFIP International Conference on Formal Techniques for Networked and Distributed Systems* [PQ03].

The method to guarantee the execution of a specified path, presented in Chapter 4, was a joint work with Doron Peled and published in Electronic Notes in Theoretical Computer Science, volume 113, 2005, as the proceeding of *the* 4th *Workshop on Runtime Verification* [PQ05a].

The models, transition systems and extended time automata, in Chapter 5 and the algorithm to compute the precondition of a partial order in Chapter 6 was published in the proceeding of *the* 1st *International Symposium on Leveraging Applications of Formal Methods* [BPQT04], in collaboration with Doron Peled, Saddek Bensalem and Stavros Tripakis.

The time unbalanced partial order presented in Chapter 6 was a joint

work with Doron Peled and accepted by *the* 5th *International Workshop on Formal Approaches to Testing of Software* [PQ05b].

Based on the model in Chapter 5, the calculation of probability of a path in a real-time system in Chapter 7 was accepted by *the* 5th *International Workshop on Formal Approaches to Testing of Software* [JPQ05]. It is a joint work with Doron Peled and Marcin Jurdziński.

The key ideas in these papers were developed by myself under the supervison of Prof. Doron Peled and the implementation for these papers was performed solely by myself.

Abstract

System validation has been investigated for a long time. Testing is used to find errors inside a system; in contrast, model checking is used to verify whether a given property holds in the system. Both methods have their own advantages and interact with each other. This thesis focuses on four methodologies for model checking and testing. In the end, they are integrated into a practical validating tool set, which is described in this thesis.

Many techniques have been developed to manage the state space for a complicated system. But they still fail to reduce the state space for some large-scale concurrent systems. We propose using code annotation as a means of manually controlling the state space. This solution provides a trade-off between computability and exhaustiveness.

When a suspicious execution is found either by testing or by model checking, it can be difficult to repeat this execution in a real environment due to nondeterministic choices existing in the system. We suggest enforcing a given execution by code transformation. In addition, we extend our method from a single path to partial order executions.

In order to repeat at least one such execution, we need to provide appropriate values satisfying the path's initial precondition in its environment. It is easy to obtain the precondition in a discrete environment, but difficult in a real-time environment, especially for a partial order, since the computation would involve time constraints in the latter case. We present a real-time model first, and then a methodology to compute the precondition on this model.

When every action in the system is associated with a probability density function, it is possible to calculate the probability of the occurrence of a particular execution. We give a method to calculate the probability by integration on a group of independent continuous random variables, each of which is corresponding to an action either executed, or enabled but not fired.

The research described in this thesis provides some new ideas for applying formal methods to classical software development tools.

Chapter 1

Introduction

Techniques to validate a system include model checking [CE81, QS82] and testing [Har00], which stand for two different ends of Formal Methods. Model checking is an automatic approach for the verification of systems. It searches the whole state space of a system to verify that the system possesses a given property. Theoretically, it is a systematic and comprehensive methodology. The system is proved to be correct with respect to a property if it cannot find a violation of the property. But it is very common that a system is so complicated that it has to be abstracted before being verified by model checking, which results in weakening the reliability and the effect of model checking. Testing is a traditional method to improve quality of systems and has been used broadly in daily development of systems. It benefits from the ability to exploit human experience and intuition in order to accelerate the validation. Its main advantage is that it can be applied easily with affordable cost, but it can hardly claim that a system is flawless if it does not find any bugs inside the system. Therefore, model checking

and testing are not able to replace each other and they are both adopted to validate systems in many cases. *This thesis is focused on providing solutions to several nontrivial problems existing in validating systems. Each solution contains not only a theoretical methodology, but also a practical tool which implements the methodology. Furthermore, the combination of these solutions constructs an integrated powerful validating tool set.*

1.1 Motivations and related work

1.1.1 Dealing with state space explosion

The first and most typical problem in model checking is state space explosion. In concurrent systems, the number of states can increase exponentially with the number of independent components. For a large system, the state space might be too huge to be checked thoroughly. Symbolic model checking [McM92] does not visit and represent each state individually, hence can sometimes be applied to cases that seem 'hopeless' for explicit state model checking [EP02]. However, the problem is still not alleviated completely, although symbolic model checking techniques have successfully enhanced the power of model checking. On the other hand, there are several cases where one prefers the explicit state model checking. In particular, this is useful for testing or simulating the code, and is also simpler for generating counterexamples.

Explicit state model checking applies a search algorithm to the state space of the verified system. Partial order reduction [Pel94, Vog93] is a very important technique to increase the capability of explicit state model

checking. But it still cannot deal with many realistic systems. Many other methods have been proposed to assist symbolic model checking and partial order reduction to tackle this problem and have shown encouraging results, such as compositional reasoning [GL91], abstraction [CGL94], symmetry [CFJ93, ES93] and induction [KM89] (see [CGP99] for a detailed survey). But often they still fail to reduce the number of accessed states to become manageable. This thesis presents another method in order to limit the state space being checked.

1.1.2 Managing concurrent behavior

The second problem is how to repeat a suspicious path of a concurrent system in a real environment. When a model checker reports a counterexample (represented by an execution, or simply, a path) against a property, users need to verify whether this path violates the property or it is only a "false negative" since it is often not the code itself that is being verified, but rather a model of the code. Furthermore, they may want to understand why it is generated. Thus, they usually run the system to execute this path against the actual code. Testers also encounter such problems when they trace a bug or are required to show a suspicious behavior actually occurs during some run of the code. However, there could be some nondeterministic choices in the system. Hence the system behaves nondeterministically so that it cannot ensure to execute the specified path. Another difficulty that might occur is that the execution of the path requires some uncommon scheduling which may be very hard for the testers to reproduce in an execution. We generalize the execution of a specified path, taking into account only the es-

sential relations among actions on the path. This provides us with a partial order [Lam78] that is consistent with the path, but may also be consistent with other paths. All these paths are *equivalent* with respect to the partial order.

The partial order semantics was studied extensively as an alternative for the more traditional interleaving (linear) semantics [NPW81, Pra86]. The advantage of the interleaving semantics is that it can be handled with simpler mathematical tools (such as linear temporal logic [Pnu77] or Büchi automata [Tho90]) than the tools needed for partial order semantics. But partial order semantics gives a more intuitive representation of concurrency. We suggest that the rationale to recover the concurrent execution related to the behavior, rather than a completely synchronous linear execution, gives another motivation for using the partial order semantics. This thesis proposes a method to instrument the code so that the execution of a path represented by a particular partial order is guaranteed.

1.1.3 Path condition under partial order relations

The third problem is also related to a specified partial order. In order to guarantee a partial order execution, users must provide the weakest precondition [Dij75] of the partial order to the system. The problem is how to obtain the precondition. In a discrete system (untimed system), the precondition of a partial order is the same as that of any single path which retains the partial order. We obtain the precondition of a partial order by calculating the precondition of any path in the group of equivalent paths.

However, the above conclusion is not valid in a real-time system

(timed system). One reason is that time constraints in a timed system falsify some equivalent paths and thus they cannot be executed in a real environment. If there are some parameters in time constraints, the precondition of a path could contain a predicate which limits the possible value of the parameters. Different paths have different parameter predicates. Therefore, it is much more complicated to calculate the precondition of a partial order in a timed system than in an untimed system.

After the first paper introducing the untimed path condition was published [Dij75], weakest precondition for timed system has been studied in [BMS91, HNSY94, SZ92]. The paper [BMS91] extended the guarded command language in [Dij75] to involve time. But it only investigated sequential real-time program. The paper [SZ92] gave definition of the weakest precondition for concurrent timed program, based on discrete time, not dense time. The weakest precondition in [HNSY94] is defined in guarded-command real-time program or equivalently timed safety automata. None of these papers worked on path condition under partial order. We present a methodology in this thesis to compute the precondition of a partial order in a timed system.

1.1.4 Probability of real-time traces

The fourth problem is relevant to the second and the third problem. When we describe the second problem, we mentioned that the specified path (or a partial order) might not be executed due to nondeterministic choices. For a discrete system, we proposed using code transformation, i.e., adding extra code to the original code, to ensure the execution of the specified path.

In real-time systems, however, adding code that controls the execution to capture the particular scenario typically changes the time constraints and therefore the checked system. A naive solution can be to try executing the system several times given the same initial conditions. But it is not a priori given how many times one needs to repeat such an experiment, nor even if it is realistic to assume that enough repetitions can help. When every non-deterministic choice is decided according to a probabilistic distribution, one can compute the probability of executing the suspicious trace and therefore decide the times we expect to run the system until the scenario occurs. Another important issue is the frequency of the occurrence of problems found in the code. In some cases, it is more practical and economical to develop recoverable code than foolproof one. Given a discovered failure, the decision of whether to correct it or let a recovery algorithm try to catch it depends on the probability of the failure to occur. In such cases, an analysis that estimates the probability of occurrence of a given scenario under given initial conditions is necessary. Consider a typical situation that a bug is found in a communication protocol. This bug causes a sent packet to be damaged. If the chance of the bug being triggered is very small, e.g., one packet out of 100,000 can be damaged, one would allow retransmitting the damaged packet. On the contrary, if its probability is one out of 10, one would redesign the protocol.

Now the problem is to estimate with what probability the system executes the specified path under the given initial condition. We assume that each transition that the system can make must be delayed for a random period, which is bounded by a lower and upper time limit and obeys a

continuous probability distribution. Such an assumption has gained more and more attention, e.g., [ACD91, BD04, KNSS00]. In the literature, many probabilistic systems have been proposed. Probabilistic timed automata (PTA) have been studied by [KNSS02, Seg95]. PTA in both of these papers have discrete probabilistic choices. Thus probability can be calculated using Markov chains. However, when the execution of a transition in a path depends on part of, or even the entire execution history before this transition, the probabilistic model we define is not Markovian[1] if not all of distributions are exponential. The work in [BHHK03] considered continuous-time Markov chains (CTMC), which can only be generated if all distributions in the model are exponential distributions.

In order to allow general continuous distributions, a semi-Markov chain model, which is an extension of CTMC, was studied in [LHK01]. The models defined in [ACD91, BBD03, KNSS00, Spr04, YS04] are similar to ours. These works proposed to model systems by generalized semi-Markov processes (GSMP) [Whi80, Gly89], a compositional extension of semi-Markov chains. However, the processes have uncountably many states. In [ACD91], the processes are projected to a finite state space, which, unfortunately, is not Markovian. Although it is possible to approximate the probability of a particular path in this finite state space, the calculation would suffer from high complexity. In [YS04], the GSMP is approximated with a continuous-time Markov process using phase-type distributions and thereafter turned to a discrete-time Markov process by *uniformization*. However,

[1] A model is Markovian if at any time, the probability of executing a new transition does not depend on the execution of previous transitions and thus the probability of a path can be computed in a stepwise manner.

this method gives only approximate results. The algorithm in [KNSS00] adopted a similar technique to calculate the probability. Since the time delay for any transition on the path from the beginning of enabledness to fire is independently determined, it is intuitive to perceive that the probability to execute the path can be calculated through integration on a multidimensional random variable. A method using integration to model checking stochastic automata was informally discussed in [BBD03] through an example. No concurrency was shown in the example since it contained only one automaton. An integration formula is presented in [Spr04] for the probability of executing a transition. However, no discussion on computing that formula was given in the paper. Similarly, the semantics of the GSMP model has been studied in the context of process algebra, e.g. [KD01, RLK$^+$01], but no means of computing the probability of a given path has been reported.

Therefore, based on the methodology for the third problem, this thesis suggests a methodology to calculate the exact value for the probability of executing a path using integration. Moreover, our methodology can be optimized for the probability of a partial order in some cases.

1.2 Contribution

The main theoretical contribution of this thesis is composed of four methodologies in terms of the above four problems respectively.

- Our method utilizes human intelligence to limit the state space searched automatically by model checkers, i.e., it provides a way for users to guide model checkers to search only the part of the state space they

are interested in. The approach is based on adding annotations to the verified code. The annotations change the behavior of the checked code in a controlled way, allowing avoiding part of the verification performed by model checkers. We suggest new programming constructs for representing annotations, which are not part of the verified code. They are used for obtaining information during the model checking. The information is collected using new variables, disjoint from the original program variables. Though such variables are updated only within the annotations, they can be updated by the value of program variables. The annotations can then control the search using the gathered information, such as forcing immediate backtracking, committing to the nondeterministic choices made so far or terminating the search and reporting the current executions.

A related method was developed in parallel by Gerard Holzmann for his new version of SPIN (described in the new reference manual [Hol03]). The new SPIN version is capable of annotating the checked code using C commands, including adding new variables to support the annotation. One difference between the SPIN annotation and ours is that we allow structural annotation, namely, annotating programming structures, e.g., as all the actions associated with a `while` loop, and the nesting of annotations. We also separate in the annotations the enabledness condition from the code to be applied. Our annotation methodology is strongly based upon this feature, allowing us to control the search by making some search directions disabled through the annotations.

- In order to recover the suspicious behavior, this thesis suggests an automatic transformation that can be applied to the code so that the code is forced to behave according to a given execution. The transformation is analyzed in the framework of partial order semantics and equivalence between execution sequences. It has the following properties: minimize the changes to the system; enforce the specified execution exactly under an appropriate initial condition; preserve any concurrency or independence between executed actions; maintain the checked property; apply the construction to a finite representation of infinite execution. This approach gives testers a tool for checking and demonstrating the existence of the bug in the code.

- We model concurrent systems using transition systems[2]. This model is quite detailed and realistic in the sense that it separates the decision to take a transition from performing the transformation associated with it. Thus separate time constraints, lower bounds and upper bounds, are allowed for both parts. Alternative choices in the code may compete with each other and their time constraints may affect each other in quite an intricate way. The transition systems are translated into a collection of extended timed automata, which are then synchronized with constraints stemming from the given execution sequence. We then obtain a directed acyclic graph of executed transitions and use it to calculate weakest precondition through time zone analysis. This methodology can be adopted to automatically generate test cases.

[2] "Transition system" is a standard notation in verification, which abstracts away from particular syntax.

During the implementation of this methodology, a paradoxical problem caused by time constraints was identified. If the execution time of all actions belonging to one process in the path is shorter than that of actions belonging to another process, then the path condition could be *false*. The reason is that the first process is required by time constraints to execute additional actions on top of those appearing in the path, after it has finished executing its actions in the path but before the second process has finished execution. The *false* path precondition could confuse testers in the sense that they would believe that some sequence of actions cannot be executed and draw a wrong conclusion about the path. In addition, this problem can be extended to partial orders naturally. It forces testers to be very careful when specifying a partial order to compute its precondition in timed systems. We suggest a remedy method to assist testers in dealing with the problem. Moreover, the remedy method is helpful to simplify computation in some circumstances, one of which is identified in this thesis.

- The timed systems studied have a characteristic that every transition must be enabled for a period of time before it is triggered. The period is bounded by a lower bound and an upper bound and the length of the period is probabilistically distributed between the lower bound and the upper bound. The probability of execution of a given path from a given initial condition depends not only on the given scenario, but also on other probabilistic choices available (but not taken) by the analysed system. Thus, the probabilistic analysis of a path involves considering large parts of the code, rather than only the transitions

participating in the given scenario. We present the timing relation among transitions which contribute the probability of a path and propose an algorithm to use integration to calculate the probability. Then based on the probability calculation for a single path, we give a general algorithm to calculate the probability of a partial order. Furthermore, we analyze the possibility and difficulty of optimizing the calculation for partial orders and propose a heuristic algorithm with respect to synchronous communications.

1.3 Thesis Outline

There are nine chapters in this thesis. We provide an overview of preliminaries for this thesis in Chapter 2. Chapter 3-8 are the main part of this thesis. Note that we prefer to keep the presentation at the intuitive level rather than on the theoretical-formal one. We discuss the first two interesting problems in Chapter 3 and 4 respectively. Then, based on the discussion of a real-time system model in Chapter 5, the last two problems are discussed in Chapter 6 and 7, respectively. Chapter 8 is dedicated to the implementation of our methodologies for these problems. The last chapter is the conclusion of this thesis. A short description for each chapter is also given in the rest of this section.

- Chapter 2 offers the reader the necessary background knowledge for this thesis. It reviews the existing models of systems: flow charts, timed automata and timed transition diagrams. A flow chart is an untimed model used in Chapter 3 and 4, and it is translated into timed

models in Chapter 5 for the work in subsequent chapters. The last two models form the basis of Chapter 5, which provides timed models studied in Chapter 6 and 7. Chapter 2 also explains the basic algorithms used in explicit state model checking, such as depth first search and breadth first search, which provide the reader with preliminaries to understand Chapter 3. The definition of a partial order, which is employed by Chapter 4, 5, 6 and 7, is presented in Section 2.3. We demonstrate in Section 2.4 calculating the weakest path precondition in an untimed system, which is also used in calculating path precondition in a timed system in Chapter 6. Section 2.5 gives the presentation of symbolic execution of a program, which is used in Chapter 7 to collect time constraints of a path. Some probability concepts are introduced in Section 2.6 to provide theoretical support for Chapter 7. Finally, a tool PET (Path Exploration Tool) is described since the methodologies in Chapter 4, 5, 6 and 7 were implemented in it. Other tools which were used in implementation, such as SPIN and DOT, are introduced as well.

- Chapter 3 describes the methodology to limit state space search in explicit state model checking. We first introduce the extra data structure for annotations. There are two kinds of variables, history variables and auxiliary variables, and four control constructs: `commit`, `halt`, `report` and `annotate`, which are defined by a BNF grammar. Then we prove that the annotations preserve the viability of counterexamples, which gives us confidence that we obtain correct result from our methodology. We also did an experiment to demonstrate the potential

and flexibility of annotations.

- Chapter 4 discusses the way to guarantee an execution of a path with a specified essential partial order. We start with the description of the data structure to transform programs with shared variables. Basically, we use semaphore operations to construct the structure. There are three kinds of assignments: *Free*, *Wait*, and counter increment. These assignments are inserted into code through conditional statements. Meanwhile, we prove that the code transformation preserves the checked property in order to make sure that the essential partial order is retained. Furthermore, we indicate how to transform another kind of computational model which uses message passing instead of shared variables and argue that our methodology can be applied to infinite traces.

- Chapter 5 provides a timed model for Chapter 6 and 7. At the beginning, we define transition systems, which are adapted from timed transition diagrams, and extended timed automata, which augment timed automata with program variables. In addition, we explain the translation from a transition system to an extended timed automaton. Then we reveal in detail how to model shared variables because shared variables are accessed mutually exclusively and how to model synchronized communication transitions. We also give the algorithm of intersecting the product of extended timed automata with partial order automaton to generate a directed acyclic graph (DAG). We finish Chapter 5 with an illustration of the whole procedure from transition

systems to DAGs by an example.

- A methodology is proposed in Chapter 6 to calculate the weakest precondition of a partial order in timed systems modeled by transition systems. At first, based on the definition of the weakest path precondition, we introduce the untimed precondition of a DAG. Afterwards, timing information is added to the DAG to obtain its timed precondition. The data structure representing timing information is presented first and then the algorithm to calculate the precondition of a DAG is provided. Moreover, we demonstrate by an example the paradoxical problem mentioned in the previous section when specifying a partial order. We give its formal definition and propose a remedy method for this problem. We also suggest that the remedy method can be applied to simplify the computation of the maximum and the minimum bounds of time parameters.

- Chapter 7 suggests a methodology to calculate the probability of executing a path in a timed system, which is modeled by transition systems as well. We start with an introduction to the probabilistic behavior of the system by associating a count-down clock with every transition. Here the concept of transition is slightly different from the one in previous chapters. Secondly, we analyse the timing relations of a given path. Thirdly, we give the formula to compute the probability with respect to the timing relations, which is illustrated by a detailed example. At last, we describe how to calculate the probability for a partial order. We indicate that it is very complicated to acquire the

timing relations of a partial order in the presence of shared variables and synchronous communications, and therefore we only apply optimization to the calculation for synchronous communications rather than the general case.

- Chapter 8 mainly introduces the timed extension of PET (a validating tool set) via some screen shots. A review of existing work which has been done in the original PET is given in the first section. Then the implementation of code transformation in Chapter 4 is explained. Finally, the features and the structure of the timed extension of PET and the implementation details of the extension are presented.

- Chapter 9 concludes the thesis by summarizing the research presented in this thesis and discussing the potential directions for future research and for enhancing the functionalities of the tool set.

Note that many figures in this thesis are generated by PET and its extension. Those figures are indicated in the summary section of each chapter. Other figures were drawn manually.

Chapter 2

Background

This chapter provides a brief introduction to background knowledge for the rest of the chapters. Existing models, including discrete model *flow charts* and real-time models *timed transition diagrams* and *timed automata*, are presented. We also introduce explicit state model checking, partial order, untimed path precondition, symbolic execution of an untimed program, basic probability theories, and related software tools.

2.1 Existing models

2.1.1 Flow charts

Flow charts, which are directed graphs, were first introduced in [Flo67]. A sequential program can be represented by a flow chart, where the control flow of the program is clearly displayed by edges. There are four kinds of basic nodes in a flow chart. Each flow chart has a *begin* node, which indicates the entry of the program; an *end* node, which indicates the end of the

program; a number of *assignment* nodes, each of which represents an assignment statement; a number of *branch* nodes, each of which represents a branch structure. Every node is allocated a *program counter* (PC) value labeling it when translating code into a flow chart. A directed edge connects two nodes such that it departs from the *source* node and points to the *target* node. It indicates that the program counter ranges from one node to another.

The *begin* node has one outgoing edge which points to the first statement of the program and no incoming edges. It is depicted by an ellipse. The *end* node, which is also depicted by an ellipse, does not have outgoing edges and could have multiple incoming edges. If no statement can be executed after the execution of a statement, that statement has an outgoing edge pointing to the end node. An assignment has the form $v := expr$, where v is a program variable and $expr$ is an expression. An *assignment* node, whose shape is a rectangle, has one outgoing edge and may have multiple incoming edges. A branch structure has a first order predicate and two branches. If the predicate is evaluated to *true*, then the control flow proceeds via one branch; otherwise, the control flow proceeds via the other branch. Therefore, a branch node has two outgoing edges, which are labeled as *true* and *false* respectively, and might have multiple incoming edges. Its shape is a diamond. Figure 2.1 referenced from [Pel02] demonstrates the four kinds of nodes.

Both the condition statements, such as "if-then-else" statements and the loop statements, such as "while" statements, can be translated into the combination of *branch* nodes and *assignment* nodes. Figure 2.2 shows the translation of an "if c then S_1 else S_2" statement (in the left) and a "while(c)

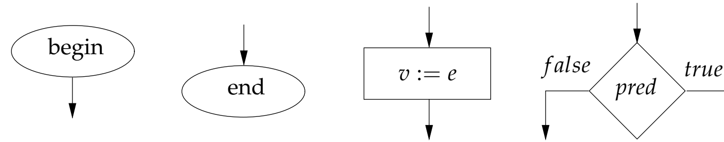

Figure 2.1: Flow chart nodes

S" statement (in the right).

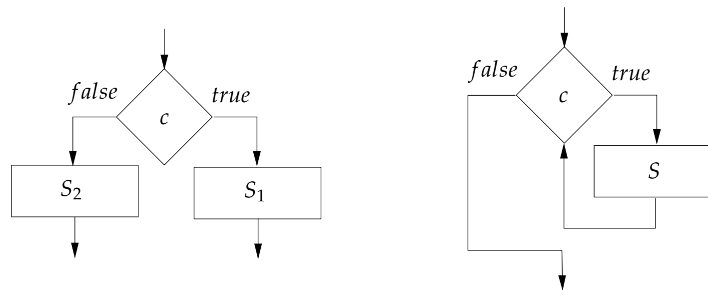

Figure 2.2: Translation of *branch* statements

A *path* of a sequential program is a consecutive sequence of nodes in the flow chart. The projection of an execution sequence of a concurrent program on each program counter is a path through the nodes labeled with values of this program counter in the corresponding flow chart.

2.1.2 Timed automata

Timed automata (TA) were introduced in [AD94] for the first time. Since then, the formalism has become a popular technique to model timed systems. A timed automaton is a tuple $\langle \Sigma, T, A, S, S^0, E \rangle$, where

- A finite set Σ of labels, ranged over by l.
- A finite set T of clocks, ranged over by t.

- A finite set A of assertions over clocks. Each assertion is of the form

$$\varphi := t \leq c \mid t < c \mid t \geq c \mid t > c \mid \varphi_1 \wedge \varphi_2,$$

  where c is a constant.

- A finite set S of locations. Every $s \in S$ is associated with an assertion $I(s) \in A$, which must be always satisfied within any state which contains s. $I(s)$ is known as *location invariant*.

- A set $S^0 \subseteq S$ of initial locations.

- A finite set $E \subseteq S \times \Sigma \times A \times 2^T \times S$ of actions. An action $\langle s, l, \psi, \lambda, s' \rangle$ provokes the change of locations from *source location* s to *target location* s'. It is labeled as l, is associated with ψ to ensure that the action can be fired only when ψ holds, and resets every clock in the set λ.

A state of a timed automaton contains a location and assigns a real value to each clock $t \in T$. A state is an initial state if it contains an initial location and the value of every clock in T is 0. Let $t(q)$ be the value of the clock t in the state q containing location s. The state q can be changed to q' which contains s' due to one of two kinds of transitions:

- Elapse of time such that $s = s'$ and $t(q') - t(q) = \delta$ for any clock $t \in T$ and a real value $\delta \geq 0$. For any real value δ' such that $0 \leq \delta' \leq \delta$, $I(s)$ holds when each clock's value in q is increased by δ'. This kind of transition is denoted as $q \xrightarrow{\delta} q'$.

- Execution of an action $\langle s, l, \psi, \lambda, s' \rangle$ such that the clock values in q satisfy ψ and $t(q') = 0$ for all clocks $t \in \lambda$ and $t(q') = t(q)$ for all clocks $t \in T - \lambda$.

A run of a TA is a finite or infinite sequence $\rho = q_0 q_1 q_2 \ldots$ of states iff the sequence satisfies the following requirements:

- q_0 is an initial state.
- For all $i \geq 0$, there is a transition τ_i changing state q_i to q_{i+1} such that τ_i belongs to one of the two kinds of transitions above.
- If ρ is an infinite sequence, for all transitions representing elapse of time $\tau_i = q_i \xrightarrow{\delta_i} q_{i+1}$ $(i \geq 0)$, $\sum_{i \geq 0} \delta_i$ diverges.

2.1.3 Timed transition diagrams

In [MP92], untimed concurrent systems were modeled by transition diagrams. Each process is represented by a transition diagram, which is a directed graph. A node is a program control location. A directed edge, as shown in Figure 2.3, connecting two nodes represents a transition from the source node to the target node, labeled as $c \rightarrow f$ where c is the enabling condition and f is the transformation. The transformation might be executed only if the program control is located in its source node and the enabling condition c is evaluated to *true*. More than one edge may start at the same location so that a process is able to have nondeterministic behaviors.

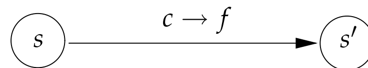

Figure 2.3: An untimed transition

In order to model timing characteristics, time was incorporated into transition diagrams in [HMP94]. A transition $c \rightarrow f$ is associated with a pair

of time bounds $[l, u]$ where l is the lower bound and u is the upper bound. Figure 2.4 illustrates a timed transition. To execute its transformation, the process must reside at its source node and during the residence the condition c holds continuously for at least l time units and at most u time units. Then the transformation is executed instantaneously without time elapse.

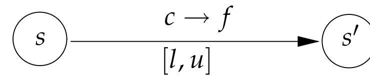

Figure 2.4: A timed transition

2.2 Explicit state model checking

Explicit state model checking is an automatic verification technique [VW86], which is usually based on a search through the state space of the analyzed system. The system is translated first into a collection of *atomic actions* [MP83, Pel02] (or simply *actions*). Atomic actions are the smallest visible units that can be observed to induce a change in the system. A *state* represents the memory of the system at some point of the computation. It is often described as a mapping from the program variables, the interprocess message queue and the program counters into their values. An action contains two parts: a *condition*, and a *transformation*. An action is *enabled* at a state, if its condition holds for that state. Then it can be executed, in which case the transformation is applied to the state. By applying the transformation of an enabled action to a state we obtain a new state, usually causing at least one of the program counter values to be changed. Concurrent systems often allow *nondeterminism*, when there is a choice of more than a single atomic ac-

tion enabled from some states. Nevertheless, each action is itself deterministic, i.e., when applied to the same state it will always generate the same successor. Some states are distinguished as the *initial states* of the system.

An *execution* is a finite or infinite alternating sequence of states and actions $s_0\, \alpha_0\, s_1\, \alpha_1\, \ldots$ where (1) s_0 is an initial state, (2) α_i is enabled at the state s_i, for $i \geq 0$, and (3) s_{i+1} is obtained from s_i by applying the transformation of α_i, for $i \geq 0$. We denote the set of executions of a system A by $\mathcal{L}(A)$ (the *language* of A). Depending on the specification formalism used (e.g., linear temporal logic [Pnu77]), we may decide to include in $\mathcal{L}(A)$ only the sequences of states or the sequences of actions, projecting out the action or state components, respectively. The state space of a system is a graph $\langle S, E \rangle$, where the nodes S represent the states and the directed edges E are labeled by actions, such that $s \xrightarrow{\alpha} s'$ when $s' \in S$ is obtained from $s \in S$ by applying α. Let $S^0 \subseteq S$ be the set of initial states. An execution is hence represented as a path in the graph, starting from an initial state. A state is *reachable* if it appears in some execution of the system. A search through the state space of a system can be performed in order to exercise the code for testing, to check violations of some invariants, or to compare the collection of execution sequences with some system specification.

For a concurrent system with multiple parallel processes, we obtain the collection of all the actions of the various processes. An execution is still defined in the same way, allowing actions from different processes to interleave in order to form an execution. In some cases, we may want to impose some *fairness* constraints [Fra86], disallowing an execution where, e.g., one process or one transition is ignored from some state continuously

(i.e., in every state) or infinitely often. According to [Pel02], there are four kinds of fairness. *Weak transition fairness* prohibits the situation where a transition is enabled but not executed forever from a time point on; *strong transition fairness* requires that if a transition is enabled infinitely often, it must be executed; *weak process fairness* does not allow the case that, from a time point on, a process is always enabled[1] but no any transitions in it are executed forever; *strong process fairness* requires that if a process is enabled infinitely often, one of its transitions must be executed. Note that we did not impose maximality or fairness on sequences in this thesis because it is not necessary when we concentrate on checking safety.

A search of the state space has several distinct parameters:

- The direction of the search. A *forward* search often starts from the initial states and applies actions to states, obtaining their successors [Hol03]. Applying an action to a reachable state guarantees obtaining a reachable state. A *backward* search is applied in the reverse direction, and does not necessarily preserve reachability [McM93].

- Level of abstraction of search. In an explicit search [Hol03, Kur95], we usually visit the states one at a time, and represent them individually in memory. In symbolic search [Hol03] we represent a collection of states, e.g., using some data structure or a formula. By applying an atomic action, we obtain a representation of a new collection of states (either the successors or the predecessors of the previous collection).

- The search strategy. Algorithms such as *depth first search* (DFS) [Hol03]

[1] A process is enabled if one of its transitions is enabled.

or *breadth first search* (BFS) [Kur95] can be used. Different search strategies have distinct advantages and disadvantages. For example, BFS can be used to obtain a shortest execution that violates some given specification. On the other hand, DFS can focus more efficiently on generating the counterexample.

- The reduction method. Because of the high complexity of the search, we often apply some techniques that reduce the number of states that we need to explore. Reduction methods are based on certain observations about the nature of the checked system, e.g., commutativity between concurrent actions, symmetry in the structure of the system [CEJS98], and data independence [LN00, Wol86]. The goal of the reduction is to enhance the efficiency of the search and be able to check bigger instances, while preserving the correctness of the analysis.

2.3 Partial order

Partial order was proposed first in [Lam78] and has been studied intensively in many areas. Here we give an introduction to partial order on executions.

A concurrent system is composed of a group of processes $p_1, \ldots, p_n$. Each process is represented by a flow chart. A flow chart node is an untimed action. We denote by $A(p_i)$ the actions belonging to the process p_i. A pair of synchronized actions, e.g., message passing actions, is deemed as a shared action which belongs to two processes. Let $A = A(p_1) \cup \cdots \cup A(p_n)$ be the set of actions in the system. The *dependency relation* $D \subseteq A \times A$ is

a reflexive and symmetric relation over the actions. It captures the cases where concurrent execution of actions cannot exist. Thus, $(\alpha, \beta) \in D$ when

- α and β are in the same process, or
- α and β use or define (update) a mutual variable[2].

This dependency corresponds to the execution model of concurrent programs with shared variables. Let ρ be a path, i.e., a sequence of actions from A. We denote by α_k the kth occurrence of action α in the sequence ρ. Thus, instead of the sequence of actions $\alpha\alpha\beta\alpha\beta$, we can equivalently denote ρ as the sequence of occurrences $\alpha_1\alpha_2\beta_1\alpha_3\beta_2$. We denote the set of occurrences of a sequence ρ by E_ρ. In the above example, $E_\rho = \{\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2\}$.

Now we define a binary relation $\rightarrow_\rho$ between occurrences on a sequence ρ. Let $\alpha_k \rightarrow_\rho \beta_l$ on a sequence ρ when the following conditions hold:

1. α_k occurs before β_l on ρ.
2. $(\alpha, \beta) \in D$.

Thus, $\alpha_k \rightarrow_\rho \beta_l$ implies that α and β refer to the same variable or belong to the same process. Because of that, α_k and β_l cannot be executed concurrently. According to the sequence ρ, the imposed order is α_k before β_l. Let $\rightarrow_\rho^*$ be the transitive closure completion of $\rightarrow_\rho$. This is a partial order, since it is transitive, asymmetric and irreflexive. The *partial order view* of a path ρ is $\langle E_\rho, \rightarrow_\rho^* \rangle$.

[2]depending on the hardware, we may allow α and β to be concurrent even if both only use a mutual variable but none of them define it.

However, the relation $\rightarrow_\rho$ contains many pairs of $\alpha_k \rightarrow_\rho \beta_l$. We reduce $\rightarrow_\rho$ by removing pairs of actions $\alpha_k \rightarrow_\rho \beta_l$ that have a chain of related (according to $\rightarrow_\rho$) occurrences between them. The *reduced* relation between occurrences of ρ is denoted by $\rightsquigarrow_\rho$. It is defined to be the (unique) relation satisfying the following conditions:

1. The transitive closure of $\rightsquigarrow_\rho$ is $\rightarrow_\rho$.
2. There are no elements α_k, β_l and γ_m such that $\alpha_k \rightsquigarrow_\rho \beta_l$, $\beta_l \rightsquigarrow_\rho \gamma_m$ and $\alpha_k \rightsquigarrow_\rho \gamma_m$.

Calculating the relation $\rightsquigarrow_\rho$ from $\rightarrow_\rho$ is simple. We can adapt the Floyd-Warshall algorithm [Flo62, War62] for calculating the transitive closure of $\rightarrow_\rho$. Each time a new edge is discovered as a combination of existing edges (even if this edge already exists in $\rightarrow_\rho$), it is marked to be *removed*. At the end, we remove all marked edges.

Although an execution is represented by a sequence, we are in general interested in a *collection* of *equivalent* executions. To define the equivalence between executions ρ and σ, let $\rho|_{\alpha,\beta}$ be the projection of the sequence ρ that keeps only occurrences of α and β. Then $\rho \equiv_D \sigma$ when $\rho|_{\alpha,\beta} = \sigma|_{\alpha,\beta}$ for each pair of interdependent actions α and β, i.e., when $(\alpha, \beta) \in D$. This also includes the case where $\alpha = \beta$, since D is reflexive. This equivalence is also called *partial order* equivalence or *trace* equivalence [Maz86]. It relates sequences ρ and σ for the following reasons:

- The same occurrences appear in both ρ and σ.
- Occurrences of actions of a single process are interdependent (all the actions of a single process use and define the same program counter).

Thus, each process executes according to both ρ and σ the same actions in the same order. This represents the fact that processes are sequential.

- Any pair of dependent actions from different processes cannot be executed concurrently, and must be sequenced. Their relative order is the same according to both ρ and σ.

- Occurrences of independent actions that are not separated from each other by some sequence of interdependent actions can be executed concurrently. They may appear in different orders in trace-equivalent executions.

2.4 Path precondition

Path precondition was first studied in [Dij75]. A program is composed of a group of *guarded commands* of the form $\langle guard \rangle \rightarrow \langle statement \rangle$. A guard is a boolean expression and a statement can be an assignment, a condition statement, a loop, or another guarded command. A state is a mapping from program variables and program counters to values. The statement can be executed in a state only if the guard is satisfied in that state. A path as a sequence of flow chart nodes can be seen as a sequence of guarded commands. An *assignment* node is a guarded command with the guard *true*. A *branch* node is a guarded command with a null statement. Its guard is the predicate if the edge labeled *yes* is chosen; otherwise, its guard is the negation of the predicate. When a condition R expressed as a first order predicate is

given after the execution of a path ρ, the *weakest path precondition*, denoted by $wp(\rho, R)$, is the necessary and sufficient condition (also expressed as a first order predicate) that guarantees R is satisfied in the state reached by the execution of ρ. R is said to be a postcondition of a path. $wp(\rho, R)$ is calculated from the last command to the first command on the path. Indeed, any guarded command can be translated into a set of simple guarded commands whose statements are a multiple assignment to program variables. The translation is similar to the one that translates "if-then-else" statements or "while" statements into flow charts. Therefore, a path is a sequence of simple guarded commands. For a simple guarded command $c \rightarrow v := expr$ and a postcondition R,

$$wp(c \rightarrow v := expr, R) = c \wedge R[expr/v],$$

where $R[expr/v]$ denotes the condition obtained by replacing every occurrence of v in R by $expr$. For a path $\rho = S_1 S_2 \ldots S_n$,

$$wp(\rho, R) = wp(S_1, wp(S_2, wp(\ldots, wp(S_n, R) \ldots))).$$

In deterministic code, when we start to execute the code from the first node on the path in a state that satisfies the path precondition, we are guaranteed to follow that path. In case of nondeterminism, the execution of a path is not guaranteed any more due to different schedules.

2.5 Symbolic execution

Symbolic execution of a program was first studied in [Kin76], where the initial condition of the program is *true* and only integers and their expressions

are considered. A sequential program can be expressed by assignments and "If-then-else" statements (like a flow chart). An input is supplied as a symbol (or a constant in trivial cases). Thus, the value of a program variable is an integer polynomial. A state of a program execution includes the values of program variables and the program counter. The symbolic execution of an assignment $v := expr$ replaces the old value of v (which can be a program variable or the program counter) in a state by the new value *expr*. Of course, *expr* is composed of input symbols and constants.

Let φ be the accumulated condition which is satisfied by input symbols when executing the program. As mentioned above, φ is initialized as *true*. φ is not changed when executing an assignment. Let q be the condition in an "If-then-else" statement. When executing this statement, $\varphi = \varphi \wedge q$ if $\varphi \wedge q$ is satisfied; or $\varphi = \varphi \wedge \neg q$ if $\varphi \wedge \neg q$ is satisfied. In this case, the execution of the "If-then-else" statement is a *nonforking* execution. When neither $\varphi \wedge q$ nor $\varphi \wedge \neg q$ is satisfied, there is one set of values in the domain of inputs satisfying the former and another set satisfying the latter. Hence, the execution is split into two parallel executions. One execution follows the "then" branch and the other follows the "else" branch. The execution in this case is a *forking* execution.

2.6 Probability theory

The content of this section is based on Chapter 3 of [Bor98]. Let $Y_1, \ldots, Y_n$ be independent continuous random variables on a common probability space. Let $y_1, \ldots, y_n$ be their values and $\hat{f}_1(y_1), \ldots, \hat{f}_n(y_n)$ be their density func-

tions. The n-dimensional vector $(Y_1, \ldots, Y_n)$ is called a *multidimensional random variable*, whose domain is a set of ordered vectors $(y_1, \ldots, y_n)$. Its density function is $\hat{f}(y_1, \ldots, y_n)$. The distribution of the vector, also called the *joint distribution* of these variables, is the probability distribution over the region defined by

$$P(B) = P((Y_1, \ldots, Y_n) \in B),$$

where B is a region in the n-dimensional space. The density function of the vector is

$$\hat{f}(y_1, \ldots, y_n) = \hat{f}_1(y_1) \times \cdots \times \hat{f}_n(y_n).$$

Therefore, $P(B)$ is given by the following integral:

$$P(B) = \int \cdots \int_{(Y_1, \ldots, Y_n) \in B} \hat{f}(y_1, \ldots, y_n)\, \mathrm{d}y_n \cdots \mathrm{d}y_1.$$

The probability density function for the continuous uniform distribution on the interval $[l, u]$ $(l < u)$ is

$$\hat{f}(x) = \begin{cases} \frac{1}{u-l} & l \leq x \leq u; \\ 0 & \text{elsewhere.} \end{cases}$$

For the sake of simplicity, we say $\hat{f}(x)$ is $\frac{1}{u-l}$ in this thesis.

Suppose $Y_1, \ldots, Y_n$ are continuous random variables with joint density $\hat{f}(y_1, \ldots, y_n)$, and random variables $X_1, \ldots, X_n$ are defined as $X_1 = g_1(Y_1, \ldots, Y_n), \cdots, X_n = g_n(Y_1, \ldots, Y_n)$. The joint density of $(X_1, \ldots, X_n)$ is then given by $\hat{f}(y_1, \ldots, y_n)|J|$ where $|J|$ is the determinant of the Jacobian of the variable transformation, given by

$$J = \begin{pmatrix} \partial y_1/\partial x_1 & \cdots & \partial y_1/\partial x_n \\ \vdots & \vdots & \vdots \\ \partial y_n/\partial x_1 & \cdots & \partial y_n/\partial x_n \end{pmatrix}$$

2.7 Related software tools

2.7.1 SPIN and PROMELA

SPIN [Hol03] is a model checker originally developed at Bell Labs, which aims to provide efficient software verification rather than hardware verification. The tool has been continuously improved for more than 15 years. It is now an open-source software tool. The release used in this work is Version 4.2.5.

SPIN uses PROMELA (a PROcess MEta LAnguage) as its input language. PROMELA allows nondeterminism in order to model real-world behaviors. Each command in PROMELA can be seen as a guarded command [Dij75]. It also supports I/O operations based on Hoare's CSP language [Hoa85].

In addition to the simulation of behaviors of the system, SPIN can exhaustively verify the specified correctness properties. The theoretical foundation of SPIN for the verification is based on the automata-theoretic approach [VW86]. It can check if a property represented by a Linear time Temporal Logic (LTL) formula [Pnu81] is maintained by a system. The system, described in PROMELA, is modeled by finite-state automata. The negation of the LTL formula is modeled by a Büchi automaton [Tho90]. Then the synchronous product of the automata of the system and the property is generated. If the language accepted by the product is empty, the property holds in the system. Otherwise, an error execution (counterexample), which violates the property, is reported. SPIN employs the explicit state model checking technique to do this automata-based verification. The state

space checked during the verification is generated on-the-fly [Pel94] rather than statically. In order to reduce the state space, SPIN adopts a partial order reduction technique [HP94].

Usually, the model of the system is either written in PROMELA directly or translated from the C, Java or other language code into PROMELA programs automatically, e.g. [HP00]. Since version 4, embedded C code can be included into PROMELA models. This is done by five new primitives: *c_expr*, *c_code*, *c_decl*, *c_state* and *c_track*. Thus it is possible to verify the implementation (in C) of a system directly.

2.7.2 The Omega library

The Omega library [KMP+95] was developed in C++ for Omega Test, one of two major components of the Omega project at the Computer Science Department of the University of Maryland, College Park. The library is used to simplify and verify Presburger formulas. A Presburger formula [KK67] is a formula which only contains affine constraints (either equality constraints or inequality constraints) on integer variables, logical connectives $\neg$, $\wedge$ and $\vee$, and quantifiers $\exists$ and $\forall$. The release used in this work is Version 1.2.

2.7.3 DOT

DOT [GN00] is a tool in the Graphviz software package, which was developed originally at AT&T Research and is now an open source software. DOT defines an input format, which is used to describe a directed graph. The description contains the definition, such as name, label and shape, of

nodes and edges. DOT generates a hierarchical layout of nodes and edges of the graph. The layout contains the screen coordinates of each node and each edge from its description. Then the layout can be displayed on the screen either by DOT or other software. DOT can also convert the layout to other known graphic formats. The release used in this work is Version 2.6.

2.7.4 Pascal

Pascal [JWMM91] is an imperative programming language, which was first developed in 1970. It was named after the mathematician and philosopher Blaise Pascal. Pascal is a structured language, which means that the flow of control of a program is structured into standard statements, such as `if` and `while` constructs, ideally without `goto` statements. It supports scalar variables, such as integer variables and boolean variables, and array variables. It groups a program into procedures, which do not return values, and functions, which return values. The language was originally intended to teach students structured programming. Later it had been broadly used in both teaching and software development. Since new languages, such as C and Java, appeared, Pascal has been adopted less often.

2.7.5 The C Language

The C programming language [KR88] is one of most widely used imperative programming language nowadays, developed in the early 1970s at Bell labs. Since it was used to write the UNIX operating system, it has gained widespread acceptance in software development. C has many sim-

ilar characteristics to those of Pascal. For example, it is structured, it supports many data types similar to Pascal's, and it supports procedural programs. In this thesis, we only consider the basic statements of C, such as compound statements, assignments, conditional statements, loops, and integer variable definitions. These statements can be translated into Pascal easily and vice versa. For example, "{" and "}" are used in C to delimit the beginning and the end of a block, and are translated into `begin` and `end` in Pascal; assignment operator "=" in C is translated into ":=" in Pascal. A statement "`if (condition) S1 else S2`" in C is translated into "`if (condition) then S1 else S2`" in Pascal and `while` statements have the same grammar in both C and Pascal. A variable definition "`int v`" in C is translated into "`var v: integer`" in Pascal. However, C also has some features which do not exist in Pascal. GCC [WvH03] is a popular C compiler used in Linux and Unix systems. The GCC release used in this work is Version 3.2.3.

2.7.6 Tcl/Tk

Tcl/Tk [Ous98] stands for the Tool Command Language and the Tool Kit. They were developed originally at the University of California Berkeley, later at Sun, and now at a company whose name is also Tcl. Tcl is a very simple scripting programming language such that a new Tcl programmer who has experience with other programming languages can learn it very quickly and easily. Tk is a tool kit based on Tcl, providing many reusable graphical components to speed up development of graphical user interfaces. Tcl/Tk has been widely adopted to develop software systems. Programs written

in Tcl/Tk are interpreted by the Tcl interpreter to execute. Thus their execution is slower than the execution of those written in C. But, in general, Tcl/Tk is fast enough for interfaces that do not need great speed and its ease of use more than outweighs the loss of speed. The release used in this work is Version 8.4.11.

2.7.7 Lex & Yacc

Lex and Yacc [LMB92] are acronyms for A Lexical Analyzer Generator and Yet Another Compiler-Compiler, respectively. Lex and Yacc are standard tools in Unix systems. Usually they work together. Lex reads an input file which specifies lexical rules, and generates a lexical analyzer in C. The analyzer scans the source code of a program constructed according to the lexical rules, and decomposes the source code into a sequence of lexical tokens. Yacc generates a parser in C which recognizes a grammar composed of the tokens output by the lexical analyzer. Flex [Pax95], a fast scanner generator, is an open source analyzer generator on Linux which is compatible with Lex. The release of Flex used in this work is Version 1.2 and the release of Yacc is Version 1.9.

2.7.8 SML/NJ

SML stands for Standard Meta Language (for short, Standard ML) [MTH90]. SML is a general-purpose programming language. It combines the elegance of functional programming with the effectiveness of imperative programming. SML supports higher-order functions, which accept functions as pa-

rameters and/or return functions as results. It has strong type checking, which can eliminate many bugs at compile time. Another important characteristic of SML is that it allows programmers to handle symbolic values very easily.

SML/NJ (Standard ML of New Jersey) is an implementation of a modest revision of the language, SML' 97 [MTM97]. It was originally developed jointly at Bell Labs and Princeton University, and is now a joint project between researchers at Bell Labs, Princeton University, Yale University, and AT&T Research. Besides packages which implement the SML specification, SML/NJ contains many additional packages, such as ML_lex [AMT94], which is a lexical analyzer generator for SML, and ML_yacc [TA00], which is a parser generator for SML. ML_lex and ML_yacc have the same functionalities as Lex and Yacc. SML/NJ has been used to develop many large systems, for example, HOL90. The release used in this work is Version 110.0.7.

2.7.9 HOL90

HOL stands for Higher Order Logic. It also represents a family of interactive theorem provers. HOL90 [GM93], developed in SML/NJ at Cambridge University, is one member of the family. It was initially used to generate formal proofs by man-machine collaboration for the specification and verification of hardware designs. Now it is being applied to many other areas, such as verifying operational semantics of programming languages and distributed algorithms. HOL90 has a built-in simplification library, which is used by PET to simplify Presburger formulas. The release used in this work is Version 10.

2.7.10 Mathematica

Mathematica [Wol03] is developed by Wolfram Research Inc. It calculates many mathematical functions, such as integration formulas and differential equations. Mathematica has a graphical user interface, which allows users to input and edit mathematical functions directly by using either graphical or text commands. It also provides a programming interface, which supports C and Java languages. Users' programs can call Mathematica through this interface without users' involvement. The release used in this work is Version 5.1.

2.8 The untimed version of PET

The first version of PET was developed by Elsa Gunter and Doron Peled [GP99, GP00b, GP00a, GP02]. It aimed to provide testers with a method for the analysis of paths in concurrent systems. The basic function of PET is to calculate the weakest precondition of a path selected by testers. The first version works on untimed systems.

2.8.1 The functions of PET

The key function is to calculate the precondition of an untimed path, which is interactively selected by testers. To do that, PET has the following functions.

1. Display of interactive flow charts: PET reads in all the input processes and displays every process in the form of a flow chart. Besides the

four basic nodes explained in Section 2.1.1, PET is enhanced with extra node types: message passing and *wait* statements. Message passing includes a *sending* statement $\hat{\alpha}!expr$ and a *receiving* statement $\hat{\alpha}?v$ (which are untimed versions of synchronized communication transitions in Section 5.5). The *wait* statement only has one behavior that blocks the process it belongs to until a condition is satisfied. The shape of a flow chart node for a message passing statement is a box and that for a *wait* statement is a diamond. Nodes with different shapes are displayed in different colors.

2. Interactive concurrent path selection: PET allows the testers to select a path by mouse clicking. A node is selected by left button clicking. When the path is projected into one process, the projected path must be a consecutive sequence of nodes, i.e., for any node except the last one in the projected path, there is a flow chart edge starting from this node and pointing to the next node in the projected path. If this is not the case, PET does not allow the testers to select it. The last nodes in projected paths are displayed in a special color. The selected path is displayed in a separate window.

3. Modification of a selected path: PET allows the testers to modify the selected path such as by deleting a node from the path, exchanging the order of two adjacent nodes which belong to two different processes, clearing the path and removing from the path all of nodes belonging to a particular process.

4. Computation of the weakest precondition: PET displays the precondi-

tion of the selected path in a window. Every time the path is changed, the precondition is updated.

5. Temporal debugging: PET provides a capability to debug a system with temporal logic properties [GP02]. Linear temporal logic (LTL) formulae are translated into finite-state automata. When the testers input an LTL formula into PET, it searches the state space of the system until an execution satisfying the formula is found. Then the testers can input another LTL formula and PET searches the state space to find another execution satisfying the second formula, starting from the state satisfying the first formula. Each debugging step is to execute the system until an LTL formula is found or an error is reported.

In addition, PET has another function to facilitate the selection of a path. The source code of each process is displayed in a separate window. When the mouse cursor enters a node in the flow charts or the selected path, this node and its corresponding source code are highlighted.

2.8.2 The structure of PET

PET is composed of two parts: the graphic user interface (GUI) and the kernel. The GUI was written in Tcl/Tk since Tcl/Tk is simple and can be mastered easily. Although the Tcl/Tk program is interpreted during execution and then its running is slower than the kernel, the main work is done by the kernel and users are not aware of the relatively slow speed of the GUI. The kernel was programmed in SML. The reason for using SML lies in the fact that it can handle symbolic manipulation with relative ease because of its

capability to use higher-order functions [GP00b]. This capability considerably decreased the programming workload. An example shown in [GP00b] demonstrates this capability: the implementation of a prototype of translating a linear temporal logic formula into a Büchi automaton was done using 200 lines SML code in one day, while the actual implementation in C needed 5000 lines and took over four months.

The GUI and the kernel communicate with each other through a two-way pipe. There are a number of defined commands in the kernel and the GUI respectively. The GUI accepts the testers' commands and transfers them to the kernel in the format required by the kernel. Then the kernel returns the computed data to the GUI.

The translation of source code into flow charts is done by the kernel and each flow chart is written into a file in the input format of DOT software [GN00]. The GUI feeds the flow chart files to DOT and displays the output of DOT. The kernel also outputs two files for each flow chart for the GUI. One file contains the adjacency relation among nodes, which is needed to guarantee each projected path is consecutive. The other file, which contains the coordinates of flow chart nodes in the source code, is used to display highlights. The kernel calls HOL90 to simplify Presburger formulae.

Chapter 3

Safe annotation

The focus in this chapter is on controlling the search with annotations for reducing the amount of time and memory required for the search during model checking. This can allow an automatic analysis of the system even when the size of the state space is prohibitively high for performing a comprehensive search. In this chapter we concentrate on forward explicit search using DFS. Our framework can be applied to BFS or heuristic search [ELLL04] as well. However, in the case of a search other than DFS, the search order is different, which affects the way our method would work.

3.1 Verification of Annotated Code

We allow annotating the checked program with additional code that is applied during the automatic verification process. The suggested syntax of the annotations is presented later in this section. Our syntax is given for C programs, but similar syntax can be formed for other programming languages. We allow two ways of adding annotations to the code. The first way is to

put the annotations in between other program segments (for example, after an `if` statement, and before a `while` loop). An annotation is effective during verification when the program control passes to it, but not when the program control skips over it, e.g., using a `goto` statement.

A second way to annotate the program is to associate an annotation with a construct such as a `while` or an `if` statement. In this case, the effect of the annotation is imposed on each action translated from this construct. For example, if we annotate a `while` statement, the annotation will be effective with actions that calculate the `while` condition (which may be done in one or more atomic actions) and actions that correspond to the `while` loop body.

In order to understand how the annotations work, we need to recall that, before verifying a program, it is translated into a set of *atomic actions*. The annotations are also translated into actions, called *wrap actions*. The granularity of the former kind of actions is important in modeling the program since the model can behave differently with different granularities (see, e.g., [BA90]). Consequently, we often apply restrictions such as not allowing an atomic action to define (change) or use (check) more than a single shared program variable [OG76]. The issue of atomicity does not apply to the wrap actions since they do not represent the code of the checked program. Wrap actions can exploit the full flavor of a sequential part of the original programming language used, including several instructions and in particular using and defining multiple variables.

We allow nesting of annotations. Therefore, some code can be enclosed within multiple annotations. This means that an atomic action can be related to multiple wrap actions. When processing (executing) an atomic

action, the model checker also processes all the wrap actions related to it. Both the atomic and the wrap actions may have a condition and a transformation. Applying a combination of an atomic action and wrap actions to a state requires that the conjunction of *all* their conditions holds in that state. This means that annotating a program may have the effect of blocking some actions that could be executed before annotating. This provision may be used to disallow some useless directions in the search, or to compromise exhaustiveness for practical purposes. This needs to be done carefully, as it can also render the search less exhaustive. When enabled, the transformation part of the atomic action together with all the corresponding wrap actions are executed according to some order, e.g., 'deeper nested actions are executed later'.

The transformation of wrap actions can be an arbitrary code. This may include iterative code, such as a `while` statement. It is the responsibility of the person adding the annotation to take care that they are not a source of infinite loops. We could have enforced some syntactic restrictions on annotations, but iterative constructs inside annotations seem to be useful.

3.1.1 Programming Constructs for Annotations

Programming languages are equipped with a collection of constructs that allow them to be effective (Turing complete). Adding a new construct to a sequential programming language means adding extra convenience rather than additional expressiveness. In concurrent programming languages, additional constructs may also introduce new ways of interaction between the concurrent agents.

We suggest here programming constructs that can help with the program testing or verification search. The verified code itself is often simulated, step by step, by the testing or verification engine. This simulation usually includes saving the current state description, and providing reference points for future backtracking. The additional code annotates the program in a way that allows controlling the search.

Special Variables

Before presenting the new programming constructs, we introduce two new types of variables that can be used with the annotation. We will call the original variables that occur in the verified code, including any variable required to model the behavior of the program (such as program counters, message queues) *program variables*.

1. *History Variables*. These variables are added to the state of the program. Their value can depend on the program variables in the *current checked execution*. Because actions are deterministic, the value of a history variable in a state s_i in an execution $s_0 \alpha_0 s_1 \alpha_1 \ldots s_i \alpha_i \ldots$ is a function of the initial state $s_0 \in I$ and the sequence of actions $\alpha_0 \alpha_1 \ldots \alpha_{i-1}$. Some examples of uses of history variables include:

    - Limiting the number of times some statement can be executed in the currently checked execution prefix.

    - Witnessing that some state property held in some state of the current execution.

Updating the history variables based on values of the program variables is allowed, but not vice versa. When backtracking, the values of the history variables return to their previous values.

2. *Auxiliary Variables*. These variables are updated while the search is performed but are not part of the search space. Thus, unlike history variables, their value is not rolled back when backtracking is performed. Examples for uses of auxiliary variables include:

   - Counting and hence limiting the coverage of some parts of the code to n times during the model checking process.
   - The main loop of the program was executed k times in a previously searched execution sequence. We may want to limit the checks to include only executions where the number of iterations are *smaller* than k. (Note that this requires two new variables: a history variable that counts the number of iterations in the current execution, and an auxiliary variable that preserves this value for comparison with other iterations.)

   The value of the auxiliary variables in a state is a function of the sequence of states as discovered during the search so far, not necessarily limited to a single execution sequence. Because of their ability to keep values between different execution sequences, auxiliary variables are very useful for achieving various test coverage criteria, e.g., they can be added to record how many times program statements have been traversed. Auxiliary variables may be updated according to history and program variables, but not vice versa.

In order to define history or auxiliary variables, the user can prefix the relevant variable declaration with `history` or `auxiliary`, respectively.

New Search Control Constructs

Four new constructs, `commit`, `halt`, `report` and `annotate`, are introduced to annotate programs. We first summarize the new constructs with the following BNF grammar and then give the detail of each contructs. The complete BNF grammar of our implementation of annotations can be found in Appendix A.

stmt ⟶ **annotated** | **basic_stmt**

basic_stmt ⟶ `while ( `**condition**` ) {` **basic_stmt** `}` |
`if ( `**condition**` ) {` **basic_stmt** `}` |
`{` **list_basic_stmt** `}` | . . .

list_basic_stmt ⟶ **basic_stmt** | **list_basic_stmt** `;` **basic_stmt**

annotated ⟶ `with {` **stmt** `} annotate {` **annotation** `}` |
`annotate {` **annotation** `}`

annotation ⟶ **basic_stmt_plus** | `when ( `**condition**` )` **basic_stmt_plus** |
`when ( `**condition**` )`

basic_stmt_plus ⟶ **basic_stmt** | `commit;` | `halt;` | `report ‘‘` **text** `’’;`

`commit`

Do not backtrack further from this point. This construct can be used

when it is highly likely that the current prefix of execution that has accumulated in the search stack will lead to the sought-after counterexample.

`halt`

Do not continue the search further beyond the current state. Perform backtrack immediately, and thereafter search in a different direction. This construct is useful for limiting the amount of time and space used.

`report ''`**text**`''`

Stop the search, type the given text and report the context of the search stack.

`annotate {` **annotation** `}`

Add **annotation**, a piece of code that is responsible to update the history or auxiliary variables. The annotating code may itself include a condition and a transformation, hence it has the form

`when (` **condition** `)` **basic_stmt_plus**

Either the condition or transformation is optional. The transformation in **basic_stmt_plus** can include code that can change the history and auxiliary variables or consist of the new constructs `commit`, `halt` and `report`. Since the transformation can include loops, it is the responsibility of the annotator not to introduce nontermination. If the annotation condition holds, the transformation is executed between two atomic actions, according to the location of the `annotate` construct

within the checked program. Its execution is *not* counted within the number of atomic steps. If the annotation condition does not hold, the current process is stopped, i.e., no more actions in the current process can be executed.

`with` { **stmt** } `annotate` { **annotation** }

Similar to the previous statement, except that this annotation is added to every atomic action translated from **stmt**. The condition part of the annotation is conjoined with the condition of the original atomic action. It is possible that there are several nested annotations for the same atomic statement. In this case all the relevant conditions are conjoined together. If this conjunction holds the collection of statements in the transformation parts of all the relevant annotations are executed according to some predefined order.

Accordingly, we allow nested annotations. On the other hand, the annotating code can include some special commands (`commit`, `halt`, `report`) but cannot itself include an annotated statement. Of course, this is only a suggested syntax. One may think of variants such as allowing to annotate the conditions in `if` and `while` statements, or allowing the new commands to appear embedded within arbitrary C commands inside the annotations.

3.1.2 Examples

1. Consider the following example.

```
with {x=x+y;} annotate {z=z+1;}
```

Ordinarily, we would have translated `x=x+y` into an action with some condition that depends on the program counter value. The translation usually needs to generate a name for the program counter variable (e.g., `pc`) and its values (e.g,. `14`), as these are seldom given in the code. We may obtain the following action, where the condition appears on the left of the arrow '$\Longrightarrow$' and the transformation is given on its right.

```
pc==14 ⟹ x=x+y; pc=15;
```

The annotation is represented as the following wrap action, which is related to the atomic action above.

```
true ⟹ z=z+1;
```

The annotated action is as follows:

```
pc==14 && true ⟹ x=x+y; z=z+1; pc=15;
```

Note that the annotated action is executed as an atomic action by a model checker.

2. Consider now the code

```
with {while (x<=5) {x=x+1;}} annotate {when (t<23)
                {t=t+1;}};
```

The code of this `while` loop is translated into the following actions:

```
pc==17 && x>5 ⟹ pc=19;
pc==17 && x<=5 ⟹ pc=18;
pc==18 ⟹ x=x+1; pc=17;
```

The first action checks the loop-exit condition (which is simply the negation of the loop condition). If this holds, the program counter obtains a value that transfers control to the action outside the loop. The second action checks that the loop condition holds and subsequently transfers control to the first (and only, in this case) action of the loop body. The third action consists of the loop body. Assume that `t` is defined as an auxiliary variable and is initialized to 0. The condition of the wrap action for the annotation is `t<23`, and the transformation increments `t`. This can be used to restrict the above three atomic actions from executing more than 23 times over all the checked executions. If `t` is a history variable, we only allow 23 increments of `t` within any execution. The wrap action is therefore:

```
t<23 ⟹ t=t+1;
```

When we execute the above loop actions, `t<23` becomes an additional conjunct, which needs to hold in addition to the atomic action condition. When it does, `t` is incremented, in addition to the effect of the atomic action transformation. We obtain the following annotated actions:

```
pc==17 && x>5 && t<23 ⟹ t=t+1; pc=19;
pc==17 && x<=5 && t<23 ⟹ t=t+1; pc=18;
pc==18 && t<23 ⟹ x=x+1; t=t+1; pc=17;
```

Thus, when `t` becomes 23, these actions will become disabled. If this is sequential code, this will disable the continuation of the search at this

point, causing an immediate backtrack. In concurrent code, actions from another process may still be enabled.

3. The annotation does not apply only to the body of the loop, but also to the actions associated with controlling it. In order to annotate only the increments of x, we can use the following:

```
while (x<=5) {with {x=x+1;} annotate {when (t<23)
                 {t=t+1;}}};
```

This will result in exactly the same three actions as above (in general, the atomic actions representing the code are independent of the annotating code). We also have the same wrap action as before, namely

```
t<23 ⟹ t=t+1;
```

Only that, this time, this wrap action is associated with the third action, i.e., the one representing the loop body. That is,

```
pc==17 && x>5 ⟹ pc=19;
pc==17 && x<=5 ⟹ pc=18;
pc==18 && t<23 ⟹ x=x+1; t=t+1; pc=17;
```

Note that, if there are other processes, they might not be blocked once t reaches 23, and the search can continue.

4. There is a similar annotation as above:

```
while (x<=5) {with {x=x+1;} annotate {if (t<23)
                 {t=t+1;}}};
```

But the wrap action is different:

```
true ⟹ t<23?t=t+1:skip;
```

When `t<23`, `t` is increased by 1. It does not block the current process when `t>=23`, but rather ignores the condition. This example shows the difference between `when` and `if` in annotation. Besides, an `if` statement can be used as a clause of `when` statement.

5. In order to cause the other processes to be blocked, we can use

```
while (x<=5)
 {with {x=x+1;} annotate {if (t<23) {t=t+1;} else {halt;}}}
```

This will generate another wrap action, namely

```
true ⟹ t<23?t=t+1:halt;
```

In this case, the annotation performs a `halt` when `t` becomes 23, which causes some internal procedure in the search engine to block the search from the current state and induces an immediate backtrack.

3.1.3 Preserving the Viability of Counterexamples

The set of sequences allowed by a specification B, given, e.g., by a state machine (automaton) or a formula, is denoted $\mathcal{L}(B)$. The correctness criterion for a system A to satisfy the specification B is

$$\mathcal{L}(A) \subseteq \mathcal{L}(B). \tag{3.1}$$

It is often simpler and more convenient to provide the complement specification (for a logic based specification, we just have to prefix it with negation) $\overline{B}$, such that $\mathcal{L}(\overline{B}) = \overline{\mathcal{L}(B)}$. We can thus equivalently check for

$$\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) \neq \emptyset. \tag{3.2}$$

The annotations generate a machine (automaton) A'. We do not include in A' sequences trimmed by the annotations, that is, sequences ending with a `halt` instruction or ending due to the disabledness of all actions caused by the addition of the conditions from wrap actions. Reporting such partial sequences can result in a wrong conclusion about that system. For example, if we want to check whether a state with $x > y$ is eventually reached, the language of $\overline{B}$ consists of sequences in which a state with $\neg(x > y)$ always holds. It is possible that due to trimming of a sequence we obtain such a partial sequence, which would otherwise have continued to a state in which $x > y$. Reporting such a sequence as a counterexample would comprise a false negative.

Each execution ρ' of A' is related to some execution ρ of A in the following way: the states of ρ' may include additional variables (history). Furthermore, it is possible that ρ' includes additional actions that do not exist originally in A (due to `annotate` statements without a `with` clause). Let $proj_A(\rho')$ be the projection of ρ' that removes all such variables from the states, and the additional actions. Note that, since the annotations can change only the values of the history and auxiliary variables, removing actions that are related only to the annotation from the projection on the program variables is the same as eliminating some of the 'stuttering' (i.e., the adjacent repetition of the same state) of the execution. Extend now $proj_A$

from sequences to sets of sequences, i.e., let $proj_A(X)$ be $\{proj_A(\rho')|\rho' \in X\}$. The code annotation is designed to have the following property:

$$proj_A(\mathcal{L}(A')) \subseteq \mathcal{L}(A) \tag{3.3}$$

Note that the specification B is compared by the model checker with the projected sequence $proj_A(\mathcal{L}(A'))$. The extra variables and states do not count as part of the execution, but are merely used to control the search. Now, suppose that

$$proj_A(\mathcal{L}(A')) \cap \mathcal{L}(\overline{B}) \neq \emptyset. \tag{3.4}$$

Then it follows from (3.3) that (3.2) holds as well. That is, if a counterexample exists in the annotated program, it also exist in the original program. This means that the annotations do not generate false negatives. They may result in the search being less exhaustive, thus it is not safe to conclude that no erroneous execution exists even if none was found.

3.2 Experiments

3.2.1 Implementation

In addition to the above annotation constructs, we added some new features to the C programming language. This includes allowing concurrent threads and *semaphores* [Dij68]. These constructs allow us to check or test concurrent programs. We compile a collection of C threads, each interpreted as a concurrent process, into a set of actions. This results in two procedures per each action, one representing the condition and the other representing the transformation. The translation reuses text from the original C code, e.g.,

conditions and assignments. In this way we do not transfer the code from one language to another, but rather reuse as much as possible of the original code. This is important as modeling and translating are a common source for discrepancy between the checked code and the model. However, using text from the original code is not enough, e.g., checks and assignments to program counters (as well as message buffers), which do not appear explicitly in the code, are added.

We need to represent the relation between the procedures representing atomic actions and the procedures representing wrap actions. This is a 'many-to-many' relation. Accordingly, a wrap action can be related to several atomic actions, while an atomic action can be annotated by multiple wrap actions. The program and history variables (but not the auxiliary variables) defined in the checked system are represented in the code obtained by the compilation within a single record (structure) `state`. Thus, if the checked code has a definition

```
int x, y, z;
```

then we have a record `state` with integer fields `x`, `y` and `z`. The search stack consists of records of the same type as `state`. The history variables are kept as part of each state, in the same way as the program variables. In this way, when backtracking, the value of the history variables is rolled back. Auxiliary variables are treated in a different way. They are not part of the structure representing the states. Their value is preserved and updated between different execution sequences despite backtracking.

There are three fixed procedures: `halt`, `commit` and `report`. They are called within some of the wrap actions, when such a construct appears

as the corresponding annotation. In this way, once these procedures are called, the search engine takes the appropriate measures to induce immediate backtracking, disallow backtracking from the current state, or stopping the search and reporting the contents of the search stack, respectively.

3.2.2 An experiment

The *Sieve of Eratosthenes* [Nag03] is an algorithm for calculating prime numbers. A parallel version of implementation in PROMELA [Hol03] works with message passing. We chose this example because the partial order reduction works very well on it. i.e., the state space can be reduced to a very small size after applying the partial order reduction [CGMP99]. Therefore, we can show that a good annotation can reduce the state space searched effectively without applying reduction techniques. Our implementation works with shared variables and hence is simulating message passing using semaphore variables (which were added to the language). In its parallel version, there is a leftmost process that is responsible for generating *integer* numbers, starting from 2, and up to some limit P. There are in general N middle processes (and thus altogether $N + 2$ processes), each responsible for keeping one prime number. The ith process is responsible for keeping the ith prime number. Each middle process receives numbers from its left. The first number it receives is a prime number, and it is kept by the receiving process. Subsequent numbers, 'candidates' for prime numbers, are checked against that first number: if dividing a new number arriving from left by the first number gives a remainder of 0, this cannot be a prime number, and hence it is discarded; otherwise the new value is sent to the process on the

right. Thus, numbers that are not prime are being sifted-out. The rightmost process simply accepts values from the left. The first number it receives is kept (this is a prime number), while the other numbers are just being ignored. This allows overflow of numbers, when more prime numbers than $N+1$ are generated by the leftmost process.

The annotations we use are based on the following observation: there is concurrency in the system, e.g., processes on the left are checking some new candidates for prime numbers while processes on the right are still working on previous candidates. We can limit the concurrency and control the number S of candidate values that can be propagated at the same time in the system. We do this by introducing a new history variable `checked`. The value of `checked` is updated using annotations each time that a candidate value is either sifted out or reaches its final destination in a process. Generating a new candidate on the left is controlled by annotation, allowing the new candidates to be at most S values ahead of the currently checked value. We keep S as a constant with which we control the amount of concurrency we allow in the validation search. We list below the sieve program with $N=2$ (note that the experiments were done with $N=3$).

```
main () {
   int q0, q1, q2;
   semaphore w0 = 1, r0 = 0;
   semaphore w1 = 1, r1 = 0;
   semaphore w2 = 1, r2 = 0;
   history int checked = 1;
   thread left {
      int counter = 2;
      while(counter <= 16) {
         P(w0);
         q0 = counter;
         V(r0);
```

```
      with { counter = counter + 1; }
           annotate { when ((counter-checked) <= S); }
   }
}
thread middle1 {
   int myval1, nextval1;
   P(r0);
   with { myval1 = q0; } annotate { checked = checked+1; }
   V(w0);
   while(1) {
      P(r0);
      nextval1 = q0;
      V(w0);
      if((nextval1 % myval1) != 0) {
         P(w1);
         q1 = nextval1;
         V(r1);
      }
      else
         annotate { checked = checked+1; }
   }
}
thread middle2 {
   int myval2, nextval2;
   P(r1);
   with { myval2 = q1; } annotate { checked = checked+1; }
   V(w1);
   while(1) {
      P(r1);
      nextval2 = q1;
      V(w1);
      if((nextval2 % myval2) != 0) {
         P(w2);
         q2 = nextval2;
         V(r2);
      }
      else
         annotate { checked = checked+1; }
   }
}
thread right {
   int next;
```

```
      while(1) {
         P(r2);
         with { next = q2; } annotate { checked = checked+1; }
         V(w2);
      }
   }
}
```

In order to illustrate the effect of the annotation, we also modified the PROMELA program included in the SPIN package to obtain a PROMELA program which has similar behaviors to the above C program[1]. Then we use SPIN to generate its full state space and the reduced state space after applying partial order reduction. The PROMELA program is listed below.

```
int r[3] = 0;
int w[3] = 1;
int q[3] = 0;
int count;
active proctype left () {
    count = 2;
    do
    :: count <= 16 ->
        atomic{ w[0]==1 -> w[0]=0; }
        q[0] = count;
        r[0] = 1;
        count ++
    :: count>16 -> break;
    od
}
active proctype middle1 () {
    int myval1, nextval1;
    atomic{ r[0]==1 -> r[0] = 0;}
    myval1 = q[0];
    w[0] = 1;
    do
    ::  true ->atomic{ r[0]==1 -> r[0] = 0;}
        nextval1 = q[0];
        w[0] = 1;
```

[1]The annotation is not translated since SPIN does not support it.

```
        if
        :: (nextval1 % myval1) != 0 ->
            atomic{ w[1]==1 -> w[1] = 0;}
            q[1] = nextval1;
            r[1] = 1;
        :: else -> nextval1=nextval1;
        fi
    od
}
active proctype middle2 () {
    int myval2, nextval2;
    atomic{ r[1]==1 -> r[1] = 0;}
    myval2 = q[1];
    w[1] = 1;
    do
    ::  true ->atomic{ r[1]==1 -> r[1] = 0;}
        nextval2 = q[1];
        w[1] = 1;
        if
        :: (nextval2 % myval2) != 0 ->
            atomic{ w[2]==1 -> w[2] = 0;}
            q[2] = nextval2;
            r[2] = 1;
        :: else -> nextval2=nextval2;
        fi
    od
}
active proctype right () {
    int next;
    do
    :: true ->atomic{ r[2]==1 -> r[2] = 0;}
        next = q[2];
        w[2] = 1;
    od
}
```

There are several points that need to be noticed:

1. In this PROMELA program, processes communicate with one another by shared variables, while the Sieve program enclosed in the SPIN package uses message passing. The reason to use shared variables is

that we need to simulate our C program as closely as possible.

2. The semaphore operation `P(w)`, which requests the semaphore `w`, is simulated by the PROMELA statement `atomic{ w==1 -> w = 0;};` operation `V(w)` is simulated by `w = 1`. These PROMELA statements behave as binary semaphore operations [BA90]. However, we use Dijkstra's counting semaphore semantics [Dij68] in our implementation. The value of a counting semaphore can be negative such that its absolute value is the number of processes waiting for the semaphore. In the Sieve of Eratosthenes algorithm, a semaphore is shared by two processes. Hence, a counting semaphore can have three values: `-1`, 0 and 1. In contrast, a binary semaphore only has two values: 0 and 1. The difference between a counting semaphore and a binary semaphore indicates that our implementation would generate a larger state space than the one generated by SPIN. In order to make the comparison between our implementation and SPIN meaningful, we modified our implementation to use binary semaphores in this case since binary semaphores do not affect the behaviors of processes in the Sieve of Eratosthenes algorithm.

3. In the "middle1" process, we use statement `nextval1=nextval1` to force SPIN executing a transition when the condition

   ```
   (nextval1%myval1)!=0
   ```

   is not satisfied, because a transition is executed in our implementation when the condition does not hold. We also add a similar statement in the other "middle" processes. For the same reason, we use the con-

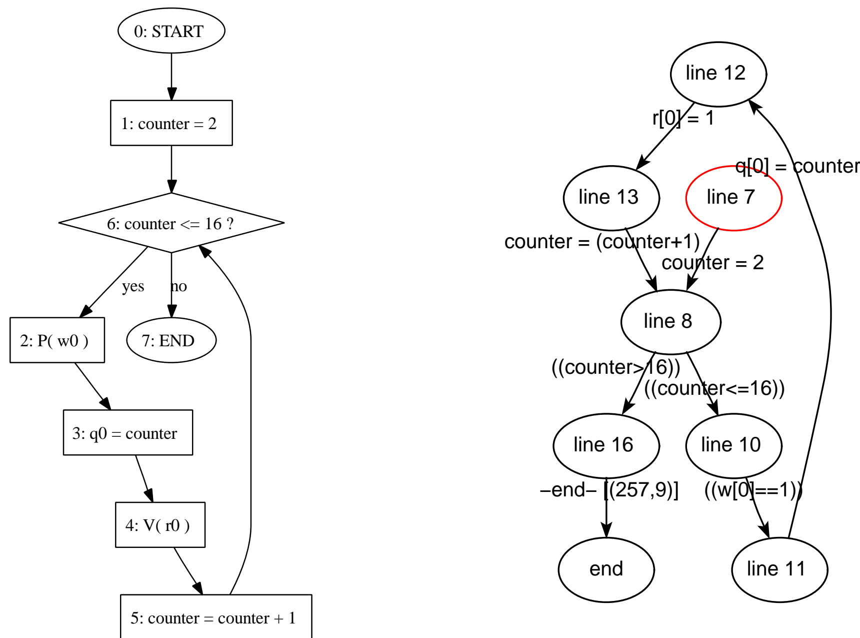

Figure 3.1: The flow chart and the automaton for the left process

dition `true` to force SPIN to generate a transition every time a loop begins in the "middle" processes and the "right" process.

It is easy to show that the above C program and the PROMELA program have equivalent transitions. Our implementation uses flow charts to perform state space searching. SPIN uses automata to perform the search. Figure 3.1 shows the flow chart (on the left) generated by our implementation and the automaton (on the right) generated by SPIN for the "left" process. In the flow chart, a transition is an *assignment* node or a branch of the *branch* node, and in the automaton, each edge, except the one from node "line 16" to node "end" (since it is labeled as "-end-"), is a transition. The circle in red is the starting node of the automaton. Indeed, each transition in the flow chart is matched to a transition in the automaton. For example, the transition from "6: counter<=16?" to "2: P(W0)" in the flow chart

is matched to the one from node "line 8" to node "line 10" in the automaton, and the transition from "6: counter<=16?" to "7: END" is matched to the one from "line 8" to node "line 16". The flow charts for the "middle" processes and the "right" process are matched to the automata in the same way.

Without annotation, a state searched by our implementation is composed of PC variables and program variables, including global and local variables. A state searched by SPIN contains not only PC variables and program variables, but also some overheads. But these overheads do not affect the number of states searched. Thus, full state space generated in our implementation (without annotation) has the same size as that generated by SPIN. The experimental results proved this.

We did an experiment for $N = 3$ and $P = 16$. At first, we ran the unannotated program by our implementation and SPIN. We removed the `else` clauses in the "middle" processes since these clauses are not necessary if we do not annotate the program and their introduction adds some more states. We use those `else` clauses in order to execute an annotation when a condition *does not hold*. For a sanity check, we have also run an unannotated version of the program with `else` clauses, which only change values of program counters. Then we ran the annotated program by our implementation with the different values of S.

The experiment was performed on a computer with one Pentium 4 2.8GHz CPU, 1 GB memory, Redhat Enterprise Linux 3. Our implementation and SPIN were compiled by GCC 3.2.3. The experimental results are summarized in Figures 3.2 and 3.3.

	Original			Modified		
	states	time	memory	states	time	memory
no else	307531	115	98.2	127604	44	40.9
with else	348286	127	109.4	147824	56	46.5
S = 0	13055	0.9	7.2	3352	0.1	2.6
1	70631	10	24.4	23088	1.9	9.0
2	182415	61	59.8	69968	9	24.5
3	290203	139	98.9	120184	34	42.0
4	340168	174	117.4	143888	55	50.7
5	348286	187	120.9	147824	56	52.2

Figure 3.2: Our implementation results

	Complete			Reduced		
	states	time	memory	states	time	memory
no else	127604	0.4	15.8	33639	0.6	6.2
with else	147824	2.9	17.9	35972	2.5	6.4

Figure 3.3: The SPIN results

The columns with the title “Original” show the number of states, running time (in second) and memory usage (in MB) when we use counting semaphores, and those with the title “Modified” show corresponding data using binary semaphores. The columns with the title “Complete” show the data generated by SPIN without the partial order reduction and those with the title “Reduced” show the data with the partial order reduction. For the unannotated program, our implementation (with binary semaphores) and SPIN generated the same number of states, which is what we expected, since we carefully constructed the equivalent PROMELA program. This gave us confidence about our implementation. Observe that the biggest reduction in the state space is when $S = 0$, i.e., only one candidate is allowed

to progress through the system. Note that the unannotated program with `else` clauses does produce more states than that produced by the version without `else` clauses, while it produces the same number of states as the worst case of annotation does.

It is interesting to observe that the SPIN system obtains a reduced state space automatically, by applying a built-in partial order reduction algorithm [HP94]. By applying our insight and using the annotation, we obtained a similar reduction without implementing the partial order algorithm. When we set S to 0 and 1, the number of searched states is smaller than the number searched by SPIN with partial order reduction. We do not suggest the use of annotations to replace automatic and effective state space reduction because, unlike the partial order reduction, the reduction using annotation does not guarantee that only equivalent behaviors are reduced. We are merely demonstrating the potential and flexibility of our annotation mechanism.

It can be shown that for some class of properties (specifically, those that preserve a partial order equivalence between executions, see e.g., [KP92]), there is no loss of information by checking only the annotated version. A proof of a similar program is discussed in [KP92]. However, such proofs are not always available, and thus we cannot always count on the fact that we will not lose some of the exhaustiveness of the verification. An important observation is that by using a parameter (denoted by S in this example), we controlled the exhaustiveness of the search. It can be proved that, in this specific example, the executions searched are equivalent to the ones that were explored [CGMP99]. In other cases we can lose exhaustiveness.

3.3 Summary

We have presented an approach that allows fine-tuning of the verification search. The approach is based on adding annotations to the verification code. In annotations, history variables and auxiliary variables are used to collect information. New language constructs, `commit`, `halt`, `report` and `annotate`, are employed to direct the search based on the information gathered. The annotations allow to avoid part of the verification or testing effort performed by a model checker. This does not cover some cases, and hence we compromise the exhaustiveness of the verification for the sake of practicality. Yet the annotations do not create false negative error traces. This helps to make affordable choices in the spectrum between the more comprehensive model checking and the more economic testing.

The experiment in the previous section shows that annotations can give a tester flexible control of the state space search. It is true that user expertise is needed to write annotations. A naive tester would prefer reduction techniques. But annotations allow an experienced tester to deal with a large system whose state space cannot be sufficiently reduced by automatic reduction techniques.

Although adding a history variable `check` did not add more states in the experiment, which has been demonstrated by the experimental results, it is important to note that the annotations, when not used carefully, may also increase the size of the state space. For example, consider the case that we use a history variable to keep the number of messages that have arrived. This encoding can result in multiple states having the same value for the

program variables, but different values for the history variable. There are cases where we may allow introducing history variables that would cause some state repetition, while, on the other hand, gaining a lot of reduction of the state space.

The method proposed has been implemented in C as a stand-alone model checker. Its input language is a subset of the C language with an extension discussed in Section 3.2.1. The complete BNF grammar in the input format of Lex/Yacc [LMB92] is shown in Appendix A.

Chapter 4

Enforcing execution of a partial order

We present in this chapter a program transformation that forces a program to execute according to a given scenario. The changes to the program code inspired by the transformation are minimal, allowing it to also have the other executions, when not started from a particular given initial state. The transformation preserves the concurrent structure of the program. Such a simple transformation can be verified or comprehensively tested in order to gain confidence in its correctness. Our transformation will be demonstrated for a given (Pascal-like) syntax, since it was implemented in PET, which uses Pascal as its input language. However, it is language-independent. We shall give a brief description of how to apply it to programs in other languages later. Thereafter, it can be used as a standard tool for testing the results of verification tools.

4.1 Transforming shared-variable programs

We assume a computational model of several concurrent processes with shared variables. Each process is coded in some sequential programming language such as *C* or Pascal. Although no explicit nondeterministic choice construct exists, an overall nondeterministic behavior of the program can be the result of the fact that the processes can operate at different relative speeds. Hence, even if we start the execution of the code with exactly the same initial state, we may encounter different behaviors. Our goal is then to enforce, under the given initial condition, that the program executes in accordance with the particular behavior.

In order to perform the transformation, we translate the code into a set of atomic actions (like what we did in Chapter 3). We keep pointers to the text location corresponding to the beginning and end of actions. In most cases, the transformation consists of adding code at these locations, i.e., before or after an action. For simplicity, we start presenting the transformation with the unrealistic assumption that we can add code for the existing actions in a way that maintains the atomicity of the actions. Since this will result in rather large actions, which cannot realistically be executed atomically, we split them in a way that is detailed and explained below.

4.1.1 Data structure of transformation

Let $A(p_i)$ be the set of actions belonging to process p_i and ρ a given execution of a sequence of actions. For each pair of processes p_i and p_j, $p_i \neq p_j$, such that for some occurrences α_k with $\alpha \in A(p_i)$, and β_l with $\beta \in A(p_j)$,

$\alpha_k \rightsquigarrow_\rho \beta_l$ (see Section 2.3), we define a variable V_{ij}, initialized to 0. It is used by process p_i to inform process p_j that it can progress. This is done in the style of the usual semaphore operations (it can be proved that a binary semaphore is sufficient here). Hence we say that α_k and β_l need to be *synchronized*. The process p_i does that by incrementing V_{ij} after executing α_k.

$$\text{Free}_{ij} : V_{ij} := V_{ij} + 1$$

The process p_j waits for the value of V_{ij} to be 1 and then decrements it.

$$\text{Wait}_{ji} : \textit{wait}\ V_{ij} > 0; V_{ij} := V_{ij} - 1$$

Let $S(p_i) \subseteq A(p_i)$ be the set of actions of process p_i that have an occurrence in ρ and are related by $\rightsquigarrow_\rho$ to an occurrence of an action in another process. Thus, $S(p_i)$ are the actions that have some (but not necessarily all) occurrences that need to be synchronized. Thus, we need to check whether we are currently executing an occurrence of an action $\alpha \in S(P_i)$ that requires synchronization. Let $count_i$ be a new local counter variable for process p_i. We increment $count_i$ *before* each time an action from $S(p_i)$ occurs, i.e., add the following code immediately before the code for α:

$$count_i := count_i + 1. \tag{4.1}$$

We define $\sharp_i \alpha_k$ to be the number of occurrences from $S(p_i)$ that appeared in ρ before α_k. We can easily calculate $\sharp_i \alpha_k$ according to the sequence ρ. This is also the value that the variable $count_i$ has during the execution of the code after the transformation, due to the increment statement in (4.1).

Suppose now $\alpha_k \rightsquigarrow_\rho \beta_l$, where $\alpha \in A(p_i)$, $\beta \in A(p_j)$, $p_i \neq p_j$. Then we add the following code *after* α_k:

$$\textit{if}\ count_i = \sharp_i \alpha_k\ \textit{then}\ \text{Free}_{ij} \tag{4.2}$$

We add the following code *before* β_l:

$$\textit{if count}_j = \sharp_j \beta_l \textit{ then } \mathrm{Wait}_{ji} \tag{4.3}$$

The notations $\sharp_i \alpha_k$ and $\sharp_j \beta_l$ should be replaced by the appropriate constants calculated from ρ. Since an action may participate in several occurrences on the same sequence, different code akin to (4.2) and (4.3) for multiple occurrences can be added. We can optimize the transformation by not counting (and not checking for the value of $count_i$ in) actions that appear only once in ρ. Similarly, we do not need to count actions that require the same added transformation code in all their occurrences.

Another consideration is to identify when the execution is finished. We can add to $S(p_i)$ the action α that appears last in the execution per each process p_i. Thus, we count the occurrences of α as well in $count_i$. Let $\sharp_i \alpha_k$ be the value of $count_i$ for this last occurrence α_k of α in ρ. We add the following code, after the code for α:

$$\textit{if count}_i = \sharp_i \alpha_k \textit{ then halt } p_i \tag{4.4}$$

(there is no *halt* statement in Pascal, so it could be implemented using a *goto*.) Again, if the last action of the process is the only occurrence of α, we do not need to count it. Note that, if we do not halt the execution of the process p_i here, we may encounter and perform a later action of p_i that is not in ρ and is dependent of an action of another process that did not reach its last occurrence in ρ. This may lead to a behavior quite different than the one we are investigating.

In order to minimize the effect of the additional code on executions other than the given execution (or those equivalent to it, under $\equiv_D$), we use

an additional flag $check_i$ (for each process p_i), whose value remains constant throughout the execution. This flag is *true* only when we run the code in the mode where we want to repeat the given behavior. Thus, in addition to the distinguished initial state for the execution, we also force $check_i = true$ for each process p_i. In all other cases, we set initially $check_i = false$. When $check_i = false$, even if we start the execution according to the initial state of the suspicious behavior, the program may follow an execution different than the suspicious one. Note that, we choose not to have one global variable *check*, since the different references of it by different processes would be interdependent and hence would defy our goal to preserve the concurrent structure of the execution.

Now, if *Code* is some code generated by our transformation, as described in (4.1)–(4.4), we wrap it with a check that we are currently tracing a given sequence, as follows:

$$\textit{if } check_i \textit{ then Code}$$

Some code simplification may be in place, for example, checking the value of $count_i$ and the value of $check_i$ can be combined to a single *if* statement.

4.1.2 Discussion

As stated above, modeling the additional code resulted from the transformation as amalgamated into the atomic actions of the original code is unrealistic. The behavior of the resulted code better corresponds to adding new actions. However, we have carefully constructed it so that it comprises additional actions that are mostly local to a process, i.e., independent of ac-

tions of other processes. The only additional dependent actions are of the form $Free_{ij}$ and $Wait_{ji}$. However, when such a pair is added, $Free_{ij}$ would be preceded by some action α_k, and $Wait_{ji}$ is succeeded by an action β_l such that $\alpha_k \rightarrow_\rho \beta_l$. The dependence of these actions ($Free_{ij}$ and $Wait_{ji}$) are the same as the existing ones (α_k and β_l). Moreover, it can be shown that there cannot be any occurrence of an action between α_k or $Free_{ij}$ (both in $A(p_i)$) that are dependent on actions from p_j. Similarly, all occurrences between the occurrence of $Wait_{ji}$ and β_l (both from p_j) are independent of actions from p_i. Hence the concurrency structure of the program is maintained and our construction does not generate new deadlocks, even when we break the actions of the transformed program in a more realistic way[1].

There is an issue which needs to be considered when we implement code transformation. In order to avoid introducing unnecessary delays or even deadlocks, we must guarantee that the implementation of Wait_{ji} must not block the process p_i. Specifically, if V_{ij} is 0 and process p_j is waiting for it to become 1, process p_i needs to be able to progress, which will allow it to eventually increment V_{ij}. Such blocking could exist, e.g., on a single processor multitasking the concurrent program, with $Wait_{ji}$ performing busy waiting for V_{ij} to become 1, the scheduler is unfair to process p_i. It is interesting that the sequence ρ_3 in Section 4.4 shows a situation in the Dekker's algorithm [BA90] that is related to such a case. (Hence, we will demonstrate, using our running example, a subtle concurrency problem in the Dekker's algorithm, which we need to avoid in the implementation of our transformation.)

[1]This issue can be formalized as action refinement [Vog93] under the partial order semantics.

4.1.3 An example

Consider Dekker's solution to the mutual exclusion algorithm in Figure 4.1. The flow charts appear in Figure 4.2. (Roughly speaking, a flow chart node is an atomic action.) Suppose we start the execution with $turn = 1$. The following execution ρ_1 can be obtained, where process $P1$ enters its critical section. Each line represents the occurrence of an action. It consists of a sequence number, the execution process, followed by the number of the flow chart node involved (in parentheses) according to Figure 4.2, and followed by the text corresponding to the action. An action corresponding to a condition is also followed by a 'yes' or a 'no', depending on whether the test succeeds or fails.

Here, both processes proceed to signal that they want to enter their critical sections, by setting $c1$ and $c2$ to 0 (lines 7 and 8), respectively. Because $turn = 1$ ($turn$ is checked in lines 11 and 12), process $P1$ has priority over process $P2$. This means that process $P2$ gives up its attempt, by setting $c2$ to 1 (line 13), while process $P1$ insists, waiting for $c2$ to become 1 (checked in line 14) and then enters its critical section (line 15).

```
1: (P1(0) : start)
2:   (P2(0) : start)
3: [P1(1) : c1:=1]
4:   [P2(1) : c2:=1]
5:   <P2(12) : true> yes
6: <P1(12) : true> yes
7: [P1(2) : c1:=0]
```

```
boolean c1, c2 ;
integer (1..2) turn;
```

```
P1::c1:=1;
  while true do
  begin
    c1:=0;
    while c2=0 do
      begin
        if turn=2 then
          begin
            c1:=1;
            while turn=2 do
             begin
              /* no-op */
             end;
            c1:=0
          end
      end;
    /* critical section 1 */
    c1:=1;
    turn:=2
  end
```

```
P2::c2:=1;
  while true do
  begin
    c2:=0;
    while c1=0 do
      begin
        if turn=1 then
          begin
            c2:=1;
            while turn=1 do
             begin
              /* no-op */
             end;
            c2:=0
          end
      end;
    /* critical section 2 */
    c2:=1;
    turn:=1
  end
```

Figure 4.1: Dekker's mutual exclusion solution

```
8:    [P2(2) : c2:=0]

9: <P1(8) : c2=0?> yes

10:   <P2(8) : c1=0?> yes

11:<P1(7) : turn=2?> no

12:   <P2(7) : turn=1?> yes

13:   [P2(3) : c2:=1]

14:<P1(8) : c2=0?> no

15:[P1(9) : /* critical-1 */]
```

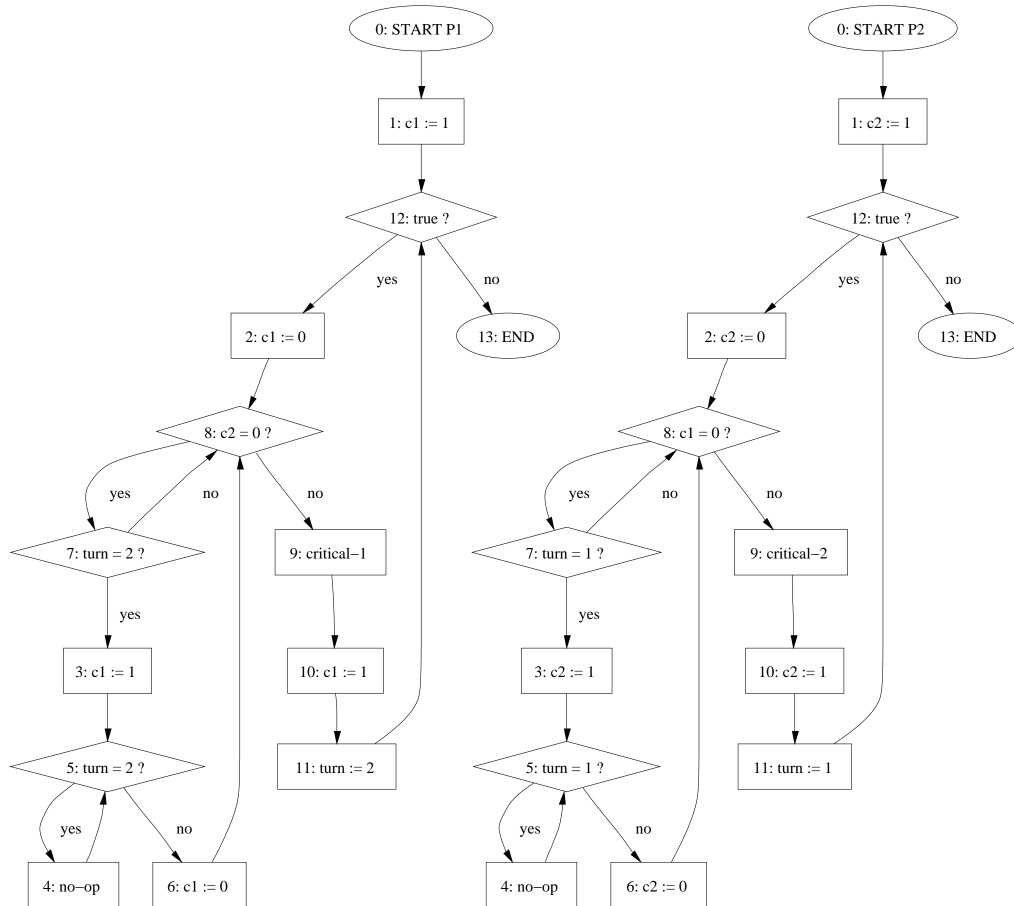

Figure 4.2: Dekker's mutual exclusion solution

A different execution ρ_2 can be obtained with the same initial state. Process $P2$ sets $c2$ to 0 (line 7), signaling that it wants to enter its critical section. It is faster than process $P1$, and manages also to check whether $c1$ is 0 (line 8) before $P1$ changes it from 1. Hence $P2$ enters its critical section (line 9).

```
1:(P1(0) : start)
2:  (P2(0) : start)
3:[P1(1) : c1:=1]
```

```
4:  [P2(1) : c2:=1]

5:  <P2(12) : true> yes

6:<P1(12) : true> yes

7:  [P2(2) : c2:=0]

8:  <P2(8) : c1=0?> no

9:  <P2(9) : /* critical-2 */>
```

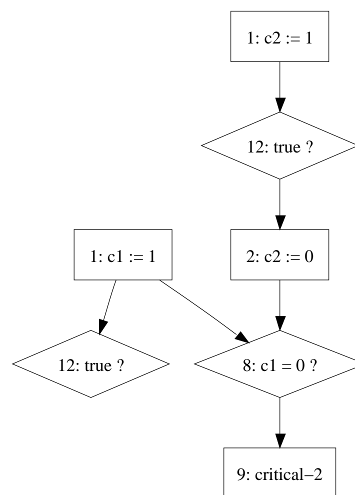

Figure 4.3: The order between occurrences in the execution ρ_2

Figure 4.3 consists of the graph of occurrences that correspond to the execution sequence ρ_2. The nodes in this figure are flow chart nodes corresponding to ρ_2 (note that the START nodes are not shown in the figure). The arrow from node 3 to 8 corresponds to the update and use of the same variable ($c1$) by the different processes (and according to $\rightsquigarrow_\rho$), while the rest of the arrows correspond to ordering between actions belonging to the same process.

The transformed Dekker's algorithm, which allows checking the path

ρ_2, is shown in Figure 4.4. The added code appears in boldface letters.

```
boolean c1, c2, check1, check2;
boolean V12 initially 0;
integer (1..2) turn;
```

```
P1::c1:=1;
  if check1 then V12:=1;
  while true do
  begin
    if check1 then halt P1;
    c1:=0;
    while c2=0 do
      begin
        if turn=2 then
          begin
            c1:=1;
            while turn=2 do
             begin
              /* no-op */
             end;
            c1:=0
          end
      end;
    /* critical section 1 */
    c1:=1;
    turn:=2
  end
```

```
P2::c2:=1;
  while true do
  begin
    c2:=0;
    if check2 then
      begin wait V12>0;
       V12:=0 end
    while c1=0 do
      begin
        if turn=1 then
          begin
            c2:=1;
            while turn=1 do
             begin
              /* no-op */
             end;
            c2:=0
          end
      end;
    /* critical section 2 */
    if check2 then halt P2;
    c2:=1;
    turn:=1
  end
```

Figure 4.4: The transformed Dekker's algorithm

4.2 Preserving the Checked Property

We define several operators on execution sequences, represented as sequences of actions (hence, in this case, ignoring the states):

$Hide_B(\rho)$ The sequence ρ after removing (projecting out) the actions from the set B.

$Cl_D(\rho)$ The set of sequences obtained from ρ by making repeated permutations between adjacent actions that are independent, i.e., not related by D. That is, $Cl_D(\rho) = \{\sigma | \sigma \equiv_D \rho\}$.

$Lin(E_\rho, \rightarrow_\rho^*)$ The set of linearizations (completions to total orders) of the partial order $\langle E_\rho, \rightarrow_\rho^* \rangle$. ($E_\rho$ is the set of occurrences of ρ.)

$Exec(P)$ The set of executions of a program P.

The operators defined here over sequences can be easily extended to sets of sequences, e.g.,

$$Hide_B(S) = \bigcup_{\rho \in S} Hide_B(\rho)$$

According to [Maz86], We have the following relation, connecting the above:

$$Cl_D(\rho) = Lin(E_\rho, \rightarrow_\rho^*) \tag{4.5}$$

That is, the trace-equivalent sequences obtained by shuffling independent events in ρ are the linearizations of the partial order view of ρ. This means that the partial order view and the trace equivalence are dual ways of looking at the same thing. This helps us to formalize the outcome of our program transformation.

In order to also take into account the preservation of the temporal properties, we present an additional view of an execution ρ. Let $\mathcal{P}$ be a set of propositions in some given temporal specification (e.g., as a linear temporal logic specification or using an automaton). Let $T : \mathcal{P} \mapsto \{true, false\}$ be a truth assignment over it. A *propositional sequence* is a (finite or infinite) sequence of truth assignments over some given set of propositions. Interpreting the propositional variables in each state of a sequence ρ (this time ignoring the actions) results in a *propositional sequence*.

In order to reason about the program transformations we propose, we will denote the original program actions by A and the augmented set of actions by $A' \supset A$ (some minor changes can be inflicted on the actions A, in particular, changes to program counter values, albeit there is no change in the code corresponding to these actions). We denote the dependency between the program actions A by the symmetric and reflexive relation $D \subseteq A \times A$. Adding new actions $A' \setminus A$ results in a new dependency relation $D' \subseteq A' \times A'$. We have that $D' \cap (A \times A) = D$, i.e., the program transformations do not add any dependencies between the original actions[2].

Let P be the original program, and P' be the result of the transformation. Then we obtain the following equation:

$$Hide_{A' \setminus A}(Exec(P')) = Cl_D(\rho) \tag{4.6}$$

That is, when hiding the additional actions from the executions of the transformed program, we can obtain any execution that is equivalent under $\equiv_D$ to the sequence ρ.

[2]This is guaranteed since we carefully construct the program transformation. See Section 4.1.2 for the detail.

Our transformation so far enables us to control the execution of the program in such a way that we are restricted to executions that are trace-equivalent to the counterexample. Suppose further that the execution ρ was obtained using a model checker, which was used to verify whether some concurrent program satisfies a property φ. We abstractly assume that the property φ corresponds to a set of propositional sequences. In practice it can be specified e.g., using a temporal formula or an automaton over (finite or infinite) sequences. Since ρ is a counterexample, it typically satisfies $\neg\varphi$.

Denoting $L(\varphi)$ as a set of propositional executions (or a sequence of actions, depending on the type of specification), the difficulty appears when φ is not closed under the trace equivalence [PWW98]. That is, we can have $\sigma \equiv_D \rho$ where $\sigma \in L(\varphi)$ and $\rho \in L(\neg\varphi)$. There are several solutions for this situation. One is to use a specification formalism that is closed under trace equivalence (see e.g., [APP95, KP90, TW97]). Another solution is to use a specification formalism that does not force trace closedness, and then apply an algorithm for checking whether φ is closed. Such an algorithm appears in [PWW98].

We propose here a third possibility, where we do not enforce φ to be trace-closed. The idea is to add dependencies so that the trace equivalence is refined, and equivalence sequences do not differ on satisfying φ. We construct a graph $G = \langle \rho, S, \Rightarrow \rangle$. Each node in S represents an execution sequence from $Cl_D(\rho)$. The *initial* node is $\rho \in S$. An edge $\sigma \Rightarrow \sigma'$ exists if σ' is obtained from σ using the switching of a single adjacent pair of independent actions. Starting the search from ρ, which satisfies φ, we check each successor node for the satisfaction of φ. Given that σ satisfies

φ but its successor σ' satisfies $\neg\varphi$, we add synchronization that prevents the permutation of σ into σ'. That is, if $\sigma = \mu\alpha_k\beta_l\mu'$ and $\sigma' = \mu\beta_l\alpha_k\mu'$ for some prefix μ and suffix μ', and occurrences β_l and α_k, we add a synchronization $\alpha_k \rightarrow_\rho \beta_l$. Note that, for optimization, we did not make the *actions* α and β interdependent, but rather synchronized two specific *occurrences*. We can reduce $\rightarrow_\rho$ using the adaptation of the Floyd-Warshall algorithm, as presented in Section 2.3.

This algorithm provides a small number of additional synchronizations for preserving the temporal property. However, its complexity is high. The size of $Cl_D(\rho)$ may be at worst exponential in the length of ρ. Other heuristic solutions are possible. For example, we can follow the partial order reduction strategies (ample sets, persistent sets or stubborn sets) and check which actions may change propositions participating in φ. Then we make all such actions interdependent. This solution is good for a specification that is stuttering closed. Details and further references can be found e.g., in Chapter 10 of [CGP99]. This solution has a much better complexity (quadratic in the number of actions), but is suboptimal since it may add some redundant synchronizations.

4.3 The Distributed Program Model

Consider now a different distributed systems model, where we have a handshake (synchronous) message passing instead of shared variables. Other models, such as buffered communication can be handled in a similar way, following the description in this and the previous section. We assume that

our program has the following kinds of actions:

- Local actions, related to a single process.
- Communication actions. We assume here a handshake communication, as in Ada or CSP. Such an action is executed jointly (and simultaneously) by a pair of processes.

We can again assign dependencies to the actions. We have that $(\alpha, \beta) \in D$ when α and β participate in a mutual process. Note that, in particular a communication participates in a pair of processes, hence it depends on actions from both processes.

We use the following syntactic construct:

$$\mathit{select}\ S_1 [] S_2 [] \ldots [] S_n\ \mathit{end}$$

where the code for S_i starts with a communication (*send* or *receive* a message), after a potential local condition, i.e, a 'guard'. One syntax for *guarded* communication [Hoa85] is of the form $en \Rightarrow co$, where en is a local condition, and co is a communication. In turn, co can be of the form $P!expr$, where P is a process and $expr$ is an expression whose calculated value is sent to P, or $Q?v$, where Q is a process, and v is a variable to receive the sent value. The joint effect of $P!expr$ on a process Q and $Q?v$ on process P is as if $v := expr$ was executed by the two processes, P and Q.

The *select* itself is not translated into an action. It is only a keyword that allows several communication actions to be potentially enabled at that location.

We add again a local counter $count_i$ for each process. The counter is incremented before each communication action inside a select. We can thus

check if the value of $count_i$ is $\sharp_i \alpha_k$ in order to select according to the given execution. We can then replace the *select* statement with a deterministic code that chooses the appropriate communication according to the suspicious execution. For example, consider the *select* statement

$$select\ \ \beta[]\gamma\ \ end$$

Suppose that β (together with a matching communication from another process) occurs in ρ as β_j and β_k and γ occurs as γ_l and γ_m. We replace the *select* statement with the following code:

case $count_i$ *of*

$\sharp_i \beta_j, \sharp_i \beta_k : \beta;$

$\sharp_i \gamma_l, \sharp_i \gamma_m : \gamma$

end

Thus, if $count_i$ is either $\sharp_i \beta_j$ or $\sharp_i \beta_k$ we need to choose the communication action β. In the other two cases, we need to choose γ. Note that, since a communication is shared between two processes, it would be counted separately by both.

As in the case of programs with shared variables, we need to add code for activating the additional checks only when enforcing a suspicious execution, i.e., when $check_i = true$. Similarly, we include the last action in every process in the counting, in order to halt the execution. Note that, with no shared variables and under handshake communication, the code added in the transformation is completely local to the processes. As the discussion for the shared variable case in Section 4.1.2, we can draw the conclusion that the transformation for programs with synchronized communications does not introduce new deadlocks.

Note that, if we want to enforce the preservation of the checked property on all executions that are trace equivalent to the suspicious one (the execution we want to monitor), we can apply the transformation given in Section 4.2. This means adding semaphores and, consequently, shared variables, even when the original code includes only interprocess communication.

4.4 Extending the Framework to Infinite Traces

A model checker may generate an infinite execution that fails to satisfy the given specification. Although infinite, such a sequence is *ultimately periodic* [Tho90]. It consists of a finite *prefix* σ and a finite recurrent sequence ρ. This is often denoted as $\sigma\rho^{\omega}$. We can apply our transformation with some small changes to the two finite parts, σ and ρ. Of course we cannot execute $\sigma\rho^{\omega}$, since it is infinite, but we can test its execution for any given finite length (depending on our patience). We use $\rightsquigarrow_{\sigma}$ to represent the partial order relation in σ, and use $\rightsquigarrow_{\rho}$ for ρ.

The first change is to adapt the counting of the actions involved in the synchronization, and in the last action of each process to behave differently according to σ and ρ. We add a variable $phase_i$ for each process p_i, initialized to 0. We do not halt the execution with σ. Instead, when we reach the last action of process p_i in σ (as described in Section 4.1), we update $phase_i$ to 1, and behave according to the execution ρ. However, if a process does not participate in the periodic sequence ρ, this process needs to be halted after its last action in σ. This is because otherwise it may progress to execute some

actions even without having their corresponding occurrences appearing in ρ. When we reach the end of σ, and each time we reach the last current process action according to ρ, we reset $count_i$ to zero.

Let $G(\rho) = \langle P, E \rangle$ be an undirected graph, whose nodes are the processes that have actions appearing in ρ, and an edge between p_i and p_j exists if there are occurrences α_k, β_l of ρ such that $\alpha_k \rightsquigarrow_\rho \beta_l$ or $\beta_l \rightsquigarrow_\rho \alpha_k$, $\alpha \in A(p_i)$, $\beta \in A(p_j)$.

There are two cases for the ultimately periodic part ρ that can be distinguished:

1. The graph $G(\rho)$ includes all the processes in one connected component (a maximal component of nodes such that there exists a path between each pair of nodes in the component). In this case, in the enforced execution, some occurrence of the ith iteration of ρ may be overtaken by the $i+1$st iteration of ρ, due to concurrency. However, such overtaking is limited, and events from the $i+2$nd iteration cannot overtake any event in the ith iteration.

2. The graph $G(\rho)$ consists of multiple disjoint connected components. In this case, the behavior is as if the components iterate independently, and there can be an unbounded overtaking between them[3]. A similar behavior is obtained when concatenating message sequence charts [MPS98].

If $G(\rho)$ does not contains all the processes in the system, there are some processes that do not have any actions participating in ρ. These processes

[3]This distinction is related to the definition of the concurrent star operator c^* [DR95] and its related infinite version c^ω. Although there may be unbounded overtaken between components, each component eventually iterate infinitely often in an infinite trace.

are either terminated (or disabled) before ρ starts, or not scheduled in ρ. In the latter case, it is possible that some processes were not given a fair chance to continue. Let P be a process that has an enabled action after ρ starts and is not scheduled in ρ. If we do not halt P when we transform the code, the transformed code is not guaranteed to behave according to $\sigma\,\rho^{\omega}$. This may result in additional actions from the underrepresented processes to be executed. Because of shared variables or message passing, this can affect the processes that are represented. Because our transformation inserts some code in which a process may wait for another based on some given execution, our transformation may result in a deadlock. Such a deadlock increases the possibility that the given execution is a false negative under fairness.

We provide here an example for an ultimately periodic sequence ρ_3, which indicates that a livelock occurs. Process $P2$ is occupied in an infinite loop, waiting for process $P1$ to relinquish its attempt to get into the critical section, while process $P2$ is making no progress. The finite prefix of ρ_3 is as follows:

```
1: (P1(0) : start)
2:   (P2(0) : start)
3: [P1(1) : c1:=1]
4:   [P2(1) : c2:=1]
5:   <P2(12) : true> yes
6: <P1(12) : true> yes
7: [P1(2) : c1:=0]
8:   [P2(2) : c2:=0]
```

```
9: <P1(8) : c2=0?> yes

10:  <P2(8) : c1=0?> yes

11:<P1(7) : turn=2?> no

12:  <P2(7) : turn=1?> yes

13:  [P2(3) : c2:=1]
```

The recurrent part of ρ_3 consists of the following two occurrences:

```
14:  <P2(5) : turn=1?> yes

15:  [P2(4) : /* no-op */]
```

The initial state is the same as in the previous example executions. In the ultimately periodic part, process $P1$ is not contributing to the execution, while $P2$ loops, waiting for *turn* to become 2 (lines 14 and 15). Since $P1$ does not execute, *turn* remains 1 and $P2$ never goes out of its loop. This execution can be the result of an analysis that does not take fairness into account. In a system which is implemented with fairness, process $P1$ would continue, and will check the value of $c2$, which now becomes 1; hence $P1$ can continue into its critical section, and eventually set *turn* to 2. Consequently, $P2$ will eventually be able to get into its critical section.

4.5 Summary

A method to transform programs has been proposed in this chapter in order to guarantee the execution of any total-order path satisfying a partial order in a concurrent environment. At first, we analyze the data dependency upon a given path and construct a partial order. We insert semaphore

operations into the code to maintain the dependency during the system execution. The execution then satisfies the partial order. This method was extended to transform programs with synchronized communications.

We have demonstrated the transformation on a PASCAL program. However, the method is not limited to PASCAL programs. For example, we have explained the similarity between PASCAL and C in Chapter 2. The transformation can be done for C programs with minor modifications to syntax. For programs which support nondeterministic behaviors, e.g., PROMELA programs, the transformation is done in the same way as for the distributed program model in Section 4.3.

The method of transforming programs was developed for discrete systems, i.e., untimed systems. If we apply it to real-time systems, it is possible that we do not obtain correct executions, because it takes time to execute the extra statements inserted into the code and thus time constraints in the system are changed. One way to enforce an execution which respects a specific partial order will be presented in Chapter 7. It is based on probabilistic behaviors of a real-time system.

The transformation method has been implemented in PET. A new button "transform" was added to the toolbar in the main window. After a user specifies a path, the transformed programs will be written into new files by clicking the "transform" button. The figures in the example section, e.g., Figure 4.1, 4.2, 4.3 and 4.4, were generated by PET. The path examples ρ_1, ρ_2 in Section 4.1.3 and ρ_3 in Section 4.4 were also generated using PET.

Chapter 5

Modeling real-time systems

This chapter describes how to model real-time systems and partial order executions. The code of a real-time system is modeled by flow charts. Each program is translated into a flow chart. A flow chart is translated into a transition system, which is then translated into an extended timed automaton. Finally, the product of automata is generated. We also model a partial order by an untimed automaton, which is synchronized with the product to generate a DAG (directed acyclic graph), which represents the partial order executions. The DAG will be used in the next chapter to calculate the timed precondition of a partial order. The procedure of generating a DAG is illustrated in Figure 5.1. Now we give the definition of models and the details of generating the DAG.

5.1 Transition systems

We describe *transition systems* (TS) over a finite set of processes $P_1 \ldots P_n$, each consisting of a finite number of transitions. The transitions involve

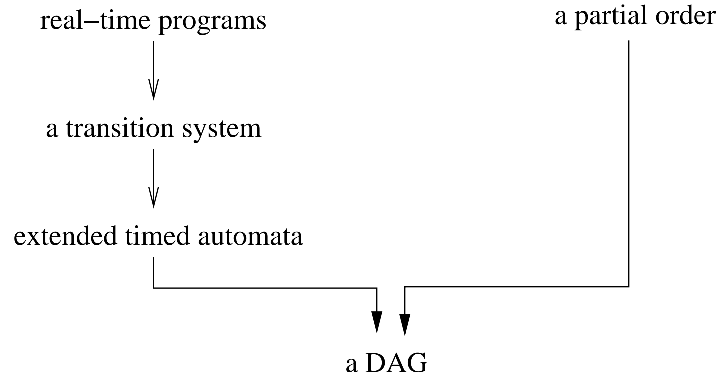

Figure 5.1: The framework of generating a DAG

checking and updating control variables and program variables (over the integers). Although the processes are not mentioned explicitly in the transitions, each process P_i has its own location counter $\hat{l}_i$. It is possible that a transition is jointly performed by two processes, e.g., a synchronous communication transition. We leave out the details for various modes of concurrency at this moment, and use a model that has only shared variables. We shall model joint transitions later in this chapter.

A transition includes (1) an enabling condition c (on the program variables), (2) an assertion over the current location of process P_j, of the form $\hat{l}_j = s$, (3) a transformation f of the program variables, and (4) a new value s' for the location of process P_j. For example, a test (e.g., *while* loop or *if* condition) from a control value s of process P_j to a control value s', can be executed when $(\hat{l}_j = s) \wedge c$, and result in the transformation f being performed on the program variables, and $\hat{l}_j = s'$. The transition is *enabled* if $(\hat{l}_j = s) \wedge c$ holds.

We equip each transition with two pairs of time constraints $[l, u]$, $[L, U]$

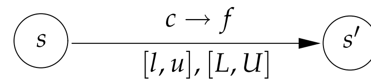

Figure 5.2: The edge

such that:

l is a lower bound on the time a transition needs to be *continuously* enabled until it is selected.

u is an upper bound on the time the transition can be *continuously* enabled without being selected.

L is a lower bound on the time it takes to perform the transformation of a transition, after it was selected.

U is the upper bound on the time it takes to perform the transformation of a transition, after it was selected.

We allow shared variables, but make the restriction that each transition may change or use at most a single shared variable.

Every process can be illustrated as a directed graph G. A location is represented by a node and a transition is represented by an edge. Figure 5.2 shows the graphic representation of a transition.

A flow chart can be translated into a transition system easily. The *begin* node and the *end* node need not be translated since they do not exist in programs. An *assignment* node is translated into a transition with the enabling condition *true* and the transformation is the assignment. A *branch* node with the predicate *pred* is translated into two transitions with null

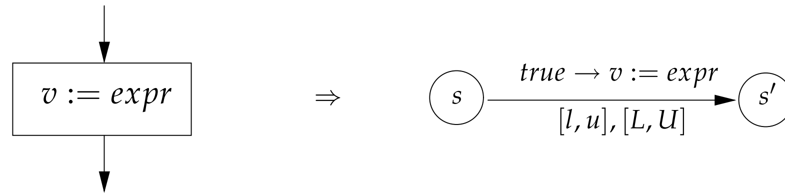

Figure 5.3: The translation of an *assignment* node

transformations. *pred* and $\neg pred$ are the enabling conditions of two transitions respectively, depending on whether the corresponding edge of the diamond is labeled *yes* or *no*. Figure 5.3 and 5.4 illustrate the translation of an *assignment* node and a *branch* node. Another kind of translation of the

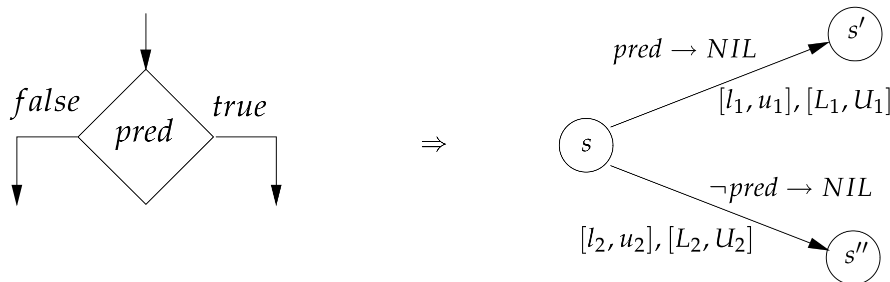

Figure 5.4: The translation of a *branch* node

branch node is shown in Figure 5.5. There are three transitions after translation. The first transition tests the predicate and stores the result in a new local boolean variable. The second one and the third one use the boolean variable and its negation form as their enabling conditions. Their transformations are null. The disadvantage of this way is that it introduces a new local variable. There might be some other ways to translate a *branch* node. But each way has its pros and cons. The timing parameters associated with each transition can be chosen according to the system specification.

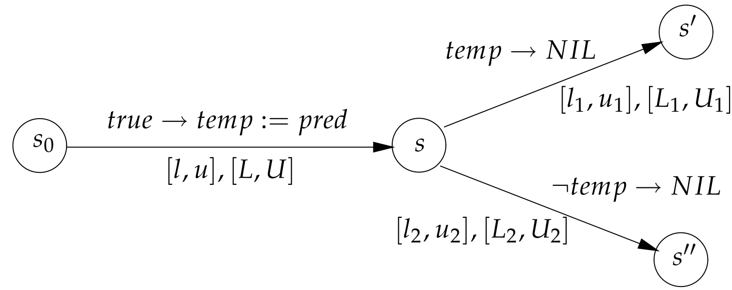

Figure 5.5: Another translation of a *branch* node

5.2 Extended timed automata

5.2.1 The definition

We model each process of the transition system as an *extended timed automaton* (ETA), which is a tuple $\langle V, X, Cl, B, F, S, S^0, \Sigma, E\rangle$ where

- V is a finite set of program variables.
- X is a finite set of assertions over a set of program variables V.
- Cl is a finite set of clocks,
- B is a set of Boolean combinations of assertions over clocks of the form $x\ rl\ const$, where x is a clock, rl is a relation from $\{<, >, \geq, \leq, =\}$ and $const$ is a constant (not necessarily a value, as our timed automaton can be parametrized).
- F is a set of transformations for the program variables. Each component of F can be represented, e.g., as a multiple assignment to some of the program variables in V.

- S is a finite set of states (locations). A state $s \in S$ is labeled with an assertion s^X from X and an assertion s^B on B that need to hold invariantly when we are at the state.
- $S^0 \subseteq S$ are the initial states,
- Σ is a finite set of labels for the edges.
- E is the set of edges, i.e., transitions, over $S \times 2^{Cl} \times \Sigma \times X \times B \times F \times S$. The first component of a transition e is the source state. The second component e^{Cl} is the set of clocks that are reset to 0 on this transition. A label e^{Σ} from Σ allows synchronizing transitions from different automata, when defining the product. We allow multiple labels on edges, as a terse way of denoting multiple transitions. An edge e also includes an assertion e^X over the program variables and an assertion e^B over the clocks which have to hold for the transition to fire (known as guards), a transformation e^F over the program variables and a target state.

The above definition extends timed automata [AD94] by allowing conditions over variables to be associated with the edges and states, and transformations on the variables on the edges (similar to the difference between finite state machines and extended finite state machines).

5.2.2 The execution

An *execution* of an ETA is a (finite or infinite) sequence of triples of the form $\langle s_i, \overline{V}_i, T_i \rangle$, where

1. s_i is a state from S,

2. $\overline{V}_i$ is an assignment for the state (program) variables V over some given domain(s), such that $\overline{V}_i \models s_i^X$ and

3. T_i is an assignment of (real) time values to the clocks in Cl such that $T_i \models s^B$.

   In addition, for each adjacent pair $\langle s_i, \overline{V}_i, T_i \rangle$ $\langle s_{i+1}, \overline{V}_{i+1}, T_{i+1} \rangle$ one of the following holds:

   A transition is fired. There is a transition e from source s_i to target s_{i+1}, where $T_i \models e^B$, $\overline{V}_i \models e^X$, T_{i+1} agrees with T_i except for the clocks in e^{Cl}, which are set to zero, and $\overline{V}_{i+1} = e^F(\overline{V}_i)$.

   Passage of time. $T_{i+1} = T_i + \delta$, i.e., each clock in Cl is incremented by some real value δ.

4. An infinite execution must have an infinite progress of time, i.e., the sum of all passage of time diverges.

An initialized execution must start with $s \in S^0$ and with all clocks set to zero. However, we deal with finite consecutive segment executions in this thesis, which do not have to be initialized.

5.2.3 The product

Let $ETA_1 = \langle V_1, X_1, Cl_1, B_1, F_1, S_1, S_1^0, \Sigma_1, E_1 \rangle$ and $ETA_2 = \langle V_2, X_2, Cl_2, B_2, F_2, S_2, S_2^0, \Sigma_2, E_2 \rangle$ be two ETAs. Assume the clock sets Cl_1 and Cl_2 are disjoint. Then the product, denoted $ETA_1 \parallel ETA_2$, is the ETA $\langle V_1 \cup V_2, X_1 \cup X_2, Cl_1 \cup$

$Cl_2, B_1 \cup B_2, F_1 \cup F_2, S_1 \times S_2, S_1^0 \times S_2^0, \Sigma_1 \cup \Sigma_2, E\rangle$. For a compound state $s = (s_1, s_2)$ where $s_1 \in S_1$ with $s_1^{X_1} \in X_1$ and $s_1^{B_1} \in B_1$ and $s_2 \in S_2$ with $s_2^{X_2} \in X_2$ and $s_2^{B_2} \in B_2$, $s^{X_1 \cup X_2} = s_1^{X_1} \wedge s_2^{X_2}$ and $s^{B_1 \cup B_2} = s_1^{B_1} \wedge s_2^{B_2}$. The transitions E are defined as follows. For every transition $e_1 = \langle s_1, e_1^{Cl_1}, e_1^{\Sigma_1}, e_1^{X_1}, e_1^{B_1}, e_1^{F_1}, s_1' \rangle$ in E_1 and $e_2 = \langle s_2, e_2^{Cl_2}, e_2^{\Sigma_2}, e_2^{X_2}, e_2^{B_2}, e_2^{F_2}, s_2' \rangle$ in E_2,

- joint transitions: if $e_1^{\Sigma_1} \cap e_2^{\Sigma_2} \neq \emptyset$, E contains
  $\langle (s_1, s_2), e_1^{Cl_1} \cup e_2^{Cl_2}, e_1^{\Sigma_1} \cup e_2^{\Sigma_2}, e_1^{X_1} \wedge e_2^{X_2}, e_1^{B_1} \wedge e_2^{B_2}, e_1^{F_1} \cup e_2^{F_2}, (s_1', s_2') \rangle$.
  Any variable is allowed to be assigned to a new value by either e_1 or e_2, not both.

- transitions only in ETA_1 or ETA_2: if $e_1^{\Sigma_1} \cap e_2^{\Sigma_2} = \emptyset$, E contains
  $\langle (s_1, s''), e_1^{Cl_1}, e_1^{\Sigma_1}, e_1^{X_1}, e_1^{B_1}, e_1^{F_1}, (s_1', s'') \rangle$ for every state $s'' \in S_2$ and
  $\langle (s', s_2), e_2^{Cl_2}, e_2^{\Sigma_2}, e_2^{X_2}, e_2^{B_2}, e_2^{F_2}, (s', s_2') \rangle$ for every state $s' \in S_1$.

Note that, when we model shared variables according to Section 5.4, we shall label edges as *capture_v*, *release_v*, *v_is_released*, *v_is_captured*, *v_is_unused*, *v_is_general* or *v_is_accessible* separately. If e_1 is labeled as *capture_v* or *release_v*, and e_2 is labeled as *v_is_released*, *v_is_captured*, *v_is_unused*, *v_is_general* or *v_is_accessible*, these two edges e_1 and e_2 are deemed as a joint transition when constructing the product.

5.3 Translating a TS into ETAs

We describe a construction of an extended timed automaton for a transition system. We first show how to construct states and edges for one particular location. An ETA is generated after all locations in a TS process are trans-

lated. Any location in a process is said to be the *neighborhood* of the transitions that must start at that location. The enabledness of each transition depends on the location counter, as well as an enabling condition over the variables. Location counters are translated in an implicit way such that each different location is translated into a different set of states. For a neighborhood with n transitions $t_1, \ldots, t_n$, let $c_1, \ldots, c_n$ be the enabling conditions of n transitions respectively. The combination of these conditions has the form of

$$C_1 \wedge \cdots \wedge C_n,$$

where C_i is c_i or $\neg c_i$. Each transition t_j in the neighborhood has its own local clock x_j. Different transitions may have the same local clocks if they do not participate in the same process or the same neighborhood.

1. We construct 2^n *enabledness* states, one for each Boolean combination of enabling conditional truth values. For any enabledness states s_i and s_k, there is an *internal* edge starting at s_i and pointing to s_k. Let $\mathcal{C}_i$ and $\mathcal{C}_k$ be the combinations for s_i and s_k, respectively. The edge is associated with $\mathcal{C}_k$ as the assertion over the variables. For any condition C_j which is $\neg c_j$ in $\mathcal{C}_i$ and c_j in $\mathcal{C}_k$, the clock x_j is reset ($x_j := 0$) upon the edge, for measuring the amount of time that the corresponding transition is enabled. We do not reset x_j in other cases. We add a self-loop to each state in order to generate the product of automata. The self-loop is labeled with the same combination as the state has, but it does not reset any clocks.

2. We also have an additional *intermediate* state per each transition in the

neighborhood, from which the transformation associated with the selected transition is performed. For any enabledness state s with the combination $\mathcal{C}$ in which the condition C_j corresponding to the transition t_j is c_j, let s'_j be the intermediate state for t_j and do the following:

(a) We have the conjunct $x_j < u_j$ as part of s^X, the assertion over the variable of s, disallowing t_j to be enabled in s more than its upper limit u_j.

(b) We add a *decision* edge with the assertion $x_j \geq l_j$ from s to s'_j, allowing the selection of t_j only after t_j has been enabled at least l_j time continuously since it became enabled. On the decision edge, we also reset the clock x_j to measure now the time it takes to execute the transformation.

(c) We put the assertion $x_j < U_j$ into s'_j, not allowing the transformation to be delayed more than U_j time.

(d) We add an additional *transformation* edge labeled with $x_j \geq L_j$ and the transformation of t_j from s'_j to any of the enabledness states representing the target location of t_j. Again, this is done according to the above construction. There can be multiple such states, for the successor neighborhood, and we need to reset the appropriate clocks. We add an assertion over variables to the transformation edge. The assertion is the combination of enabling conditions which is associated to the target state of the transformation edge.

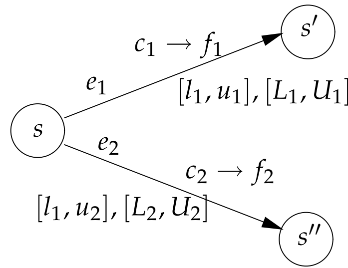

Figure 5.6: A neighborhood of two transitions

Figure 5.6 illustrates a neighborhood with two transitions e_1 and e_2 and Figure 5.7 provides the construction for this neighborhood. The states s_1, s_2, s_3 and s_4 are enabledness states, corresponding to the subset of conditions of e_1 and e_2 that hold in the current state at label s. The edges to s_5 correspond to e_1 being selected, and the edges to s_6 correspond to e_2 selected. The edges into s_5 also reset a local clock x_1 that counts the duration of the transformation f_1 of e_1, while the edges into s_6 zero the clock x_2 that counts the duration of f_2. The state s_5 (s_6, respectively) allows us to wait no longer than U_1 (U_2, resp.) before we perform f_1 (f_2). The edge from s_5 (s_6) to s_7 (s_8) allows delay of no less than L_1 (L_2) before completing f_1 (f_2). Note that s_7 (and s_8) actually represents a set of locations, in the pattern of s_1 to s_4, for the next process locations, according to the enabledness of actions in it (depending on the enabledness of the various transitions in the new neighborhood and including the corresponding reset of enabledness measuring clocks). Figure 5.9 demonstrates the translation of two sequential consecutive transitions in Figure 5.8 (note that self-loops are omitted).

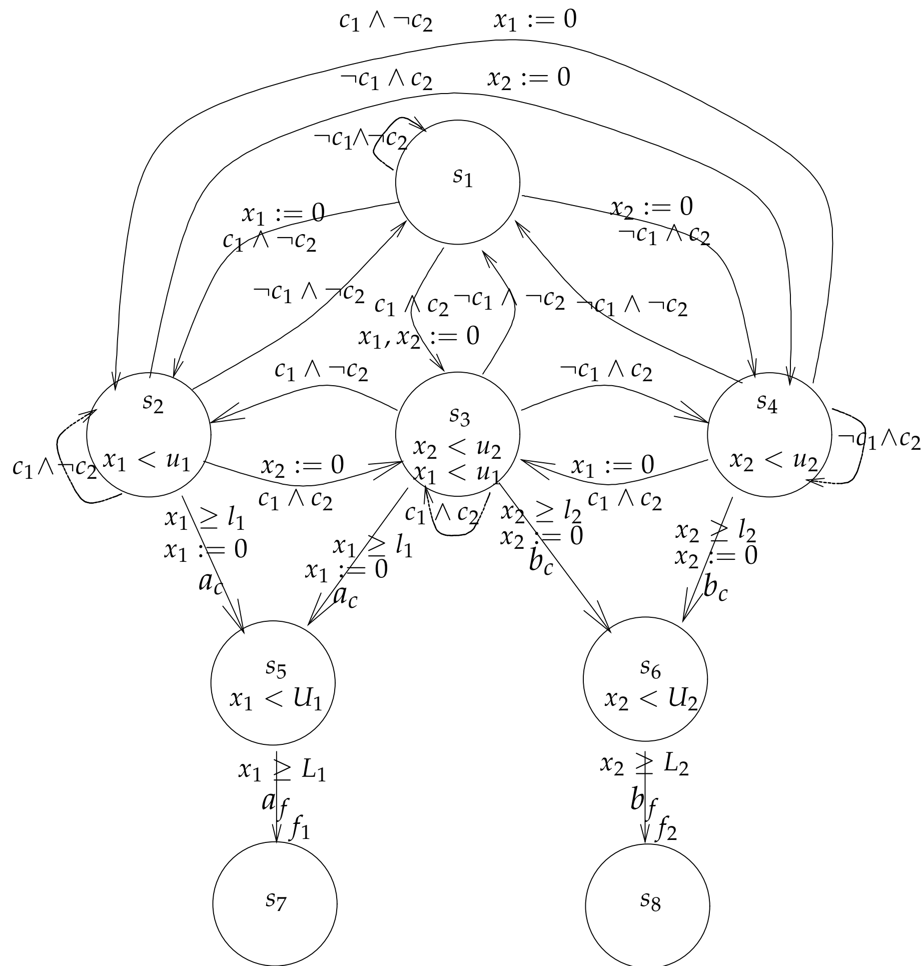

Figure 5.7: An extended timed automaton for a guarded transition

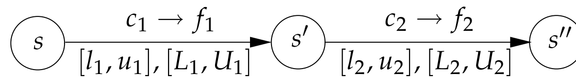

Figure 5.8: Two sequential transitions

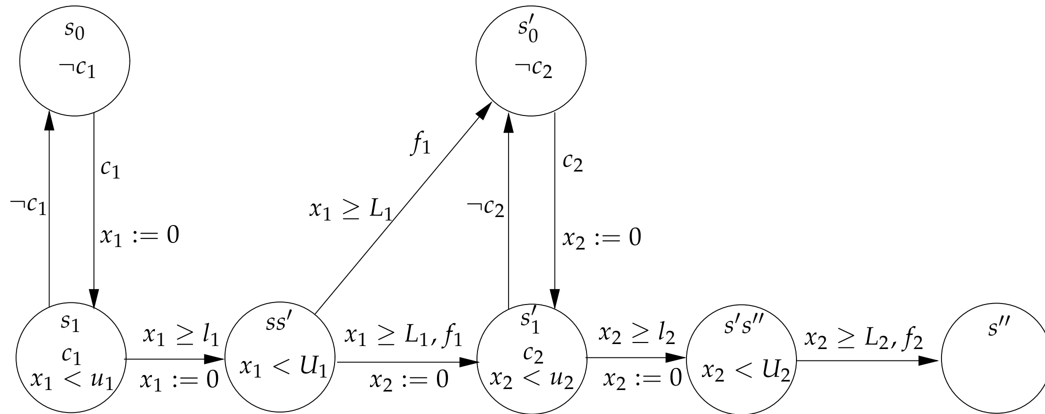

Figure 5.9: The translation of two sequential transitions

We also allow some labels to synchronize between processes, e.g., a shared communication. In this case, we need to label the different components, in the different processes, by the same label. The synchronization is done on both the decision edge and the transformation edge.

5.4 Modeling shared variables

In addition to the general description of translating TS into ETA, we present the procedure for modeling shared variables in a manner of mutual exclusion. A transition in a TS can reference a shared variable in either an enabling condition or transformation or both. We say that an enabling condition *reads* a shared variable if the enabling condition references the shared variable; a transformation *writes* a shared variable if the transformation references the shared variable. In fact, the transformation may not update the shared variable, and instead only use the value of the shared variable as the common meaning of the read operation. For example, v is a shared variable

in both assignments $y := v + 1$ and $v := y + 1$. The former assignment reads v and the latter writes v. But we do not distinguish these two cases in order not to make our model too complicated to be handled easily, because a very complicated model would divert our focus on calculating path precondition to modeling shared variables. Therefore, a shared variable can be read by an enabling condition and written by a transformation.

A shared variable needs to be protected by mutual exclusion when two or more transformations attempt to write to it concurrently. But we allow multiple concurrent read operations in order to provide maximum concurrency. For each shared variable v we provide a two state process, $\{used, unused\}$. We synchronize the decision edge of each transition that writes such a variable with an edge from *unused* to *used*, and each transformation edge of such a transition with an edge from *used* to *unused*. When a decision edge acquires v, all other processes reading v are forced to move to corresponding locations by synchronizing the decision edge with proper edges in those processes. For the same reason, a transformation releasing v is synchronized with relative edges to enable reading v.

When the two-state process is in *unused* state, we denote that v is unused. Similarly, v is used when it is in *used* state. We label each edge involving shared variables to synchronize them. For a decision edge changing shared variable v from *unused* to *used*, we label it as *capture_v*; for a transformation changing v from *used* to *unused*, we label it as *release_v*. Before we label edges among enabledness locations, we need to analyze their detailed behaviors. Let c be a condition containing a shared variable v. $\neg c$ means either c is evaluated to *false* or v cannot be accessed. Let e be an internal

edge connecting state s to state s'.

1. c is associated with s and $\neg c$ is associated with s'. Since a shared variable must be acquired before its value is changed by a transformation, here v must be captured by a decision edge and $\neg c$ means it cannot be accessed in this case. Hence e is labeled as *v_is_captured*.

2. $\neg c$ is associated with s and c is associated with s'. Based on the same reason as above, $\neg c$ here means it cannot be accessed and the change from $\neg c$ to c means a transformation releases v. Therefore e is labeled as *v_is_released*.

3. c is associated with both s and s'. v is not acquired in this case and thus e is labeled as *v_is_unused*.

4. $\neg c$ is associated with both s and s'. $\neg c$ means either the value of c is false or v is captured by another decision edge. These two situations cannot be distinguished in this case until the system is running. Then e is labeled as *v_is_general*.

The three tags *v_is_captured*, *v_is_released* and *v_is_unused* are mutually exclusive to one another. If an edge is labeled with two of them, this edge is removed from automata. For example, in such a situation that both conditions c_1 and c_2 contain the same shared variable v and an edge changes $\neg c_1 \wedge c_2$ to $c_1 \wedge c_2$, the edge is labeled with *v_is_released* and *v_is_unused*. But in fact, before $\neg c_1$ is changed to c_1, a decision edge captures v and then $\neg c_1 \wedge c_2$ is changed to $\neg c_1 \wedge \neg c_2$. Later a transformation releases v and both c_1 and c_2 are evaluated to true so that $\neg c_1 \wedge \neg c_2$ is changed to $c_1 \wedge c_2$. If an

edge is labeled as *v_is_general* and one of three tags above, the tag *v_is_general* is removed because we know the state of v in this case.

Another issue when we generate product of automata is to keep the correct access order for a shared variable. There are two cases that need to be considered:

1. A state s_i in process P_1 has an outgoing edge labeled as *capture_v* and another state w_j in P_2 has an outgoing edge labeled as *release_v*. The compound state $\langle s_i, w_j \rangle$ in the product does not have an outgoing edge labeled as *capture_v* because v has been captured by a previous edge. This ensures two processes cannot use a shared variable concurrently.

2. When a transformation in P_1 is executed, P_1 might have multiple target states to choose. (Remember in Figure 5.7, location s_7 and s_8 represent a set of states.) Thus there are multiple transformation edges, each pointing to a target state. Each transformation edge has an assertion on program variables. For example, assuming in a transition system, a transition e points to a node which has two neighbor transitions. The two neighbor transitions are translated according to Figure 5.7. Thus there are four transformation edges of e pointing to four enabledness nodes. These edges have assertions $c_1 \wedge c_2$, $\neg c_1 \wedge c_2$, $c_1 \wedge \neg c_2$, $\neg c_1 \wedge \neg c_2$ respectively. Some locations require that a shared variable v can be accessed, i.e., v is not used. We label edges pointing to these locations as a new tag *v_is_accessible*. In a formal definition, a transformation edge that has an assertion c which contains a shared variable v is labeled with a tag *v_is_accessible*. If a transformation edge

has an assertion $\neg c$ containing v, we do not label it because we do not know whether v is accessible or not.

During the generation of the product, edges labeled as *capture_v* or *release_v* are synchronized with edges labeled as *v_is_released*, *v_is_captured*, *v_is_unused*, *v_is_general* or *v_is_accessible* according to the following rules:

1. An edge labeled as *capture_v* cannot be synchronized with edges labeled as *v_is_released* or *v_is_unused*.

2. An edge labeled as *release_v* cannot be synchronized with edges labeled as *v_is_captured* or *v_is_unused*.

3. In a product state, an outgoing edge labeled as *v_is_accessible* or *capture_v* is removed if there is another outgoing edge labeled as *release_v*.

4. An edge involving two shared variables whose states are changed at the same time cannot be synchronized with any other edges. Thus this edge is removed from the product. For example, consider the condition $c_1 \wedge \neg c_2$ of the edge from s_4 to s_2 in Figure 5.7. Assume c_1 contains a shared variable v_1 and $\neg c_2$ contains a shared variable v_2. This edge is labeled with v_1*_is_released* and v_2*_is_captured*. In fact, this edge cannot be executed when we do not allow two transformation edges to execute synchronously.

We do not distinguish the case that a shared variable appears only in the transformation from the case that a shared variable appears in both the enabling condition and the transformation. The reason is to prevent the nonzenoness situation that the transition is enabled for the upper bound

of enabledness but the transformation cannot be executed because v is not released by another process. Thus we assume the enabling conditions in both cases read the shared variable. In fact, we can add $v = v$ to the enabling condition if v only appears in the transformation.

Since an internal edge which references a shared variable is synchronized with a decision edge or a transformation edge, its associated combination of conditions and reset clocks are merged to the decision or the transformation edge, and the internal edge is removed from the product. If an internal edge is not synchronized with any other edges, it is discarded from the product because it cannot be triggered. Therefore, the product does not contain any internal edges.

When we calculate a precondition backwards, we must record the status of each shared variable in order not to evaluate a condition which contains a non-accessible shared variable.

5.5 Modeling communication transitions

A pair of communication transitions usually includes one sending transition and one receiving transition. The sending transition sends a value to a named channel and the receiving transition receives the value from the same channel and assigns this value to a variable. An example is shown in Figure 5.10. $\hat{\alpha}!expr$ means sending the value $expr$ to the channel $\hat{\alpha}$ and $\hat{\alpha}?v$ means receiving the value $expr$ from $\hat{\alpha}$ and assigning it to the variable v.

The pair of communication transitions must be executed synchronously. No single transition is allowed to be triggered without executing

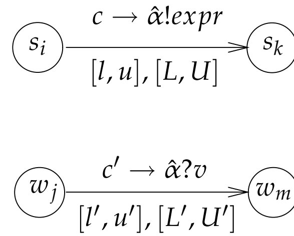

Figure 5.10: The pair of communication transitions

the other one. Therefore, their execution is *synchronized*. Only when the conjunction of both enabling conditions, for example, $c \wedge c'$, is satisfied, will their clocks start. The period during which the conjunction is satisfied is

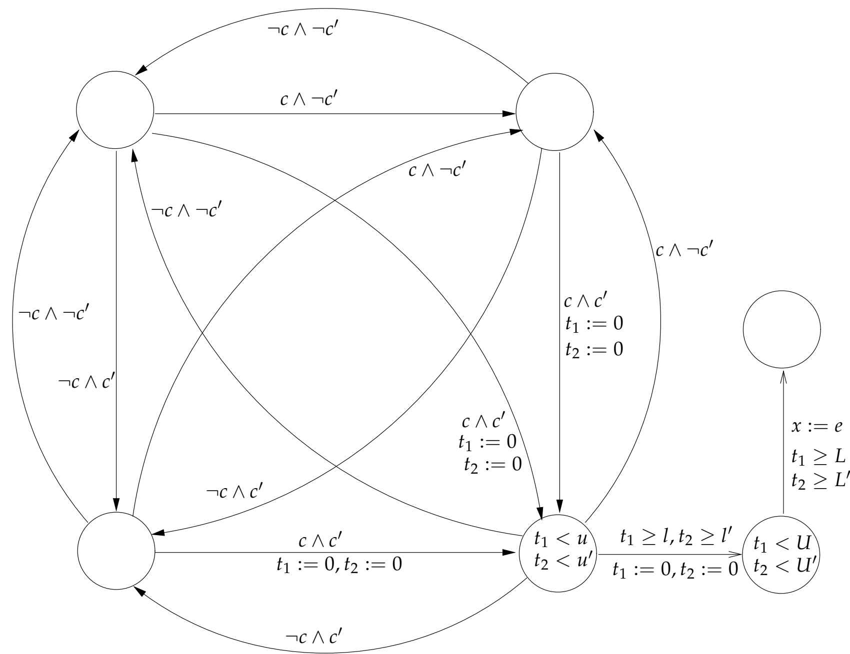

Figure 5.11: The product of communication transitions

bounded by $\max\{l, l'\}$ and $\min\{u, u'\}$. After the synchronized transition is selected, the execution is bounded by $\max\{L, L'\}$ and $\min\{U, U'\}$. The product of their ETAs can be produced as shown in Figure 5.11. In the states

where only one of the conditions c and c' is satisfied, we do not put an assertion $x_1 < u$ or $x_2 < u'$ into the them since the clocks x_1 and x_2 cannot begin to run. This is why, in Figure 5.11, there are three intermediate states that do not have any associated assertions.

Note that, in order to be consistent with the model of shared variables, at most one shared variable is allowed to appear in the joint enabling condition $c \wedge c'$, the expression $expr$ and the variable v.

5.6 Generation of the DAG

Given a selected sequence ρ of occurrences of program transitions, we calculate the *essential* partial order, i.e., a transitive, reflexive and asymmetric order between the execution of the transitions, as described below. This partial order is represented as formula φ over a finite set of *actions* $Act = A_c \cup A_f$, where the actions A_c represent the selections of transitions, i.e., waiting for their enabledness, and the actions A_f represent the transformations. Thus, a transition a is split into two components, $a_c \in A_c$ and $a_f \in A_f$. The essential order imposes sequencing of all the actions in the same process, and pairs of actions that use or set a shared variable. In the latter case, the enabledness part b_c of the latter transition succeeds the transformation part a_f of the earlier transition. However, other transitions can interleave in various ways (e.g., $d_c \prec e_c \prec e_f \prec d_f$). This order relation $\prec$ corresponds to a partial order (which is defined in Section 2.3) over Act. The formula φ is satisfied by all the sequences that satisfy the constraints in $\prec$, i.e., the linearizations (complementation to total orders) over Act. In

particular, ρ is one (but not necessarily the only) sequence satisfying the constraints in φ. Let $\mathcal{A}_\varphi$ be an automaton that recognizes the untimed language of words satisfying φ.

The representation takes into account the time constraints. We take the product of the extended timed automata for the different processes and label each transition in the product with respect to $A_c \cup A_f$. For example in Figure 5.7, the edges $s_2 \rightarrow s_5$ and $s_3 \rightarrow s_5$ can be labeled with a_c, the edges $s_3 \rightarrow s_6$ and $s_4 \rightarrow s_6$ can be labeled with b_c. The edge $s_5 \rightarrow s_7$ can be marked by a_f and s_6 to s_8 by b_f. Then we synchronize the product with $\mathcal{A}_\varphi$. The synchronization is done in a standard way [AD94]. Note that there is often a nondeterministic choice for taking such labeled transitions. This choice increases the branching degree on top of the branching already allowed by $\mathcal{A}_\varphi$. We obtain a DAG after synchronization.

5.7 An example

We use this example to illustrate the whole process of obtaining a DAG from a transition system and a given partial order. The system consists of two concurrent processes, which are created from programs P1 and P2 in Figure 5.12. The variable v1 is a shared variable and v2 is a local variable.

The flow chart representing program P1 is shown on the left of Figure 5.13 and the flow chart for program P2 is on the right. These flow charts are generated according to the rules in Section 2.1.1. The if statement in program P1 is translated into a *branch* node with two *assignment* nodes and the assignment statement in P2 into an *assignment* node.

```
Program P1                      Program P2
begin                           begin
  if v1=0 then                    v1 :=1
    v2 := 1                     end.
  else
    v2 := 0
end.
```

Figure 5.12: An example system

According to the rules in Section 5.1, program P1 is translated into a process of a transition system, which is on the left of Figure 5.14, and program P2 is translated into the process on the right of this figure. Here the enabling condition and the transformation of a transition are displayed by two edges respectively, which are connected by a node whose name has a suffix "-tran". For example, transitions corresponding to the *branch* node of program P1 in Figure 5.13 are represented by edges from node P1_3 to node P1_1 via node P1_3_P1_1-tran and from P1_3 to node P1_2 via node P1_3_P1_2-tran, respectively. In this way we can easily distinguish each action used to specify a partial order (see Section 5.6). Time bounds in this transition system are chosen as follows:

- An enabling condition *true* has a lower bound 0 and an upper bound 1.
- Other enabling conditions and all transformations have a lower bound 5 and an upper bound 10.

The ETA translated from the process for program P1 is illustrated on the left of Figure 5.15, and the ETA for program P2 on the right. The states in the shape of an ellipse are initial states. The states P1_3-0, P1_3-1 and

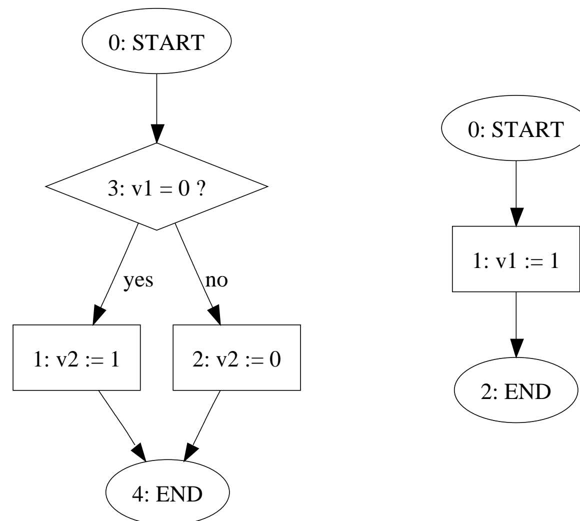

Figure 5.13: Flow charts for Program P1 (left) and P2 (right)

P1_3- 2 are the enabledness states with respect to the node P1_3 of program P1 in Figure 5.14. In brief, the enabledness state(s) of a node in the transition system in Figure 5.14 is(are) the state(s) whose name has a prefix which is the name of the node. The expression below the state name in a state is the assertion over program variables of the state and the other expression in the state is the assertion over clocks (note that P1d_t1, P1d_t2 and P2d_t1 are clocks.). If an assertion over clocks is true, it is omitted. There are three enabledness states[1] for node P1_3. The combination of conditions v1 = 0 and (not v1 = 0) cannot be satisfied so that the corresponding state is removed from the ETA. The state with the assertion (not v1 = 0 and (not (not v1 = 0)) represents that the shared variable v1 cannot be accessed. Note that ~1 is used to express -1 in SML and the condition not v1 = 0 is simplified into 1 <= v1 or v1 <= ~1 by the Omega library. The states whose name end with

[1]There are four enabledness states for a neighbourhood with two transitions according to Section 5.3.

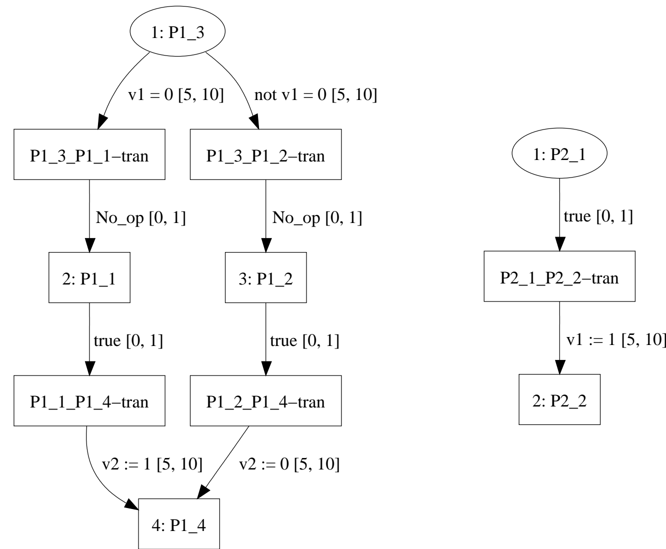

Figure 5.14: Processes for programs P1 (left) and P2 (right)

"-tran" are intermediate states. The label of an edge is the assertion and set of clocks being reset upon the edge. The label for a self loop is not shown in order to give a succinct illustration.

The product of ETAs in Figure 5.15 is shown in Figure 5.16. It is constructed according to the rules in Section 5.2.3. Each state in the product is a compound state, which includes one state from the ETA of program P1 and one from the ETA of P2. In order to give a readable graph, only states are displayed. Other elements, such as assertions, can be found from separate ETAs. The ellipse states are initial states. Each initial state represents a different initial condition, which is the conjunction of assertions associated to the initial states of P1 and P2. The initial conditions for states {P2_1-1, P1_3-1}, {P2_1-1, P1_3-2} are {not v1 = 0} and {v1 = 0}, respectively. The state {P2_1-1, P1_3-0} is removed from the figure since the initial condition which represents that v1 is not accessible is not satisfiable.

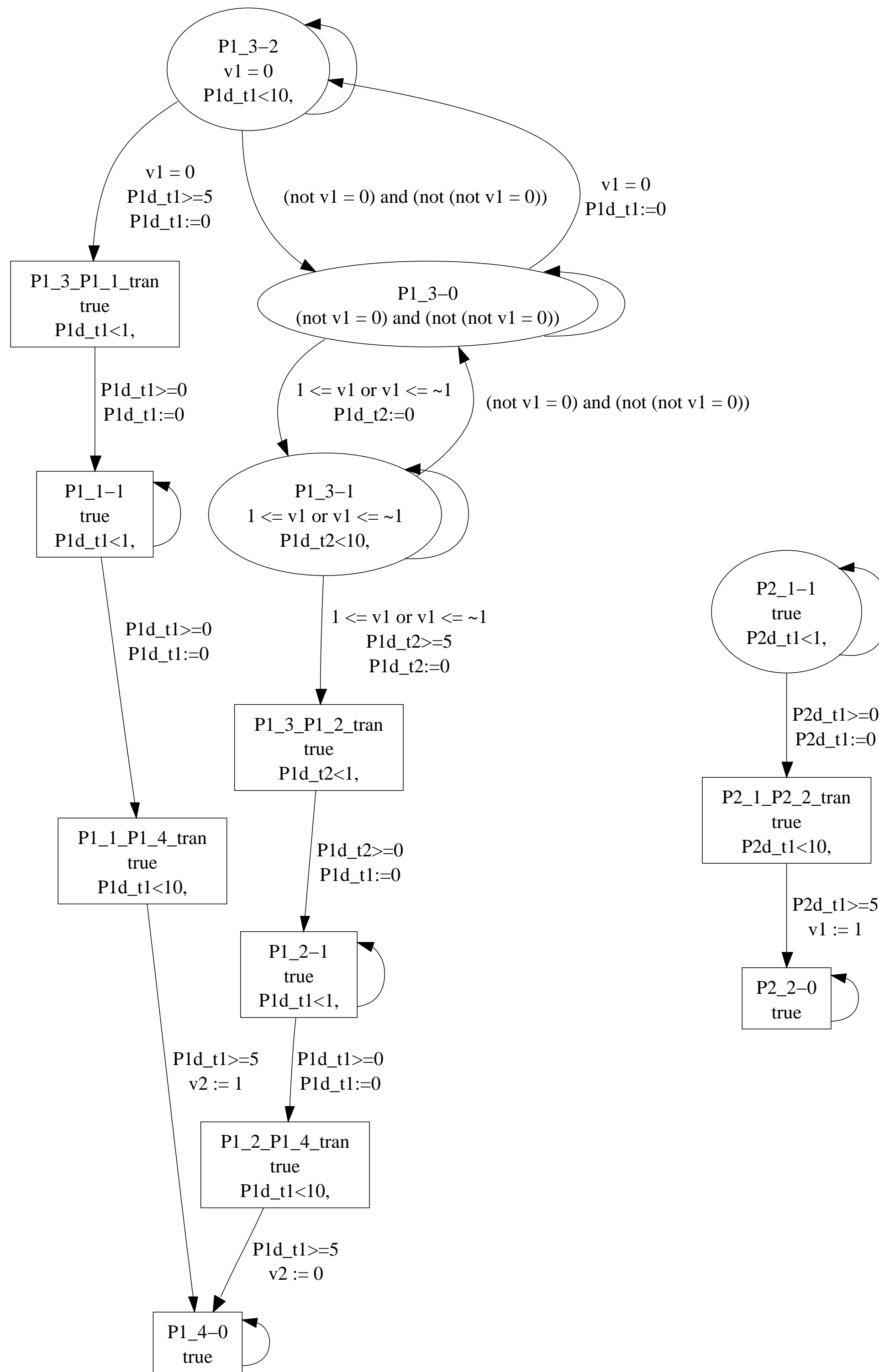

Figure 5.15: ETAs for programs P1 (left) and P2 (right)

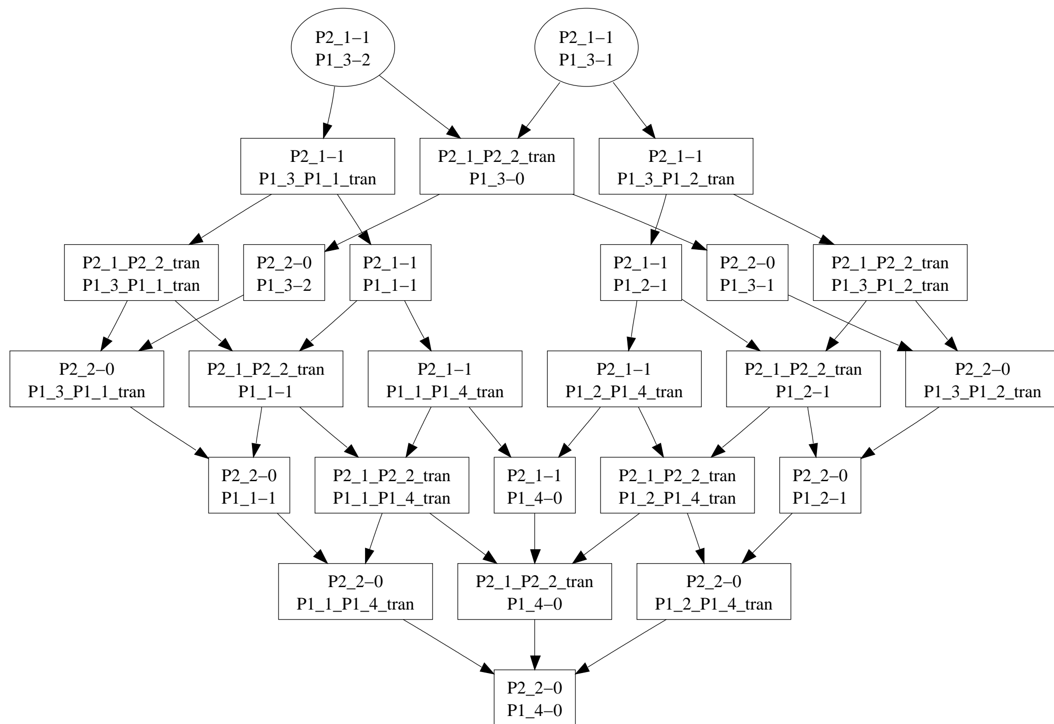

Figure 5.16: The product

The left part of Figure 5.17 shows a given partial order represented by flow chart nodes. The partial order in flow chart nodes can express the partial ordering relations intuitively. However, in order to generate a DAG, the partial order in flow chart nodes must be translated into one represented by actions, which is shown on the right of Figure 5.17. The node {P1_3 => P1_3_P1_1-tran} represents the enabledness condition of the transition from P1_3 to P1_1 in Figur 5.14. Other nodes have a similar meaning. On the left part of the figure, the edge from *branch* node {3: v1 = 0?} to *assignment* node {1: v1 = 1} requires that the former must be executed earlier than the latter since they reference the same shared variable v1. In fact, we can see from the right part of the figure that only the enabledness condition action

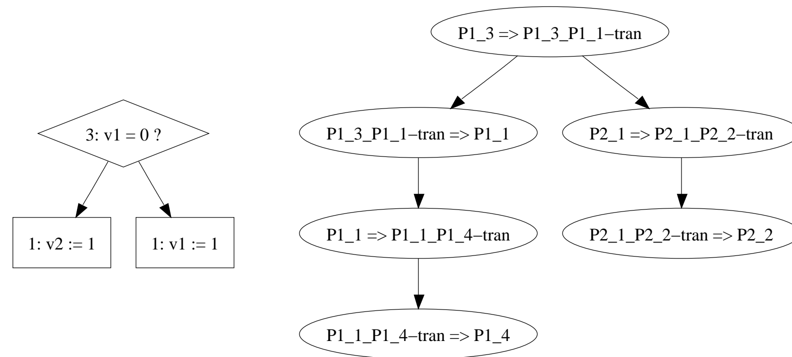

Figure 5.17: Partial order 1

of the *branch* node is guaranteed to be fired earlier than the enabledness condition action of the *assignment* node because the transformation action of the *branch* node does not access v1.

Figure 5.18 shows the DAG constructed from the partial order 1 in Figure 5.17 and the product in Figure 5.16 according to Section 5.6. Similarly to Figure 5.16, only node names are displayed. The node in the ellipse shape is the initial node. Note that there is only one initial node in this case, but there may be many in other cases.

Figure 5.19 shows another given partial order in both flow chart nodes (left) and actions (middle), and its corresponding DAG (right). This partial order required that the transition in program P2 is triggered earlier than transitions in P1.

5.8 Summary

In this chapter, we presented a transition system model and an extended timed automaton model. A real-time system is modeled by a transition

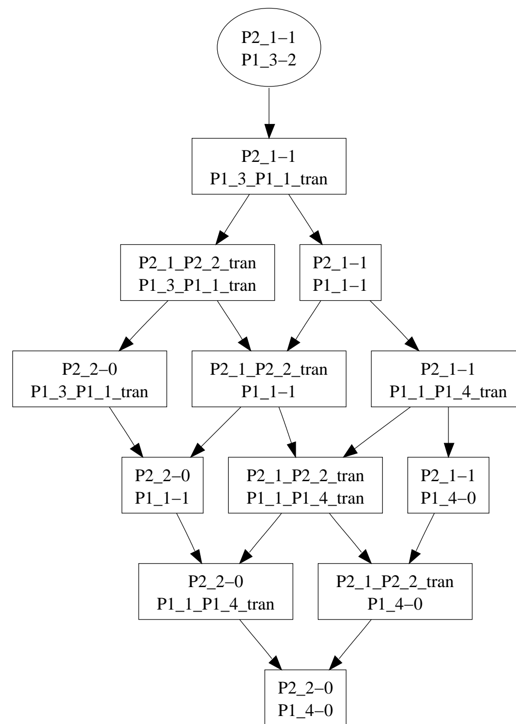

Figure 5.18: The DAG for the partial order 1

system first. By translating each node and its neighbor transitions in the transition system, a set of extended timed automata (ETAs) are constructed, each of which models a process. In a partial order provided by a user, a directed edge signifies that one ETA transition must be fired earlier than another. A DAG is generated by synchronizing the product automaton of the set of extended timed automata with the partial order. This DAG will be used in the next chapter and in Chapter 7 to calculate the precondition and the probability of the provided partial order, respectively.

When we translate a transition system into extended timed automata, each location is translated into 2^n states if it has n neighbor transitions. In theory, the total number of states can increase exponentially compared to

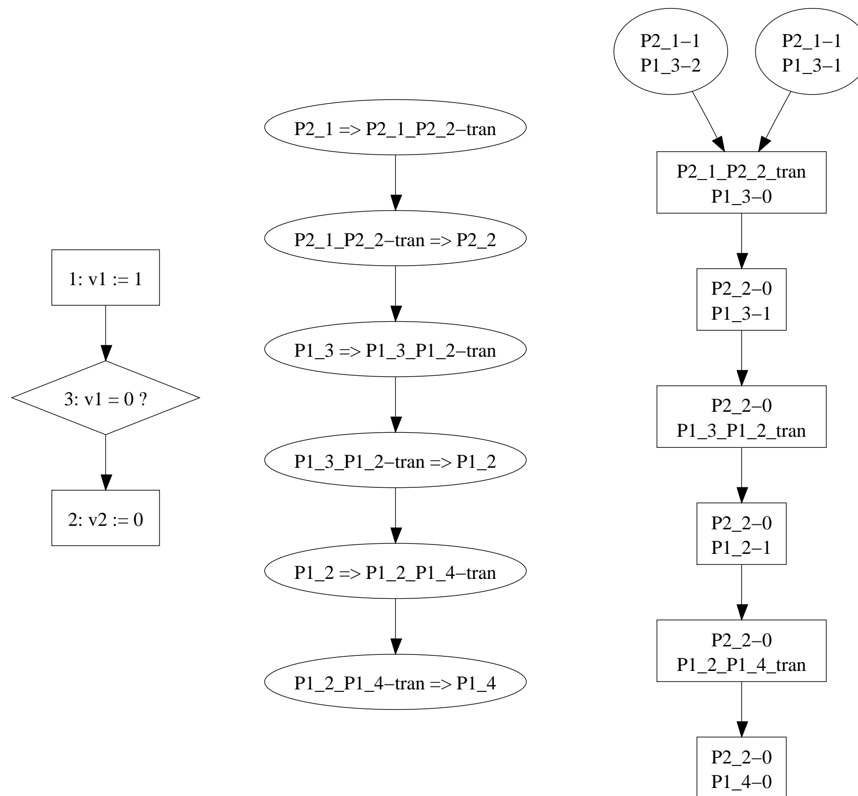

Figure 5.19: Partial order 2 and the corresponding DAG

the number of locations. In practice, however, the number of states is limited by program structures. For example, the if statement has two branches: one satisfies the condition and the other satisfies the negation of the condition. After the branches are translated into transitions, there are less than four combinations of enabling conditions, since the two enabling conditions cannot both be satisfied. A timeout scenario [HMP94] often involves two neighbor transitions, one of which has the enabling condition *true* and thus there are two combinations in this case.

The complete procedure of generating a DAG, including the TS model, the ETA model, the translation from a TS to ETAs, the generation of the

product and the partial order, the synchronization of the product automaton and the partial order, were implemented in RPET, the real-time extension to PET. The figures in the example section, e.g., Figure 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18 and 5.19, were generated by RPET.

Chapter 6

Calculating the precondition of a partial order

This chapter provides the methodology for calculating the required precondition of a partial order. The intersection of the product automaton with the partial order gives us a finite DAG. We can now compute the precondition for that DAG from the leaves backwards through time analysis based on time zone calculation. The precondition uses the usual *weakest* precondition for the program variables, and a similar update for the time variables, involving the local clocks and the time constraints. When a node has several successors, we combine the conditions obtained on the different edges into a disjunction. The precondition calculation can be used in a model checking search, hunting for a path satisfying a given temporal property, in the automatic generation of test cases for concurrent real-time systems, or in the synthesis of real-time systems.

6.1 Calculating untimed path condition in timed systems

We begin with an introduction on calculating the weakest path conditions on program variables in timed systems. We first translate the flow chart nodes in a path into transitions according to the rules in Section 5.1. But we do not consider the timing information in this section. We obtain a graph with nodes representing locations and edges representing transitions. For example, when we translate the path at the left of Figure 6.1, we obtain the graph on the right of that figure. To avoid confusion, we use *node* for the flow chart nodes, and use *point* for the nodes in the translated graph.

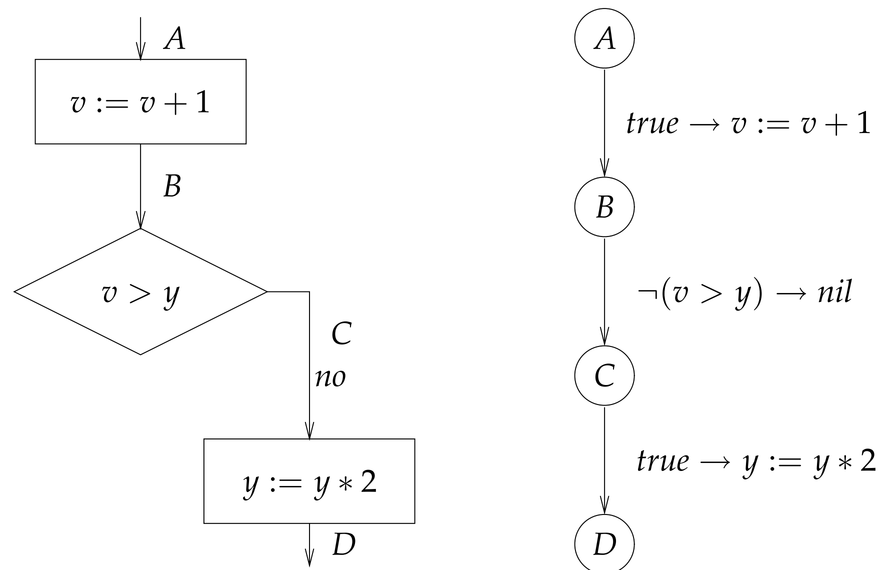

Figure 6.1: A path

The calculation of the untimed precondition of a path follows the algorithm in Section 2.4. An *accumulated path condition* represents the condi-

tion to move from the current point in the calculation to the *end of the path*. The current point moves backwards at each step in the calculation of the path condition, over one edge (node) to the previous point. We start with the condition *true*, at the end of the path (i.e., after the last node). Going backwards from a given location (point) over an edge marked with a transition with condition c and transformation f and into another point, we perform the following transformations to the accumulated path condition φ to obtain new accumulated path condition φ^R:

- We "relativize" the condition φ with respect to the assignment representing the transformation; if the assignment is of the form $v := expr$, where v is a variable and $expr$ is an expression, we substitute $expr$ instead of each free occurrence of v in the path condition. This is denoted by $\varphi[expr/v]$.

- Then we conjoin the transformation condition c. We simplify the new accumulated path condition obtained using various first order logic equivalences.

Thus, φ^R is defined as follows:

$$\varphi^R = \varphi[expr/v] \wedge c. \tag{6.1}$$

Calculating the path condition for the example in Figure 6.1 backwards, we start at the end of the path, i.e., point **D**, with a path condition *true*. Moving backwards through the assignment $y := y * 2$ to point **C**, we substitute every occurrence of y with $y * 2$. However, there are no such occurrences in *true*, so the accumulated path condition remains *true*. Conjoining *true* with the transition condition *true* maintains *true*. Progressing

backwards to node **B**, we now conjoin the accumulated path condition with $\neg(v > y)$, obtaining (after simplification, which gets rid of the conjunct *true*) $\neg(v > y)$. This is now the condition to execute the path from **B** to **D**. Passing further back to point **A**, we have to relativize the accumulated path condition $\neg(v > y)$ with respect to the assignment $v := v + 1$, which means replacing the occurrence of v with $v + 1$, obtaining $\neg(v + 1 > y)$. Again, conjoining that with *true* does not change the path condition.

6.2 Untimed precondition of a DAG

For a DAG of nodes G, each edge is marked by a transition from a concurrent program. We are initially not concerned with time constraints. Each node corresponds to a location in a sequential program, or a combination of locations in a concurrent program. There are some nodes that have no predecessors, distinguished as *initial* nodes. The nodes that have no successors are *leaves*.

The condition to perform at least one path from an initial node to a leaf node in the DAG can be calculated as follows:

1. Mark all the nodes of the DAG as *new*.
2. Attach the assertion *true* for each leaf node, and mark them as *old*.
3. While there are nodes marked with *new* do
   (a) Pick up a node $\overline{z}$ that is marked *new* such that all its successors $Z = \{z_1, \ldots, z_k\}$ are marked *old*.
   (b) Relativize each assertion φ_i on a node $z_i \in Z$ to form φ_i^R.

(c) Attach $\varphi_1^R \vee \ldots \vee \varphi_k^R$ to node $\overline{z}$. Mark $\overline{z}$ as *old*.

6.3 Date structure to represent time constraints

Time constraints are a set of relations among local clocks. We also use a global clock to count system execution time from its initial state to its last state which, unlike local clocks, is not reset during the execution. Time constraints can be obtained from reachability analysis of clock zones. A Difference-Bound Matrix (DBM) [Dil89] is a data structure for representing clock zones.

6.3.1 The definition

A DBM is a $(m+2) \times (m+2)$ matrix where m is the number of local clocks of all processes. Each element $D_{i,j}$ of a DBM D is an upper bound of the difference of two clocks x_i and x_j, i.e., $x_i - x_j \leq D_{i,j}$. We use x_1 to represent the global clock and $x_2, \cdots, x_{m+1}$ to represent local clocks. x_0 is a special clock whose value is always 0. Therefore, $D_{i,0}$ $(i > 0)$, the upper bound of $x_i - x_0$, is the upper bound of clock x_i; $D_{0,j}$ $(j > 0)$, the upper bound of $x_0 - x_j$, is the negative form of the lower bound of clock x_j. To distinguish non-strict inequality $\leq$ with strict inequality $<$, each element $D_{i,j}$ has the form of (r, F) where $r \in \mathbb{R} \cup \{\infty\}$ and $F \in \{\leq, <\}$ with an exception that F cannot be $\leq$ when r is ∞. Addition $+$ over $F, F' \in \{\leq, <\}$ is defined as follows:

$$F + F' = \begin{cases} F & F = F' \\ < & F \neq F' \end{cases}$$

Now we define addition $+$ and comparison $<$ for two elements (r_1, F_1) and (r_2, F_2).

$$(r_1, F_1) + (r_2, F_2) = (r_1 + r_2, F_1 + F_2).$$

$$(r_1, F_1) < (r_2, F_2) \text{ iff } r_1 < r_2 \text{ or } r_1 = r_2 \wedge F_1 =< \wedge F_2 = \leq .$$

The minimum of (r_1, F_1) and (r_2, F_2) is defined below:

$$min((r_1, F_1), (r_2, F_2)) = \begin{cases} (r_1, F_1) & \text{if } (r_1, F_1) < (r_2, F_2) \\ (r_2, F_2) & \text{otherwise} \end{cases}$$

A DBM D is *canonical* iff for any $0 \leq i, j, k \leq (m+1)$, $D_{i,k} \leq D_{i,j} + D_{j,k}$. A DBM D is *satisfiable* iff there is no sequence of indices $0 \leq i_1, \cdots, i_k \leq (m+1)$ such that $D_{i_1,i_2} + D_{i_2,i_3} + \cdots + D_{i_k,i_1} < (0, \leq)$. An unsatisfiable DBM D represents an empty clock zone.

A leaf represents the end of a path. When we start at a leaf to calculate time constraints backwards, we do not know the exact value of the global clock when the execution of a path ends. We assume its value is d. We need not assume a value for any local clock. Thus their values range from 0 to ∞. Their exact value ranges can be computed during backward calculation. The final DBM $\mathcal{D}_0$ is then defined below:

$$D_0 = \begin{pmatrix} (0, \leq) & (-d, \leq) & (0, \leq) & \cdots & (0, \leq) \\ (d, \leq) & (0, \leq) & (d, \leq) & \cdots & (d, \leq) \\ (\infty, <) & (\infty, <) & (0, \leq) & \cdots & (\infty, <) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\infty, <) & (\infty, <) & (\infty, <) & \cdots & (0, \leq) \end{pmatrix} \quad (6.2)$$

6.3.2 Operations on DBMs

Backward calculating time constraints following an edge e which is from source location s' to target location s has been explained in [Yov98]. Let $I(s')^X$ be the assertion on clocks in location invariant of s', and ψ^X be the assertion on clocks within edge e. D is the time constraint at s represented by a DBM. $I(s')^X$ and ψ^X are represented by DBMs as well. The time constraint D' at s' is defined as follows:

$$D' = ((([\lambda := 0]D) \wedge I(s')^X \wedge \psi^X) \Downarrow) \wedge I(s')^X \qquad (6.3)$$

"$\wedge$" is conjunction of two clock zones. Calculating $D' = D^1 \wedge D^2$ sets $D'_{i,j}$ to be the minimum value of $D^1_{i,j}$ and $D^2_{i,j}$, i.e.,

$$D'_{i,j} = min(D^1_{i,j}, D^2_{i,j}).$$

"$\Downarrow$" is time predecessor. Calculating $D' = D \Downarrow$ sets the lower bound of each clock to 0, i.e.,

$$D'_{i,j} = \begin{cases} (0, \leq) & \text{if } i = 0 \\ D_{i,j} & \text{if } i \neq 0 \end{cases}$$

"$[\lambda := 0]D$" is reset predecessor. Calculating $D' = [\lambda := 0]D$ is as follows:

1. Resetting a clock x to 0 corresponds to substituting x by x_0. Let x' be a clock which is not reset. Before resetting, we have constraints $x' - x_0 \leq c_1$ and $x' - x \leq c_2$. After resetting, we obtain constraints $x' - x_0 \leq c_1$ and $x' - x_0 \leq c_2$ by replacing x with x_0. Then conjunction is applied on yielding these constrains $x' - x_0 \leq min(c_1, c_2)$. Hence,

when we calculate time constraints from after resetting back to before resetting, we substitute $x' - x_0$ by $min(x' - x_0, x' - x)$ and $x_0 - x'$ by $min(x_0 - x', x - x')$. Therefore, for a clock x_i which is not reset, update its upper and lower bounds as follows:

(a) $D'_{i,0} = min\{D_{i,k} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$.

(b) $D'_{0,i} = min\{D_{k,i} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$.

2. On the other hand, for a clock x_k which is reset, its value before resetting can be any non-negative real number. Thus its lower bound is 0 and upper bound is ∞, i.e., $D'_{0,k} = (0, \leq)$ and $D'_{k,0} = (\infty, <)$. Furthermore, for any other clock x_j $(j \neq k \wedge j > 0)$, $D'_{k,j} = (\infty, <)$.

3. For a clock x_i which is not reset and a clock x_k which is reset, update $x_i - x_k$ as $D'_{i,k} = D'_{i,0}$. (Note that this step must be done after the upper bound of x_i is updated.)

4. For two clocks x_i and x_j that are not reset, $D'_{i,j} = D_{i,j}$.

D' needs to be changed to canonical form after each operation. This is done using the Floyd-Warshall algorithm [Flo62, War62] to find the all-pairs shortest paths.

The reset operation in backward DBM calculation needs special treatment, which is not explained in [Yov98]. Consider the example in Figure 6.2. We start computation at node s_3 with DBM D_0 (for the sake of simplicity, we

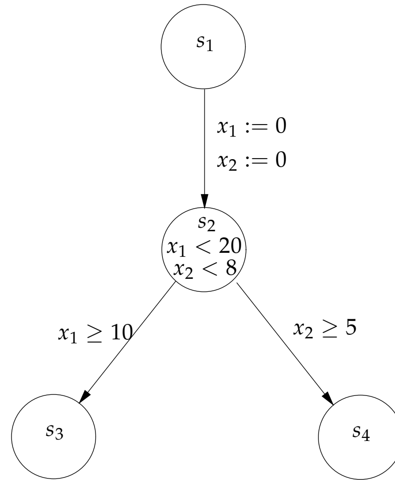

Figure 6.2: An example

neglect the global clock here.):

$$\begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) \\ (\infty,<) & (0,\leq) & (\infty,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}$$

The clock x_1 is encoded in the second row and x_2 in third. After backward calculation to s_2, we obtain a new DBM D'

$$\begin{pmatrix} (0,\leq) & (-2,<) & (0,\leq) \\ (20,<) & (0,\leq) & (20,<) \\ (8,<) & (-2,<) & (0,\leq) \end{pmatrix}$$

The DBM calculated backwards at s_1 is

$$\begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) \\ (\infty,<) & (0,\leq) & (\infty,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}$$

This last DBM means the path from s_1 to s_3 is possible, while, in fact, it is not. This situation implies that the backward reset operation loses some

useful information which can tell whether the path is possible or not. $D'_{2,1}$ represents $x_2 - x_1 < -2$, which means x_1 goes longer than x_2. However, the fact that x_1 and x_2 are reset at same time requires their time values must also be the same. The contradiction reveals this path is impossible. Therefore, we add an extra operation before the reset operation: If more than one clock is reset at the same time, we check whether an upper bound of the differences among them is smaller than 0. If the answer is yes, the DBM cannot be satisfiable.

6.4 Calculating the timed precondition of a DAG

We describe now how to add the time constraints for the DAG precondition. The backward calculation of the timed path precondition is as follows:

1. Mark each leaf node as *old* and all other nodes as *new*. Attach the assertion $\varphi = true$ on program variables and the assertion on clocks represented by DBM $\mathcal{D}_0$ to each leaf, noted by $\varphi \wedge \mathcal{D}_0$.

2. While there are nodes marked with *new* do

   (a) Pick up a node $\overline{z}$ that is marked *new* such that all its successors $Z = \{z_1, \dots, z_k\}$ are marked *old*.

   (b) Assume each $z_i \in Z$ has an assertion attached over program variables and clocks. The assertion has the form of

   $$\bigvee_{1 \leq j \leq m_i} (\varphi_{i,j} \wedge \mathcal{D}_{i,j}).$$

   ($m_i = 1$ if z_i is a leaf.) Since $\varphi_{i,j}$ is an assertion on program variables and $\mathcal{D}_{i,j}$ is an assertion over clocks, $\varphi_{i,j}$ and $\mathcal{D}_{i,j}$ must be

updated to $\varphi^R_{i,j}$ and $\mathcal{D}^R_{i,j}$ separately when we calculate precondition on $\overline{z}$ following the edge from $\overline{z}$ to z_i. We obtain $\varphi^R_{i,j}$ from $\varphi_{i,j}$ according to formula (6.1) and $\mathcal{D}^R_{i,j}$ from $\mathcal{D}_{i,j}$ according to formula (6.3).

(c) Attach

$$\bigvee_{\substack{z_i \in Z \\ 1 \leq j \leq m_i}} (\varphi^R_{i,j} \wedge \mathcal{D}^R_{i,j}) \tag{6.4}$$

to node $\overline{z}$. Mark $\overline{z}$ as *old*. Note that when $\varphi^R_{i,j} = false$ or $\mathcal{D}^R_{i,j}$ is not satisfiable, $\varphi^R_{i,j} \wedge \mathcal{D}^R_{i,j}$ is removed from formula (6.4).

3. When an initial node is reached during the backward calculation, the combination of conditions over program variables that it represents (refer to Section 5.3 for detail) must be combined via conjunction with the accumulated precondition in order to get the initial precondition for this node, because this combination is not processed during the backward calculation. The combinations represented by non-initial nodes are processed through the edges pointing to them. All initial preconditions are combined into a disjunction together to form the final initial precondition.

6.5 A running example

Let us consider the following example in Figure 6.3. A timed system is composed of two processes represented by programs 1 and 2. $v1, cont, v2, f1$ and $f2$ are local variables and pkt and ack are shared variables.

Program 1
```
    begin
p1:   v1:=1;
p2:   while (true) do
      begin
p3:     pkt:=v1;
p4:     wait(ack>0,l,f1);
p5:     if (f1=0) then
        begin
p6:       ack:=0;
p7:       v1:=v1+1
        end
        else
p8:       cont:=1
      end
    end.
```

Program 2
```
    begin
q1:   while (true) do
      begin
q2:     wait(pkt>0,-1,f2);
q3:     v2:=pkt;
q4:     pkt:=0;
q5:     ack:=1
      end
    end.
```

Figure 6.3: A simple concurrent real-time system

The semantics of the *wait* statement is described as follows. It has three parameters. The first one is the condition it waits for to become true. The second is the time limit and the third is a variable. A timer is started when the statement is executed. If the time limit is reached before the condition becomes true, a timeout is triggered and the variable is set to 1. If the condition becomes true before timeout, the variable is set to 0 and the timer is stopped. It is not appropriate to detect whether the *wait* statement timeouts or not by testing the condition because the condition may not be accessed after the *wait* statement. That the time limit is -1 means the process can wait for the condition forever without timeout. In this example, l is a parameter which is the time limit of a timer. (Note that l can be substituted to a constant as well.) If the condition $ack > 0$ is not detected before the

time limit is reached, a timeout would be triggered. The value range of l is computed automatically during precondition calculation and given by a predicate in the precondition. The ranges for program variables are given in the precondition as well.

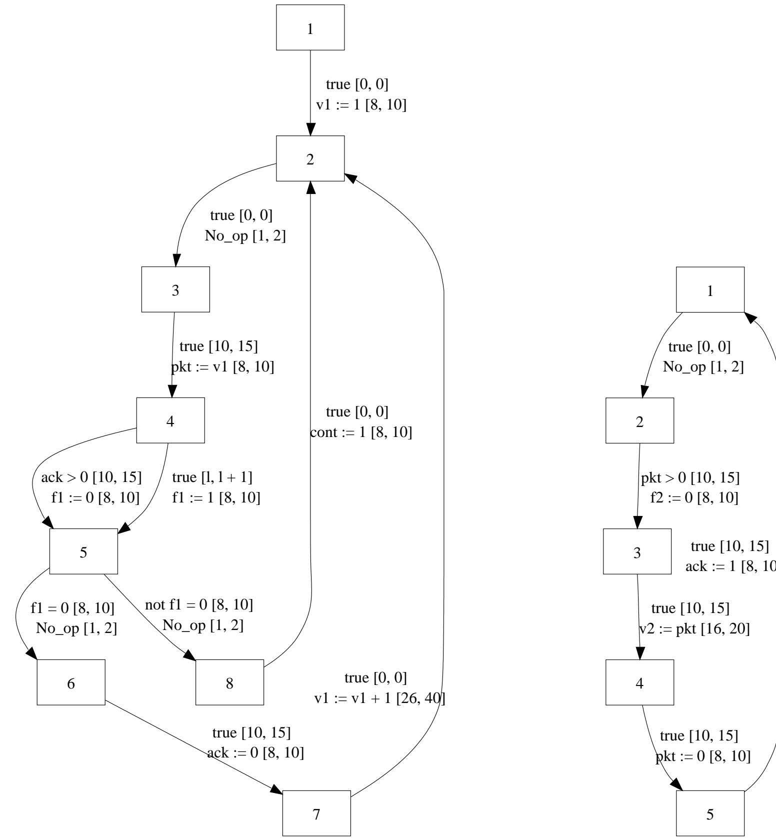

Figure 6.4: Program 1 and 2

The corresponding transition system of Program 1 and 2 is shown in Figure 6.4. The time bounds are chosen as follows: the bound for condition *true* in the timeout transition is $[l, l+1]$ and the bound for condition *true* in other transitions is $[0, 0]$[1]; the bound for evaluating the nontautological en-

[1]When the upper bound is zero, we use $x \leq 0$ as the assertion over clocks.

abling condition of a transition which does not access any shared variable is $[8,10]$; the bound is $[10,15]$ if the transition accesses a shared variable; assigning an instant value to a variable is bounded by $[8,10]$; addition operation has the bounds $[10,20]$ and No_op has the bounds $[1,2]$.

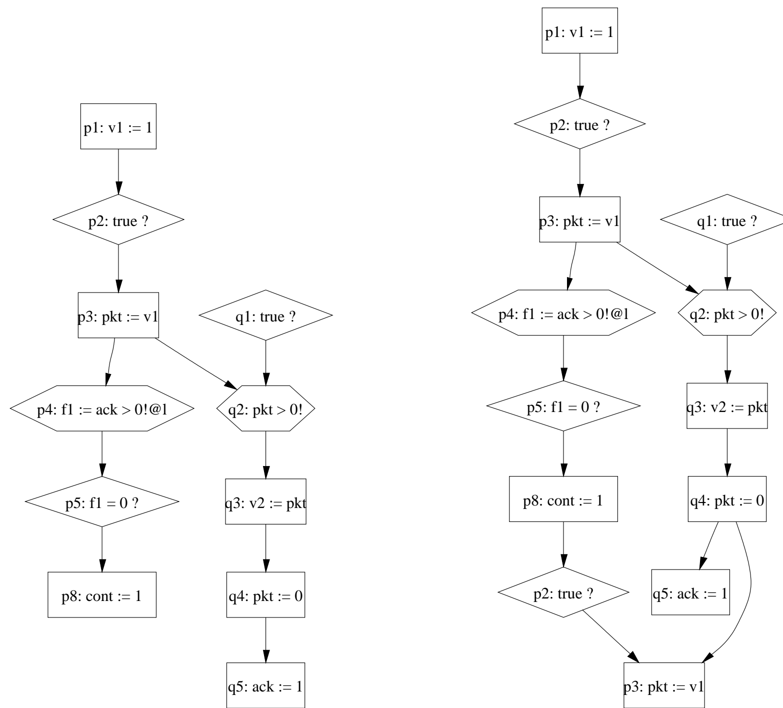

Figure 6.5: Partial order 1 and 2

In Figure 6.5, partial order 1 is on the left and partial order 2 is on the right. They are composed of flow chart nodes. When generating DAGs, they are translated into the ones composed of ETA transitions.

Figure 6.6 shows the initial node and a part of DAG nodes generated by the synchronization of the partial order 1 and the product of ETAs (translated from the TS in Figre 6.4). The dotted lines in the figure denote the part of the DAG being omitted. The ETAs and the full DAG are not displayed

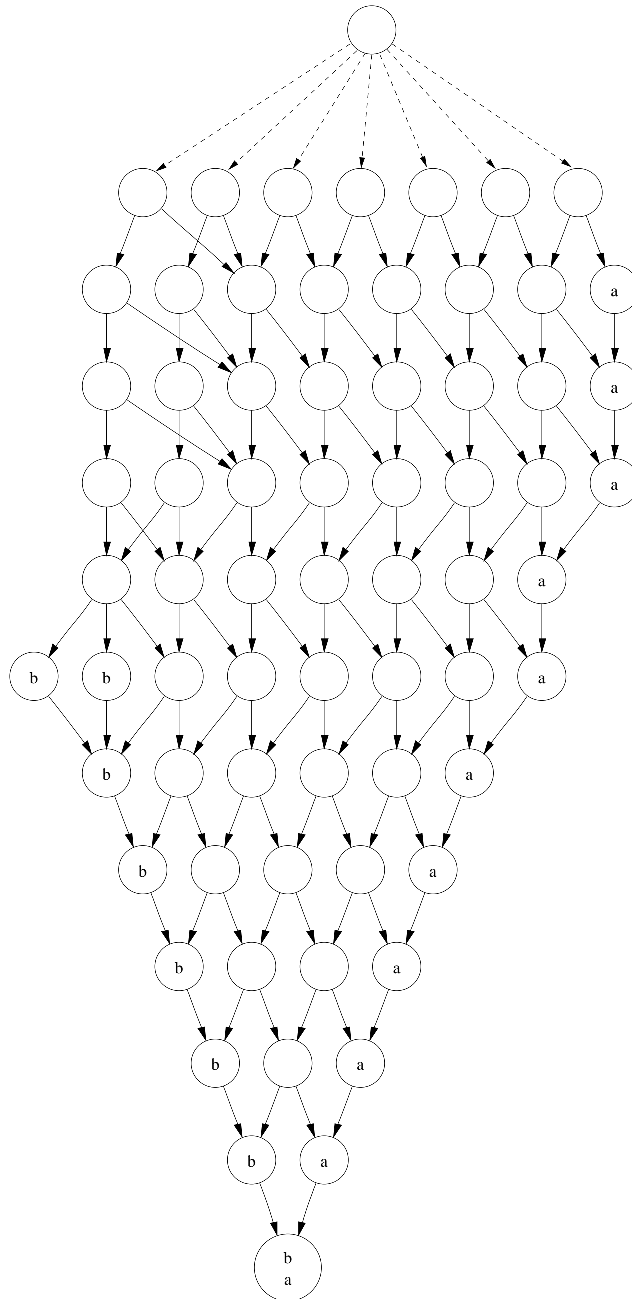

Figure 6.6: The DAG for Partial order 1

since they are too complicated to be illustrated in detail. Each DAG node is a compound location which contains a location in Process 1 and one in Process 2. More explanation for this figure is given in the next section.

The precondition[2] of the partial order 1 is calculated on the DAG in Figure 6.6. The precondition is

$$ack \leq 0 \wedge pkt \leq 0 \wedge 48 \leq l \leq 84.$$

The precondition of the partial order 2 is calculated in the same way and shown below:

$$(ack \leq 0 \wedge pkt \leq 0 \wedge 13 \leq l \leq 65) \vee (ack \geq 1 \wedge pkt \leq 0 \wedge 13 \leq l \leq 14).$$

(Note that the DAG corresponding to the partial order 2 is omitted.)

6.6 Time unbalanced partial order

We denote that a partial order is *executable* if its precondition is not *false*; otherwise, the partial order is *unexecutable*. Obviously, if we replace l by 40 in Program 1, the partial order 1 is unexecutable, while the partial order 2 is executable. The difference between these two partial orders is that Process 1 executes a few more statements in the partial order 2 than in the partial order 1, while Process 2 executes the same statements in both partial orders. Both partial orders are executable in untimed systems because the extra statements in the partial order 2 do not change the precondition on program variables. This case reveals that time constraints could distinguish two partial orders that cannot be distinguished in untimed systems.

[2]The Omega library, which we used to simplify Presburger formula, operates on integers such that it simplifies $l < n$ to $l \leq (n-1)$.

Now we explain how time constraints affect preconditions in this case. In both partial orders, statement $q2$ is first enabled at the same time or after statement $p4$ becomes enabled because $q2$ can only be enabled after both $q1$ and $p3$ are executed, while $p4$ could be enabled after $p3$ is executed but before $q1$ is executed. The sequences $\langle p4, p5, p8 \rangle$ and $\langle q2, q3, q4, q5 \rangle$ in the partial order 1 are local to Process 1 and 2 respectively since there is no dependence between them. (Note that $p4$ in $\langle p4, p5, p8 \rangle$ behaves as the transition $true \rightarrow f1 := 1$, which can be seen in Figure 6.4.) Therefore, the execution of $\langle p4, p5, p8 \rangle$ is independent of the execution of $\langle q2, q3, q4, q5 \rangle$. To be executed, each sequence is translated into a sequence of automaton locations and edges. Let a be the last automaton location of the sequence $\langle p4, p5, p8 \rangle$ after translation, b that of the sequence $\langle q2, q3, q4, q5 \rangle$. a and b are labeled in Figure 6.6. A node labeled as only a or b means that this node contains a or b but not both. In the DAG, there are a group of interleaving paths, each of which represents an execution schedule of these two sequences and ends with a compound location containing both a and b. By adding all lower bounds and the upper bounds along these two sequences, it is easy to see that the maximum execution time of the sequence $\langle p4, p5, p8 \rangle$ is 73, while the minimum execution time of the sequence $\langle q2, q3, q4, q5 \rangle$ is 80, under the condition $l = 40$. This means that during the execution of any interleaving path, the system would definitely reach some states $\mathcal{Q}$ (which are DAG nodes labeled a in Figure 6.6) that contain the location a, but not b. (Computation on DBMs shows that a state containing only b is unreachable.) For each state after $\mathcal{Q}$, Process 1 will stay at a until Process 2 reaches b. However, at location a, the statement $p2$ is en-

abled and thus there is a time constraint in the location invariant of a that requires Process 1 to execute $p2$ and then leave a before Process 2 reaches b. In other words, the given partial order requires that the system reaches a state which contains a and b, but the time constraints make this state unreachable. Starting from an unreachable state, the backward calculation of path precondition gives $false$ as the result.

On the other hand, let a' be the last automaton location of the sequence $\langle p4, p5, p8, p2, p3 \rangle$ in the partial order 2. b is still the last location of $\langle q2, q3, q4, q5 \rangle$. By applying the algorithm in the next section, we know that the maximum execution time of $\langle p4, p5, p8, p2, p3 \rangle$ is 92. For the partial order 2, therefore, the system can reach the final state which contains both a' and b, i.e., the system could enter a state after which either Process 1 stays at a' until Process 2 reaches b or Process 2 stays at b until Process 1 reaches a'. Thus the precondition of the partial order 2 is not $false$.

6.6.1 The definition

The problem above can be formally interpreted by *time unbalanced partial order*, which is defined as follows. Let ρ be a partial order in a system composed of n ($n > 1$) processes $P_1, \ldots, P_n$. Let $\rho^i = \alpha^i_0 \alpha^i_1 \ldots \alpha^i_{m_i}$ be the *projected path* which is the projection of ρ onto P_i. Each α^i_j ($0 \leq j \leq m_i$) is a statement of P_i. For example, in the partial order 1 in Figure 6.5, there are two projected paths: $\rho^1 = \langle p1, p2, p3, p4, p5, p8 \rangle$ and $\rho^2 = \langle q1, q2, q3, q4, q5 \rangle$. Another example is the partial order in Figure 6.7. This partial order is defined over Program 1 and 2 in Figure 6.3 as well. It is similar to the partial order 2 in Figure 6.5 except that the partial order 2 contains the statement $q5$

in Program 2, while this partial order does not. The edge from $q4$ to the second appearance of $p3$ in Figure 6.7 means that $q4$ must be fired earlier than the second appearance of $p3$. This partial order has two projected paths:

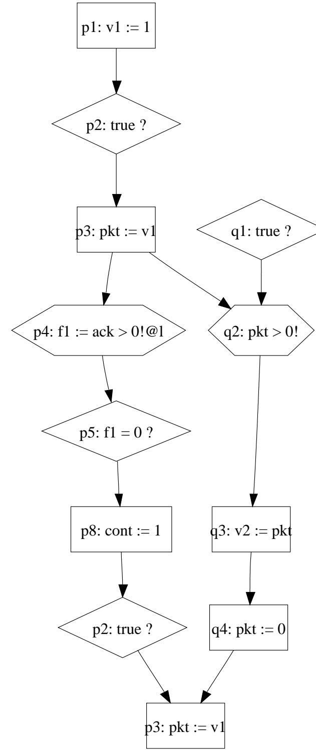

Figure 6.7: An example partial order

$\rho^3 = \langle p1, p2, p3, p4, p5, p8, p2, p3\rangle$ and $\rho^4 = \langle q1, q2, q3, q4\rangle$.

Let $T(\rho^i)$ be the execution time of ρ^i. For any two sequences ρ^i and ρ^j ($1 \leq i, j \leq n$ and $i \neq j$), $T(\rho^i) < T(\rho^j)$ if ρ^i and ρ^j satisfy one of the following conditions:

1. The last statement of ρ^j, which is $\alpha^j_{m_j}$, *depends* on $\alpha^i_{m_i}$, which is the last statement of ρ^i. That is, $\alpha^i_{m_i}$ is required by the partial order to be fired

before $\alpha^j_{m_j}$ is fired. For example, the second appearance of $p3$ in ρ^3, which is the last statement of ρ^3, depends on $q4$, the last statement of ρ^4, because of the edge from $q4$ to the second appearance of $p3$ in Figure 6.7.

2. If $\alpha^i_{m_i}$ and $\alpha^j_{m_j}$ do not depend on each other, the maximum execution time of ρ^i is smaller than the minimum execution time of ρ^j. For example, the maximal execution time of ρ^1 in the partial order 1 is 110 and the minimal execution time of ρ^2 is 117.

The projected paths ρ^i and ρ^j form a *time unbalanced projected path pair*, where ρ^i is the *short* projected path and ρ^j is the *long* one. For two projected paths ρ^k and ρ^l, $T(\rho^k) \approx T(\rho^l)$ if $T(\rho^k) \not< T(\rho^l)$ and $T(\rho^l) \not< T(\rho^k)$. A partial order ρ is a time unbalanced partial order if there exist some time unbalanced projected path pairs in ρ. For example, the partial order 1 is a time unbalanced partial order because ρ^1 and ρ^2 satisfy the second condition and then construct a time unbalanced projected path pair. The partial order in Figure 6.7 is time unbalanced as well since ρ^3 and ρ^4 satisfy the first condition.

Time unbalanced partial order is very common in timed automata. For example, a linear partial order, where there is only one root statement, one leaf statement and one path from the root to the leaf, is a time unbalanced partial order. We denote that a partial order ρ_1 is a *prefix* of another partial order ρ_2 if any equivalent path represented by ρ_1 is a prefix of an equivalent path represented by ρ_2. ρ_2 is *longer* than ρ_1. A time unbalanced partial order ρ whose precondition is *false* is *extendable* if it is a prefix of a longer path ρ' whose precondition is not *false* and the projected paths which

have the longest execution time in both ρ and ρ' are the same. The second requirement ensures that only the projected paths which have short execution time are extended.

When calculating the precondition of a time unbalanced partial order, there is a danger that the partial order precondition could be *false*, but the precondition of the corresponding untimed partial order is not *false*, which is the case that occurred in the partial order 1. However, not all time unbalanced partial orders have *false* as their precondition. Although an unbalanced pair of projected paths could lead the system into entering a state from which the final state is unreachable, whether the system enters such a state invariantly or not depends on not only time unbalanced projected path pairs, but also their successive statements. Let a be the last automaton location of the short projected path of a pair. Let τ be any transition starting at a and u be the upper bound for its enabling condition. If τ is continuously enabled longer than u before the long projected path ends, the partial order precondition is *false* because u forces τ to be fired (but τ does not appear in the partial order). On the other hand, if a partial order is not extendable, i.e., it is unexecutable due to some time constraints other than the gap between execution times of two processes in an unbalanced projected path pair, its precondition should be *false*, though the corresponding untimed partial order could have a non-false precondition.

Computing each projected path's running time is as follows. Let n_i be a node in the partial order. Let $MAX(n_i)$ and $MIN(n_i)$ be the maximum execution time and the minimum execution time of a projected path which contains n_i after n_i is executed, and $max(n_i)$ and $min(n_i)$ be the max-

imum execution time and the minimum execution time of n_i. $max(n_i)$ and $min(n_i)$ are obtained from time bounds. If n_i is a root node, $MAX(n_i) = max(n_i)$ and $MIN(n_i) = min(n_i)$. Otherwise, assume n_i has k predecessors $n_j, \ldots, n_{j+k-1}$:

$$MAX(n_i) = \max\{MAX(n_j), \ldots, MAX(n_{j+k-1})\} + max(n_i)$$

$$MIN(n_i) = \max\{MIN(n_j), \ldots, MIN(n_{j+k-1})\} + min(n_i).$$

A projected path's maximum and minimum execution time is the MAX and the MIN of its last node respectively.

The algorithm above only gives an estimated execution time, not an accurate time, because the real execution time may depend on the transitions that do not appear in the partial order. Therefore, it cannot be used to substitute the one in Section 6.4 to calculate preconditions. But it is adequate enough to tell us whether a particular partial order is time unbalanced.

Time unbalanced partial order reveals why the lower bound of l in the partial order 1 is 48. It seems that Process 1 would timeout even when $l < 48$. But 48 ensures that the partial order 1 is not time unbalanced. For the same reason, the lower bound of l is 13 in the partial order 2.

6.6.2 A remedy to the problem

Though it is helpful to indicate that a partial order is time unbalanced in addition to telling users the partial order precondition, it is better to let the users learn more about the partial order than that the partial order precondition is *false*. The basic reason that a partial order is time unbalanced is

that there is a gap between the executions of two processes in an unbalanced pair. If we allow the process which has the short projected path to continue running, i.e., leave its last location on the projected path, the gap could be filled and the unbalanced pair changed to a balanced one. But the successive transitions should not alter the interprocess partial ordering relations, i.e., the successive transitions should not change the value of shared variables used by other processes.

Since the successive statements of a projected path are different in different partial orders and in different systems, and the execution time gap between a pair of unbalanced projected paths is different among partial orders and systems, it could be difficult to explore all possibilities to allow a process executing successive statements, in particular, in the case of nondeterminism. A simple method, which allows processes to execute extra transitions and does not change the partial order, is, for each projected path, to remove the time constraints in location invariant of the last node during constructing the DAG. This method allows a process to execute any successive statements without caring which statements are chosen and how many statements need be executed to fill the time gap. After applying this method, the precondition of the partial order 1 under $l = 40$ is $ack \leq 0 \wedge pkt \leq 0$, which is what we expect. The new precondition is noted as the *underlying precondition*.

However, this method only remedies the gap between execution times among processes for time unbalanced partial orders and therefore it cannot be used carelessly. It cannot be applied to partial orders that are not extendable, since for a time unbalanced and not extendable partial order, it is not

the gap between execution times of two processes that makes the precondition of the partial order become *false*. Thus, the method would calculate a wrong precondition if we apply it to an unextendable partial order. For example, consider the following programs. We choose the time bounds ac-

Program 3

```
      begin
P1:      wait(v>0,40,f1);
P2:      if (f1=0) then
P3:         v1 := 0
         else
P4:         v1 := 1;
      end.
```

Program 4

```
      begin
Q1:      v : =1
      end.
```

Figure 6.8: A negative example

cording to Section 6.5, i.e., the upper bounds for the enabling condition and the transformation of transition Q1 are 15 and 10, respectively. The time limit of the timer in statement P1 is 40 according to the semantics of the `wait` statement. The partial order on the left part of Figure 6.9 is time un-

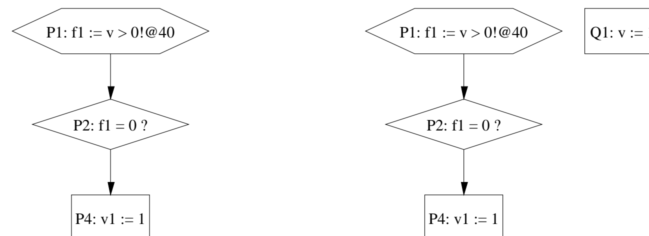

Figure 6.9: Two unextendable partial orders

balanced and cannot be extended to an executable balanced partial order. It is also not executable since it can only be extended to the partial order on the right of the figure, which is not executable because the condition $v > 0$

is satisfied before time progresses up to 40 units and `f1` is set to 0.

Until now, the methodology assumes that all processes in a partial order start at the same time. It is a reasonable assumption in many cases, especially when a partial order contains the first transition of each participant process. In fact, a partial order does not necessarily start at the first transition of every process. The algorithm in Section 6.4 also does not have such a requirement. This gives testers an advantage because it may be very difficult to know the whole sequence of transitions from the beginning that causes an error but it is easy to know a part of the sequence that causes the error. Testers are able to concentrate on this part of the sequence, without worrying about the reat of the sequence. However, this feature causes troubles. The precondition of a partial order could be $false$ because one process starts later than another. Let $\overline{a}_i$ be the first location of Process i in the partial order. That is, we assume the system begins to run from a state containing all $\overline{a}_i$ locations, while this state is unreachable from the initial state. But it is often difficult to know which starting state is a reachable state from the initial state. In such cases, the concepts introduced previously can be extended easily to handle this situation. Removing the time constraint in location invariant of every first location $\overline{a}_i$ allows that each process starts at a different time and then we can calculate the underlying precondition.

6.6.3 An application of the remedy method

This method has another usage in that it also releases users from specifying a set of partial orders to obtain a complete precondition. One partial order could give them the complete precondition. For instance, we may

be interested for the example in Section 6.5 in the maximal upper bound and minimal lower bound of l between which timeout may occur. When $z < 0$, the conditions of l in the partial order 1 and the partial order 2 are $48 \le l \le 84$ and $13 \le l \le 65$. It is obvious that 48 is not the minimal lower bound since $48 > 13$ and 65 is not the maximal upper bound since $65 < 84$. Furthermore, we cannot draw the conclusion that 13 is the minimal lower bound and 84 the maximal upper bound because of an obvious fact that timeout would occur when $l = 0$. We have to check more partial orders, such as the partial order 3 and the partial order 4 in Figure 6.10, in order to obtain the proper bounds. These partial orders are similar to the partial

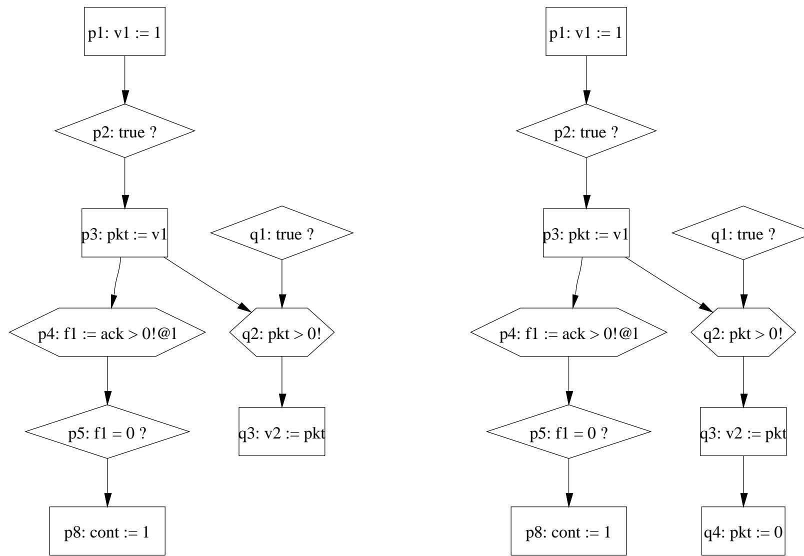

Figure 6.10: Partial order 3 (left) and 4 (right)

order 1 in Figure 6.5, except that Process 2 has fewer transitions involved in them. The conditions of l in the partial order 3 and 4 are:

$$(ack \le 0 \wedge pkt \le 0 \wedge 12 \le l \le 49) \vee (ack \ge 1 \wedge pkt \le 0 \wedge 12 \le l \le 14)$$

and

$$ack \leq 0 \wedge pkt \leq 0 \wedge 30 \leq l \leq 74$$

respectively. These conditions are still not what we expect. This example demonstrates it is not easy to obtain the proper bounds for time parameters.

However, we obtain promising bounds when we apply the remedy method to the partial order 1. The underlying precondition of the partial order 1 is

$$(ack \leq 0 \wedge pkt \leq 0 \wedge 0 \leq l \leq 92) \vee (ack \geq 1 \wedge pkt \leq 0 \wedge 0 \leq l \leq 14).$$

0 is the lower bound we expect. 92 is believed to be the correct upper bound after we check the time constraints of the partial order 1 carefully. Furthermore, the new condition also tells us that, under $0 \leq l \leq 14$, timeout can occur even when $ack > 0$. This coincides with the time constraints of statement $p4$. The reason why we use this method in the partial order 1 is that sequence $\langle q2, q3, q4, q5 \rangle$ is the maximal one before condition $ack > 0$ holds. The minimal lower bound is obtained by executing $\langle q2, q3, q4, q5 \rangle$ after $p4$ is executed, and the maximal upper bound is obtained by executing $p4$ after $\langle q2, q3, q4, q5 \rangle$ is executed. Using the partial order 2 generates the same bounds but it takes much longer to calculate the precondition for the partial order 2 than for the partial order 1.

6.7 Summary

We have presented an algorithm to calculate the precondition of a partial order. The input to the algorithm is a DAG generated with respect to the

partial order and the extended timed automata in the previous chapter. The output is a set of conditions, each of which contains a boolean expression over program variables and time parameters, and a DBM. The DBM is mainly used to analyze reachability. The disjunction of all boolean expressions is the precondition that we are interested in.

We gave the definition of a time unbalanced partial order and designed an algorithm to estimate the execution time of each projected path in the partial order in order to check if a partial order is time unbalanced. We also proposed a remedy method to compute the underlying precondition of a time unbalanced partial order which is extendable. In addition, the remedy can be used to simplify the computation of the minimum and the maximum bounds of a time parameter.

The methodology of constructing a DAG and calculating the precondition of the DAG is not limited to the model defined in Chapter 5. It can be applied to other system models, e.g., Timed Transition Diagrams (TTDs) [HMP94], with appropriate modifications. We need to modify the rules of translating a TS into ETAs to handle the translation from a TTD to ETAs, and the rules of generating a partial order from a given execution. Only a minor modification is needed in this case since TTDs have structure similar to that of TSs, with the exception that a TTD transition has one pair of bounds, while a TS transition has two pairs. Generally speaking, if a system can be modeled by ETAs and a partial order is given over these ETAs, our methodology can be applied to calculate the precondition of the partial order.

In the literature, there are several papers studying parametric model

checking [AHV93, HRSV02]. Constraints on parameters can be deduced by parametric model checking as well. The advantage of parametric model checking is that it is an automatic technique to obtain the constraints with respect to a property. But it might search a very large state space and visit irrelevant states. In contrast, our method is not automatic, but requires the users to specify the partial order to compute the constraints. In other words, it involves human intelligence. Hence, the advantage of our method is that it can search a far smaller state space than parametric model checking does if users provide an appropriate partial order. Of course, choosing an appropriate partial order may not be achieved easily for a complicated system. Therefore, our method is more suitable for advanced users than for inexperienced users.

The algorithms for computing the precondition over a DAG, estimating the execution time of each projected path and removing timing constraints from location invariants were implemented in RPET. Figure 6.4, 6.5, 6.6, 6.7, 6.9 and 6.10 were generated by RPET. Note that the graph automatically generated by RPET for Figure 6.4 was modified, such that a transition is represented by one edge, rather than two edges as in Figure 5.14, in order to give a succinct graph.

Chapter 7

Calculating the probability of a partial order

In this chapter we shall discuss a methodology which exploits the probability of executing a path in a concurrent real-time system. Our probabilistic model of real-time systems requires that, for each transition, the period from the time when its enabling condition becomes satisfied to the time when it is fired is bounded. The length of the period obeys a continuous probability distribution, in particular, a uniform distribution. Instead of using a path as a linear order between occurrences of transitions, it is sometimes more natural to look at a partial order that can be extracted from a path, and represent dependencies between transitions, according to the processes in which they participate. We consider then an extended problem, where we calculate for given initial conditions the probability that the system executes one of a group of paths represented by the partial order.

7.1 The model

7.1.1 Probabilistic transition systems

We create the probabilistic transition systems on top of the transition systems proposed in Chapter 5. At first, we look at the transition systems from a different angle. A transition of the form $c \rightarrow f$ with two pairs of time bounds $[l, u]$ and $[L, U]$ is regarded as two transitions in the probabilistic transition systems. One transition is of the form $c \rightarrow nil$ with bound $[l, u]$ and the other is of the form $true \rightarrow f$ with $[L, U]$. The transformation nil (f) is executed instantaneously when the enabling condition c ($true$) is satisfied continuously for at least l (L) time but at most u (U) time. From now on we do not distinguish the bounds $[l, u]$ and $[L, U]$ any more and deem all transitions have a uniform form. Each transition must be enabled for a period of time between its lower bound l and upper bound u before its transformation is fired. For brevity, we say the transition is fired, rather than the transformation of the transition is fired.

To introduce the probability, we associate a count-down clock [ACD91] $cl(\alpha)$ to a transition α with enabling condition $en(\alpha)$ and transformation F_α over program variables, and time bounds l_α and u_α. When α becomes enabled, i.e., $en(\alpha)$ holds, $cl(\alpha)$ is set to an initial value which is chosen randomly from the interval $[l_\alpha, u_\alpha]$ according to the uniform distribution with the density function $\hat{f}_\alpha = \frac{1}{u_\alpha - l_\alpha}$ on $[l_\alpha, u_\alpha]$. Then the clock begins to count down. When it reaches 0, α is triggered. If α is disabled before $cl(\alpha)$ reaches 0, $cl(\alpha)$ is set to 0 as well, but α is not triggered. In addition to the clocks per each transition, we have a global clock gt. The global clock is used to

measure the time elapsed since the system began to run so that its reading keeps increasing after being started. Any clock except the global clock stops running when its reading is 0.

Definition 1 *A* state *of a probabilistic transition system $\mathcal{T} = \langle V, E \rangle$, where V is the set of program variables and E is the set of transitions, contains: (1) An assignment to the program variables V; (2) A non-negative real-time value for each transition clock $cl(\alpha)$ for $\alpha \in E$; (3) A non-negative real-time value for the global clock gt.*

We denote the value of a clock $cl(\alpha)$ in a state s by $cl(\alpha)(s)$ and the value of gt in s by $cl(gt)(s)$. Similarly, the value of a variable v in s is $v(s)$. We also generalize this and write $V(s)$ for the valuation of all the variables V at the state s. A transition α is *enabled* at state s if $en(\alpha)$ holds in s; then we say that $s \models en(\alpha)$ holds.

Definition 2 *An* initial state *of a probabilistic transition system $\mathcal{T} = \langle V, E \rangle$ assigns a value to the clock of every transition $\alpha \in E$ that is enabled in the initial state. Each value is chosen randomly between l_α and u_α according to the uniform distribution. The initial state assigns the value 0 to every clock $cl(\alpha')$ for all disabled $\alpha' \in E$ in the initial state and to the global clock.*

Definition 3 *A* system *$\mathcal{S}$ is a pair $\langle \mathcal{T}, s \rangle$ with s an initial state of $\mathcal{T}$. Thus, we assume each system has a given initial state.*

Probabilistic behavior is reflected in repeated executions, not a single one. We define a *probabilistic execution* of the system induced from multiple executions.

Definition 4 *A probabilistic execution of a system $\mathcal{S}$ is a finite sequence of the form $s_0 g_1 \alpha_1 s_1 g_2 \alpha_2 \ldots$ where s_i, g_i are states, and α_i are transitions. An execution has to satisfy the following constraints:*

- *s_0 is the initial state of $\mathcal{S}$.*
- *For any adjacent pair of states s_i, g_{i+1} (representing time passing):*
  - *For the clock of any enabled transition α we have that*

    $$cl(\alpha)(s_i) - cl(\alpha)(g_{i+1}) = cl(gt)(g_{i+1}) - cl(gt)(s_i),$$

    which means that all clocks move at the same speed. The clock of any disabled transition β remains 0.
  - *For every variable $v \in V$, $v(s_i) = v(g_{i+1})$.*
- *For any sequence g_i, α_i, s_i on the path (the execution of transition α_i), the following hold:*
  - *$g_i \models en(\alpha_i)$ and $V(s_i) = F_{\alpha_i}(V(g_i))$.*
  - *$cl(\alpha_i)(g_i) = 0$ and $cl(gt)(g_i) = cl(gt)(s_i)$*
  - *For each $\beta \in E$ we have that if $g_i \models \neg en(\beta)$ and $s_i \models en(\beta)$ (β became enabled by the execution of α_i), or $\beta = \alpha_i$ and $s_i \models en(\beta)$ (although $\alpha_i = \beta$ was executed, it is enabled immediately again), then $cl(\beta)(s_i)$ is a random non-negative value which is chosen between l_β and u_β according to the uniform distribution. If $g_i \models en(\beta)$ and $s_i \models \neg en(\beta)$, then $cl(\beta)(s_i) = 0$. That is, when a transition becomes disabled, its clock is set to zero. Otherwise, $cl(\beta)(g_i) = cl(\beta)(s_i)$.*

- *For $\beta \neq \alpha_i$ such that $g_i \models en(\beta)$ we have that $cl(\beta)(g_i) > 0$. The reason is that the probability of two events being triggered at the same time*[1] *is 0 from probability point of view. Detailed discussion can be seen in [KNSS00].*

Definition 5 *A* path *is a finite sequence of transitions $\sigma = \alpha_1\alpha_2\alpha_3\ldots$. A path σ is* consistent *with an execution ρ if σ is obtained by removing all the state components of ρ.*

7.1.2 Calculating participating transitions

Since transitions may occur several times on a path, we refer to the *occurrences* of transitions. So we either rename or number different repeated occurrences of transitions to identify them. In fact, we need to look not only at the transitions that occur on the path, but also at those that become enabled (but not executed).

As a preliminary step for the calculation performed in the next section, we need to compute the transitions that are enabled (but not necessarily executed) given a path σ. Assume that the transitions are executed according to the order in σ. Then consider the states of the form g_i, where the transition α is fired. We can ignore now the timing considerations, thus also the states of the form s_j (that are the same as the g_i states). The states $g_1, \ldots, g_n$ are easily calculated, as $V(g_i) = F_{\alpha_i}(V(g_{i-1}))$. Moreover, it is easy to calculate whether $g_i \models en(\beta)$ holds for $0 \leq i \leq n$.

Definition 6 *The* active interval *of an occurrence of a transition α_j, is a maximal*

[1] *Here a pair of synchronized transitions is considered as one transition.*

interval, where it is enabled. That is, each such interval is bounded by states g_i and g_j such that, for each $i \leq k \leq j$, $g_k \models en(\alpha_j)$. Furthermore, g_i and g_j are maximal in the sense that these conditions do not hold for the interval g_{i-1} to g_j (if $i > 0$) nor for the interval g_i to g_{j+1} (if $i < n$).

Thus, the transitions participating in a path σ are those that have a non-empty active interval.

7.1.3 An example system

In Figure 7.1, a system is composed of three processes, each of which is represented by a subsystem. Process 1 has four transitions a, b, c, d, process 2 has one transition g and process 3 has one transitions h. The transitions of the different subsystems are being put together into the set of transitions $\mathcal{T}$. The variables $V = \{s, v, w\}$ representing the "program counters" of the different processes are used to control the enabledness of these transitions. The initial state of the system has the assignment $s = 1$, $v = 1$ and $w = 1$. The bounds for transition a, b, c, d, g and h are $[1, 5]$, $[2, 5]$, $[1, 4]$, $[2, 4]$, $[2, 6]$ and $[3, 7]$ respectively. The density functions of transitions in this system are $\hat{f}_a = \hat{f}_g = \hat{f}_h = \frac{1}{4}$, $\hat{f}_b = \hat{f}_c = \frac{1}{3}$ and $\hat{f}_d = \frac{1}{2}$.

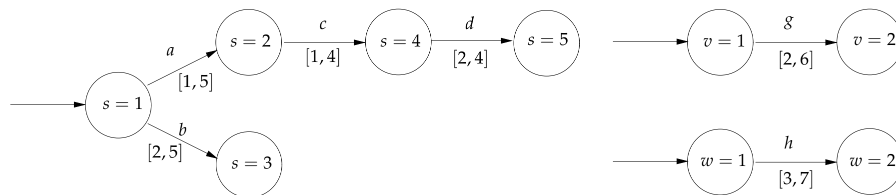

Figure 7.1: The example system

7.2 The probability of a path

7.2.1 Timing relations along a path

As per Definition 5, a path ρ is a sequence of transitions $\alpha_1\alpha_2\ldots\alpha_n$. The corresponding execution is $s_0g_1\alpha_1s_1g_2\alpha_2s_2g_3\ldots s_{n-1}g_n\alpha_ns_n$. The transition α_i $(1 \leq i \leq n)$ has the lower bound l_i and the upper bound u_i. The path is executed from state s_0 at global time 0, which is represented as x_0. Let x_i be $cl(gt)(g_i)$ or, equivalently, $cl(gt)(s_i)$, i.e., the value of the global clock when the transition α_i is fired. So we have a sequence of global time points $x_0x_1\ldots x_n$. We obtain the relation among these time points:

$$x_0 < x_1 < \cdots < x_n. \tag{7.1}$$

For a transition α_j that becomes enabled at x_i and is triggered at x_j, the duration of its enabledness, which is decided by the initial value of $cl(\alpha_j)(s_i)$ at x_i, satisfies the formula:

$$l_j \leq x_j - x_i \leq u_j. \tag{7.2}$$

Now let us consider an occurrence of a transition α' that does not appear in the given path but is first enabled at s_i, while disabled at s_j $(0 \leq i < j \leq n)$ or remains enabled after s_n. In the latter case, α' is said to be disabled by the end of path at x_n. Thus we need not distinguish these two cases. Let x_i and x_j be the global time points with respect to s_i and s_j respectively. $cl(\alpha')(s_i)$ is the initial value of the clock when α' becomes enabled. Let $x_{\alpha'} = x_i + cl(\alpha')(s_i)$. Then $x_{\alpha'}$ satisfies the following formulae:

$$l_{\alpha'} \leq x_{\alpha'} - x_i \leq u_{\alpha'}, \tag{7.3}$$

$$x_j < x_{\alpha'}. \tag{7.4}$$

Obviously, $x_{\alpha'} - x_i$, the initial clock value of α', is bounded by the lower bound and the upper bound of α'. The formula (7.4) must hold because otherwise α' would have been triggered before α_j is triggered and would have appeared in the path. All of the occurrences of transitions that are enabled at some states and disabled at some later states form the set $\{\alpha'_1, \alpha'_2, \ldots, \alpha'_m\}$. Every x_i or $x_{\alpha'_k}$ is the value of a random variable X_i or X'_k.

Consider for example a path $\rho = ag$ of the system in Figure 7.1. At time x_0, there are four enabled transitions a, b, g and h. According to ρ, a is fired earlier than others at time x_1. At this point, b is disabled, while g and h are continuously enabled. Also, c becomes enabled at x_1. To make it clear, we use x_a to replace x_1 and x_g to replace x_2. Therefore, we obtain constraints $1 \leq x_a \leq 5 \wedge 2 \leq x_b \leq 5 \wedge x_a < x_b$. At time x_2 after g is fired, both h and c are disabled by the end of the path. We obtain $x_a < x_g \ \wedge \ 2 \leq x_g \leq 6 \ \wedge \ 3 \leq x_h \leq 7 \ \wedge \ x_a + 1 \leq x_c \leq x_a + 4 \ \wedge \ x_g < x_h \ \wedge \ x_g < x_c$. The final constraint for the path is as follows:

$$(1 \leq x_a \leq 5) \wedge (2 \leq x_g \leq 6) \wedge (3 \leq x_h \leq 7) \wedge (2 \leq x_b \leq 5) \wedge \\ (x_a + 1 \leq x_c \leq x_a + 4) \wedge (x_a < x_b) \wedge (x_a < x_g) \wedge (x_g < x_h) \wedge (x_g < x_c).$$

7.2.2 Computing the probability of a path

For a path in which n transitions are taken and m transitions are not taken but have been enabled for a period of time, calculating the probability that the path is executed involves $n + m$ independent random variables $Y_1, \ldots, Y_n$,

$Y'_1, \ldots, Y'_m$. Each transition has a corresponding random variable (either Y_i or Y'_k) whose value is the initial value of the clock of the transition. The fact that a transition is enabled after another does not make their corresponding random variable dependent on each other because on every execution of a path, the initial value of the clock of a transition is chosen independently according to its probability distribution. Any transitions that are never enabled along the path do not compete with the transitions in the path for execution. Thus they do not contribute to the probability and need not be considered in the calculation of the probability.

The probability distribution of the path is the joint distribution of $Y_1, \ldots, Y_n, Y'_1, \ldots, Y'_m$. Let $\hat{f}_i$ $(1 \le i \le n)$ and $\hat{f}'_k$ $(1 \le k \le m)$ be the density functions of Y_i and Y'_k respectively. Let l_i and u_i be the lower bound and the upper bound of Y_i, and l'_k and u'_k be the lower bound and the upper bound of Y'_k. Thus, $\hat{f}_i = \frac{1}{u_i - l_i}$ and $\hat{f}'_k = \frac{1}{u'_k - l'_k}$. To calculate the probability, we use variable transformation from $\{Y_1, \ldots, Y_n, Y'_1, \ldots, Y'_m\}$ to $\{X_1, \ldots, X_n\} \cup \{X'_1, \ldots, X'_m\}$, because the time constraint is defined on the latter set of variables. Let y_j and y'_k be the value of the variable Y_j and Y'_k. We have $Y_j = X_j - X_i$ according to the formula (7.2), $Y'_k = X'_k - X_i$ according to the formula (7.3) and $X_0 = 0$. The density function $\hat{f}(x_1, \ldots, x_n, x'_1, \ldots, x'_m) = \hat{f}(y_1, \ldots, y_n, y'_1, \ldots, y'_m)|J|$.

$$J = \begin{pmatrix} \partial y_1/\partial x_1 & \cdots & \partial y_1/\partial x_n & \partial y_1/\partial x'_1 & \cdots & \partial y_1/\partial x'_m \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial y_n/\partial x_1 & \cdots & \partial y_n/\partial x_n & \partial y_n/\partial x'_1 & \cdots & \partial y_n/\partial x'_m \\ \partial y'_1/\partial x_1 & \cdots & \partial y'_1/\partial x_n & \partial y'_1/\partial x'_1 & \cdots & \partial y'_1/\partial x'_m \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial y'_m/\partial x_1 & \cdots & \partial y'_m/\partial x_n & \partial y'_m/\partial x'_1 & \cdots & \partial y'_m/\partial x'_m \end{pmatrix}$$

Since $\partial y_j/\partial x_j = 1$, $\partial y_j/\partial x_i = 0$ for every $i > j$, $\partial y_j/\partial x'_k = 0$ for every $1 \leq k \leq m$, $\partial y'_k/\partial x'_k = 1$, and $\partial y'_k/\partial x'_i = 0$ for every $i > k$, J is a lower triangular square matrix and every diagonal element is 1. Thus the determinant of Jacobian matrix is 1.

$$\hat{f}(x_1,\ldots,x_n,x'_1,\ldots,x'_m) = \hat{f}(y_1,\ldots,y_n,y'_1,\ldots,y'_m) = \left(\prod_{i=1}^{n}\hat{f}_i\right)\cdot\left(\prod_{k=1}^{m}\hat{f}'_k\right).$$

Now we calculate the probability over $X_1,\ldots,X_n,X'_1,\ldots,X'_m$.

The formulae (7.1)-(7.4) characterize the constraint for a path. The constraint defines a region B in the $(n+m)$-dimensional space. Let $P[\rho] = P[\alpha_1\alpha_2\ldots\alpha_n]$ be the probability of the path ρ. $P[\rho]$ is calculated by

$$P[\rho] = \int\cdots\int_{(X_1,\ldots,X_n,X'_1,\ldots,X'_m)\in B} \hat{f}_1\cdots\hat{f}_n\hat{f}'_1\cdots\hat{f}'_m \,\mathrm{d}x'_m\cdots\mathrm{d}x'_1\mathrm{d}x_n\cdots\mathrm{d}x_1. \quad (7.5)$$

The formula is calculated from the innermost integral to the outermost integral. Let $Z = \{X_1,\ldots,X_n,X'_1,\ldots,X'_m\}$. The integral range of variable $z_j \in Z$ might depend on the range of $z_i \in Z$ for $i < j$. However, a technical difficulty exists in the above formula: the dependency relation is different in the different parts of range of z_i. In our example,

$$P[ag] = \int_1^5\left(\int_{max_1}^{6}\left(\int_{max_2}^{7}\Big(\int_{max_3}^{5}\big(\int_{max_4}^{x_a+4}\frac{1}{576}\,\mathrm{d}x_c\big)\mathrm{d}x_b\Big)\mathrm{d}x_h\right)\mathrm{d}x_g\right)\mathrm{d}x_a,$$

where $max_1 = \max\{2,x_a\}$, $max_2 = \max\{3,x_g\}$, $max_3 = \max\{2,x_a\}$, $max_4 = \max\{x_a+1,x_g\}$ and $\frac{1}{576} = \frac{1}{4}\times\frac{1}{3}\times\frac{1}{4}\times\frac{1}{4}\times\frac{1}{3}$. These max functions are deduced from the constraint of the path $\langle ag\rangle$. For example, max_1 comes from the conjunction of two expressions $2 \leq x_g$ and $x_a < x_g$ in the constraint. At different parts of region, the functions obtain different values, which means that the above formula cannot be computed directly.

Therefore, the region needs to be split into a set of disjoint blocks, in each of which the dependence relation is fixed. Then the integration is performed over every block and the results are summed up. The region-splitting can be done using the Fourier-Motzkin elimination method [Sch98]. The constraint of path $\langle ag \rangle$ is split into eight disjoint regions[2]:

$R_1 = 1 \leq x_a < 2 \wedge 2 \leq x_g < 1+x_a \wedge 3 \leq x_h \leq 7 \wedge 2 \leq x_b \leq 5 \wedge 1+x_a \leq x_c \leq 4+x_a$

$R_2 = 1 \leq x_a < 2 \wedge 1+x_a \leq x_g < 3 \wedge 3 \leq x_h \leq 7 \wedge 2 \leq x_b \leq 5 \wedge x_g < x_c \leq 4+x_a$

$R_3 = 1 \leq x_a < 2 \wedge 3 \leq x_g < 4+x_a \wedge x_g < x_h \leq 7 \wedge 2 \leq x_b \leq 5 \wedge x_g < x_c \leq 4+x_a$

$R_4 = 2 < x_a < 3 \wedge x_a < x_g < 3 \wedge 3 \leq x_h \leq 7 \wedge x_a < x_b \leq 5 \wedge 1+x_a \leq x_c \leq 4+x_a$

$R_5 = 2 < x_a < 3 \wedge 3 \leq x_g < 1+x_a \wedge x_g < x_h \leq 7 \wedge x_a < x_b \leq 5 \wedge 1+x_a \leq x_c \leq 4+x_a$

$R_6 = 2 < x_a < 3 \wedge 1+x_a \leq x_g \leq 6 \wedge x_g < x_h \leq 7 \wedge x_a < x_b \leq 5 \wedge x_g < x_c \leq 4+x_a$

$R_7 = 3 \leq x_a < 5 \wedge x_a < x_g < 1+x_a \wedge x_g < x_h \leq 7 \wedge x_a < x_b \leq 5 \wedge 1+x_a \leq x_c \leq 4+x_a$

$R_8 = 3 \leq x_a < 5 \wedge 1+x_a \leq x_g \leq 6 \wedge x_g < x_h \leq 7 \wedge x_a < x_b \leq 5 \wedge x_g < x_c \leq 4+x_a.$

Each region describes the integral range for every variable and an integral of the joint distribution function $\frac{1}{576}$ is defined over it. The following are eight integrals corresponding to the eight regions:

$$P[R_1] = \int_1^2 \left(\int_2^{x_a+1} \left(\int_3^7 \left(\int_2^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_2] = \int_1^2 \left(\int_{x_a+1}^3 \left(\int_3^7 \left(\int_2^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_3] = \int_1^2 \left(\int_3^{x_a+4} \left(\int_{x_g}^7 \left(\int_2^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_4] = \int_2^3 \left(\int_{x_a}^3 \left(\int_3^7 \left(\int_{x_a}^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_5] = \int_2^3 \left(\int_3^{x_a+1} \left(\int_{x_g}^7 \left(\int_{x_a}^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

[2]These regions can be verified by feeding the time constraint into Mathematica.

$$P[R_6] = \int_2^3 \left(\int_{x_a+1}^6 \left(\int_{x_g}^7 \left(\int_{x_a}^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_7] = \int_3^5 \left(\int_{x_a}^{x_a+1} \left(\int_{x_g}^7 \left(\int_{x_a}^5 \left(\int_{x_a+1}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a$$

$$P[R_8] = \int_3^5 \left(\int_{x_a+1}^6 \left(\int_{x_g}^7 \left(\int_{x_a}^5 \left(\int_{x_g}^{x_a+4} \frac{1}{576} \mathrm{d}x_c \right) \mathrm{d}x_b \right) \mathrm{d}x_h \right) \mathrm{d}x_g \right) \mathrm{d}x_a.$$

The above integrals were calculated using Mathematica and $P[ag]$ is the sum of results of these integrals, $\frac{1489}{5760}$.

7.2.3 The complexity of the computation

The computation of the integration over the range defined by a block has linear complexity, because there are no cyclic references, the density functions do not contain integration variables and all of integration bounds are linear. The general complexity of the Fourier-Motzkin (FM) elimination method is $O((\frac{M}{2})^{2^N})$, where M is the number of inequalities and N is the number of variables. In the constraint of a path, any linear inequality has at most two variables and every coefficient belongs to the set $\{-1, 0, 1\}$. Now we give a brief estimation of the complexity of the FM method for this special case. The FM method has N recursive steps, each of which deals with one variable. Each step generates some new inequalities. The number of new inequalities is maximal when $\frac{M}{2}$ inequalities have a positive coefficient of the variable and other $\frac{M}{2}$ inequalities have a negative coefficient. The maximal number is $\frac{M^2}{4}$. After N steps, we gain the general complexity. Since we have $(n + m)$ transitions and at most three inequalities for each transition (which can be easily deduced from the formulae (7.1)-(7.4)), and there are at most two variables per inequality, there are at most $6(n + m)$

coefficients in all the inequalities. To produce the maximal number of new inequalities, each variable can appear in 6 inequalities. Three of them have a positive coefficient and the other three have a negative coefficient. Therefore, we obtain $(3 \times 3)^{n+m}$ new inequalities in the end. Then the complexity is $O(9^{n+m})$ in our case. Each new inequality means a region is split, which means we may get an exponentially increased number of integral regions. However, many of them are redundant. In practice, we expect only a small number of split regions. For example, for path $\langle ag \rangle$ we obtain 8 regions, far less than 9^5.

On the other hand, the calculation of the formula (7.5) can be seen as the calculation of the volume of the region defined by the constraint because the joint density function is a constant. Since the constraints contain only conjunction and linear inequalities, the region is convex linear. The computational complexity of computing the volume of a convex body defined by linear inequalities is $\#P$-hard [DF88]. But approximation algorithms for computing volume can be polynomial. The fastest algorithm is $O^*(n^4)$ [LV03], where n is the dimension of the body. We also expect to find (but have not found yet) some helpful results for our special case to accelerate computing volumes.

7.2.4 Simplification for exponential distribution

We have already mentioned in the introduction to this chapter that the calculation could be simple if the delay of each transition obeys exponential distribution, since the system is Markovian. Now we give a short description of how exponential distribution simplifies calculation. An exponential

distribution is defined on interval $[0, \infty)$ with rate λ and density function $\lambda \cdot e^{-\lambda \cdot y}$. At first, the formula (7.2) is changed to $0 \leq x_j - x_i < \infty$. Similarly, the formula (7.3) is simplified to $0 \leq x_{\alpha'} - x_i < \infty$. For example, consider all transitions in Figure 7.1 are following exponential distribution. The density functions for transitions a, b, c, d, g and h are $\lambda_a \cdot e^{-\lambda_a \cdot y_a}$, $\lambda_b \cdot e^{-\lambda_b \cdot y_b}$, $\lambda_c \cdot e^{-\lambda_c \cdot y_c}$, $\lambda_d \cdot e^{-\lambda_d \cdot y_d}$, $\lambda_g \cdot e^{-\lambda_g \cdot y_g}$ and $\lambda_h \cdot e^{-\lambda_h \cdot y_h}$. The constraint for path ag is ($x_a = x_1$ and $x_g = x_2$)

$$(0 \leq x_a < \infty) \wedge (0 \leq x_g < \infty) \wedge (0 \leq x_h < \infty) \wedge (0 \leq x_b < \infty)$$
$$\wedge (0 \leq x_c - x_a < \infty) \wedge (x_a < x_b) \wedge (x_a < x_g) \wedge (x_g < x_h) \wedge (x_g < x_c).$$

Each transition β in the time constraint has a subconstraint of the form

$$(0 \leq x_\beta - x_\alpha < \infty) \wedge (x_\gamma < x_\beta), \tag{7.6}$$

where (1) x_γ is the time point at which the $(j-1)$th transition in the path is triggered if β is the jth transition ($x_\gamma = x_0$ if β is the first transition), or at which β is disabled if β does not appear in the path; (2) x_α is the time point at which β becomes enabled. Note that this constraint need not be split into smaller integral blocks.

The integration over this constraint is done inductively as follows. For a transition β that does not appear in the path, its lower integral bound is $x_\gamma - x_\alpha$ (according to formula (7.6)) since its density function $\lambda_\beta \cdot e^{-\lambda_\beta \cdot y_\beta}$ is defined over $[0, \infty)$ by defining $y_\beta = x_\beta - x_\alpha$. The integration for β is

$$\int_{x_\gamma - x_\alpha}^{\infty} \lambda_\beta \cdot e^{-\lambda_\beta \cdot y_\beta} \mathrm{d}y_\beta = e^{-\lambda_\beta \cdot (x_\gamma - x_\alpha)}. \tag{7.7}$$

For a path with n transitions, we first integrate over the transitions not occurring in the path according to formula (7.7) and multiply their results.

We obtain the following result

$$e^{(\Sigma'_1-\Sigma''_1)\cdot x_1+\ldots+(\Sigma'_n-\Sigma''_n)\cdot x_n},$$

where Σ'_i is the sum of rates whose corresponding transitions are enabled at x_i, and Σ''_i the sum of rates corresponding to transitions that are disabled at x_i. Note that Σ'_n is 0.

After calculating the integration for the transitions not triggered, we need to calculate the integration for the transitions in the path. Let α_j be the jth transition with density function $\lambda_j \cdot e^{-\lambda_j \cdot y_j}$. We calculate the integration from α_n to α_1 recursively. It is easy to prove by induction that after we calculate the integration over y_{j+1}, the integration over y_j has the form of

$$e^{-\Sigma_j\cdot x_i}\cdot\int_{x_{j-1}-x_i}^{\infty}\lambda_j\cdot e^{-(\lambda_j+\Sigma_j)\cdot y_j}\mathrm{d}y_j=\frac{\lambda_j}{\lambda_j+\Sigma_j}\cdot e^{-(\lambda_j+\Sigma_j)\cdot x_{j-1}}\cdot e^{\lambda_j\cdot x_i},$$

where Σ_j is the sum of rates of transitions that are enabled between x_{j-1} and x_j.

Proof. For transition α_n, we have $e^{-\Sigma''_n\cdot x_n}=e^{-\Sigma''_n\cdot(y_n+x_l)}$, where x_l is the time point at which α_n becomes enabled, and thus

$$e^{-\Sigma''_n\cdot x_l}\cdot\int_{x_{n-1}-x_l}^{\infty}\lambda_j\cdot e^{-(\lambda_n+\Sigma''_n)\cdot y_n}\mathrm{d}y_n=\frac{\lambda_n}{\lambda_n+\Sigma''_n}\cdot e^{-(\lambda_n+\Sigma''_n)\cdot x_{n-1}}\cdot e^{\lambda_n\cdot x_l}.$$

Assume that, for transition α_{j+1}, we have

$$\frac{\lambda_{j+1}}{\lambda_{j+1}+\Sigma_{j+1}}\cdot e^{-(\lambda_{j+1}+\Sigma_{j+1})\cdot x_j}\cdot e^{\lambda_{j+1}\cdot x_k}.$$

For transition α_j, we have $e^{(\Sigma'_j-\Sigma''_j)\cdot x_j}$ and $e^{-(\lambda_{j+1}+\Sigma_{j+1})\cdot x_j}$. Let

$$\Sigma_j=\begin{cases}\Sigma''_j+\lambda_{j+1}+\Sigma_{j+1}-\Sigma'_j & x_k\neq x_j,\\ \Sigma''_j+\Sigma_{j+1}-\Sigma'_j & x_k=x_j.\end{cases}$$

For any transition α which is first enabled at x_j, its rate is included in Σ'_j. Since α is enabled between x_j and x_{j+1}, its rate is in Σ_{j+1} as well. Thus, Σ_j only contains the rates of transitions that are enabled between x_{j-1} and x_j.

Finally, the probability is calculated as

$$\prod_{j=1}^{n} \frac{\lambda_j}{\lambda_j + \Sigma_j}, \tag{7.8}$$

where Σ_j is the sum of rates of transitions that are enabled between x_{j-1} and x_j. For example, the probability of path *ag* is calculated as follows:

$$\frac{\lambda_g}{\lambda_g + \lambda_h + \lambda_c} \cdot \frac{\lambda_a}{\lambda_a + \lambda_b + \lambda_g + \lambda_h}.$$

The formula (7.8) shows that, for a system where all transition delays obey exponential distribution, we do not need calculate the integration for the probability of a path, and instead calculate the probability directly over the rates.

7.3 The probability of a partial order

A partial order represents a group of equivalent paths that have the same essential partial order. Given a path and an independence relation between its transitions (representing pairs of transitions that can concurrently overlap), one can generate the partial order relation $\rightarrow_\rho^*$ (see Section 2.3). A simple way of calculating the probability of executing a partial order is to sum up the probabilities of all equivalent paths; however, this calculation can be optimized. For example, consider the system in Section 7.1.3. The partial order containing a and g, with no order relation between them, represents two paths: $\langle ag \rangle$ and $\langle ga \rangle$. The constraint for the path $\langle ga \rangle$ is

$$(2 \leq x_g \leq 6) \wedge (1 \leq x_a \leq 5) \wedge (x_g < x_a) \wedge$$
$$(2 \leq x_b \leq 5) \wedge (3 \leq x_h \leq 7) \wedge (x_a < x_b) \wedge (x_a \leq x_h).$$

It is split into two integral regions and hence there are ten integrals to calculate in order to obtain the probability of the partial order[3]. In this partial order, a could be triggered earlier or later than g and thus we do not have the relation $x_a < x_g$ or $x_a > x_g$. The key constraint for the partial order is

$$(x_a < x_b) \wedge (x_a < x_h) \wedge (x_g < x_h) \wedge (x_g < x_c).$$

The conjunction of this constraint and the basic constraint

$$(1 \leq x_a \leq 5) \wedge (2 \leq x_g \leq 6) \wedge (3 \leq x_h \leq 7) \wedge (2 \leq x_b \leq 5) \wedge (x_a + 1 \leq x_c \leq x_a + 4)$$

is split into nine blocks. Thus, it is possible to give heuristics for optimizing the calculation of the probability of a partial order. Note that it is also possible to merge two adjacent blocks into one when they are generated from different paths. For example, there are two such blocks when we split the constraints for paths $\langle ag \rangle$ and $\langle ga \rangle$ separately. Generating all possible blocks first and merging them later wastes resources. It is more efficient to generate the merged block directly from a simple constraint.

The key idea for optimizing probability calculation for a partial order is that we remove from the time constraint of the partial ordering relations between any pair of transitions that can be fired concurrently. Hence the time constraint of the partial order only describes the necessary relations that guarantee the partial order. For instance, the relations $x_a < x_g$ and $x_g < x_a$ between transitions a and g in the above example are not necessary for the partial order a, g since either a or g can be triggered first. The time constraint for a partial order cannot be constructed simply as the disjunction of time constraints for all linearizations of the partial order with

[3]The FM method cannot handle disjunction of two sets of inequalities.

removing the unnecessary relations among concurrent transitions. The reason is that other relations in a time constraint of a linearization may depend on the ordering relations among concurrent transitions on this linearization. Optimization can be difficult when there are synchronizations among transitions. The following subsection emphasizes this difficulty.

7.3.1 The synchronization behind partial order

In the model defined in Chapter 5, there exist two kinds of synchronizations. One kind of synchronization occurs among transitions which compete with each other to access a shared variable. When one transition is granted access, other transitions are disabled through synchronization. But no synchronization can occur if a process requests access after the shared variable has been acquired by some other transition since the transition requesting the shared variable cannot be enabled. The other kind is the synchronization between transitions interacting through communication channels.

The synchronization on access of a shared variable can occur in some but not all paths that a partial order represents. For example, let us consider the partial order in Figure 7.2. Each node represents a transition. Transitions

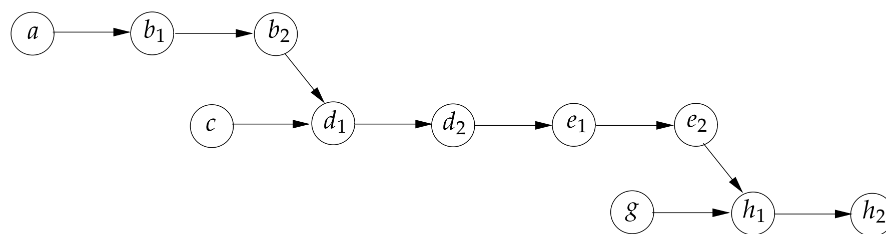

Figure 7.2: Two partial order examples

a, b_1, b_2 belong to Process 1, c, d_1, d_2, e_1, e_2 belong to Process 2 and g, h_1, h_2

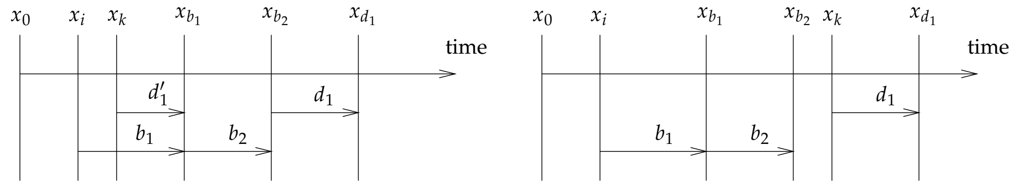

Figure 7.3: Synchronization (left) and non-synchronization (right) scenarios

belong to Process 3. Transitions b_1, d_1, e_1, h_1 request the same shared variable and b_2, d_2, e_2, h_2 release the shared variable. The edge from b_2 to d_1 represents that b_2 must be executed earlier than d_1. The left part of Figure 7.3 represents the scenario that d'_1, the first occurrence of d_1, starts the request before b_1 is executed. Here x_k is the time point when d'_1 becomes enabled and x_{b_1} is the one when it is disabled, i.e., when b_1 is triggered. A synchronization between b_1 and d'_1 occurs and causes d'_1 to be disabled by b_1. Later, the second occurrence of d_1 is enabled by b_2 since b_2 releases the shared variable. On the right part of Figure 7.3, no synchronization occurs because the first occurrence of d_1 starts request after b_2 is executed.

Now we describe the constraint for the synchronization through which d'_1 is disabled by b_1. Let γ be such a transition that (1) it references the same shared variable as d_1 and b_1 do; (2) it is triggered earlier than b_1; (3) there are no other transitions which reference the same shared variable and are triggered between the execution of γ and b_1. Let x_γ be the time point when γ is triggered. d'_1 must be first enabled at or after x_γ (otherwise, d'_1 is disabled by γ). If γ does not exist, x_γ is equal to x_0 and thus is omitted from the formula below. The left part of Figure 7.4 illustrates this scenario. The necessary time constraint is as follows:

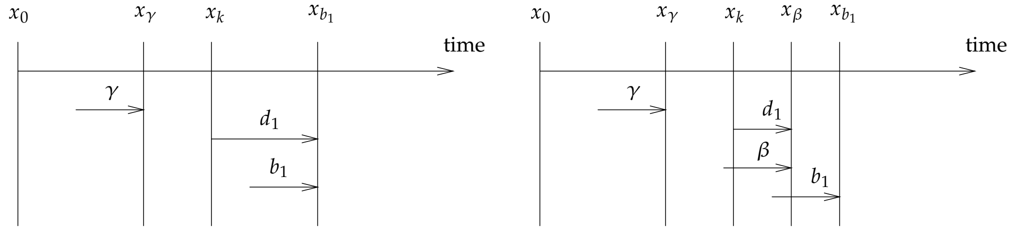

Figure 7.4: d'_1 is disabled by b_1 (left) and d'_1 is disabled by β (right)

$$(l_{d_1} < x_{d'_1} - x_k < u_{d_1}) \wedge (x_{d'_1} > x_{b_1} > x_k \geq x_\gamma).$$

On the right part of Figure 7.3, no synchronization occurs and we need to record the relation $x_k > x_{b_1}$.

In addition, when a transition β in P_1 is triggered earlier than b_1, as shown on the right part of Figure 7.4, d'_1 is disabled by β because P_1 is forced to leave the source location[4] of d_1 by the execution of β. Hence, we obtain the following condition if d'_1 is disabled by β:

$$(l_{d_1} < x_{d'_1} - x_k < u_{d_1}) \wedge (x_{d'_1} > x_\beta \wedge x_{b_1} > x_\beta > x_k \geq x_\gamma).$$

The synchronization could occur between b_1 and d_1, b_1 and h_1, d_1 and h_1 and e_1 and h_1 in Figure 7.2. The number of combinations increases exponentially with the number of possible synchronizations, and the time constraint of each combination contains different number of variables. To optimize the calculation of probability, we must deal with each combination separately. One way to perform optimization of a given partial order is to generate all combinations and optimize the calculation for each combination. But generating combinations is time-consuming and defeats the object

[4]If the source location and the target location of β are the same, d_1 is disabled first and then enabled again.

of optimization. In addition, some combinations may be prohibited by time constraints; it is a waste of time to generate them.

A synchronized communication transition not appearing in the partial order could be enabled during the execution of some paths represented by the partial order. Figure 7.5 depicts a system consisting of two processes. The two actions labeled as c represent a synchronized communication tran-

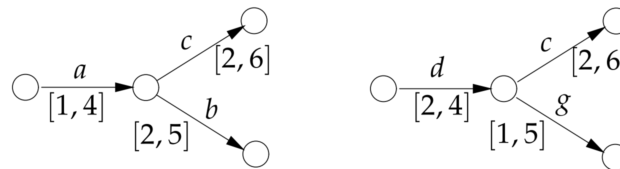

Figure 7.5: Two system examples

sition. The partial order is $a \rightarrow b, d \rightarrow g$. The possible paths represented by the partial order are $\langle adbg \rangle$, $\langle adgb \rangle$, $\langle dabg \rangle$, $\langle dagb \rangle$, $\langle abdg \rangle$ and $\langle dgab \rangle$. On the first four paths, c is enabled but not triggered, while on the last two paths, c is never enabled. Thus we should separate the calculation for the first four paths from that for the last two. If the transition c also depends on a transition in another process (which is not shown in the figure), e.g., they reference the same shared variable, the situation becomes more complicated.

7.3.2 The general algorithm

Due to the complex situation inside partial orders as presented in the previous section, optimization could be done with high cost such that the benefit is offset by extra computation, particularly under existence of shared

variables. Therefore, we do not perform general optimization on probability calculation of a partial order in this thesis. Instead, we use the simple way to do the calculation for general cases: calculate the probability of each equivalent path first and then sum them up. Optimization can be added for a special case, e.g., for synchronized communication transitions.

In Section 7.2, we use countdown clocks to deduce the timing relations along a path. Indeed, we do not need to know their values. We only need to record the time points when each transition becomes enabled and fired (or disabled). Therefore, the key idea in this algorithm is to collect these time points in terms of formula (7.1)-(7.4). Because a transition in transition systems is separated to two transitions (internal transitions are used for synchronization), one for the enabling condition and the other for the transformation, in extended timed automata after translation, collecting time points can be done on DAGs. We adopt forward search and symbolic execution, rather than the backward search used in Chapter 6. Symbolic execution on a given initial condition has been widely studied, such as [DPCPG04] and [Kin76]. Symbolic execution of a transition along a path in extended timed automata is described as follows (for simplicity, we neglect the time constraints and program control in the following exposition).

The symbolic execution of a transition of a DAG

A transition e has an enabling condition c_e on program variables and a transformation on program variables as well. Let C_e be the accumulated condition on which e is executed. Let V_e be the evaluation of program variables before e is executed. The value of a variable could be symbolic. C_e and V_e are updated from the initial condition C_0 and the initial evaluation

V_0 along the path by the execution of transitions before e. A value in V_e is the expression of initial values in V_0 and a predicate in C_e is also a Boolean combination of values in V_0.

Under the pair of $\langle C_e, V_e \rangle$, if c_e is evaluated to *false*, e cannot be executed; if c_e is evaluated to *true*, the accumulated condition after e is executed is not changed; if c_e is neither evaluated to *false* nor *true*, the accumulated condition is updated to the conjunction of C_e and the value of c_e after evaluation. After e is executed, the evaluation of the program variables is updated by the transformation: for every variable v which is assigned to a new value $expr$ in the transformation, its value in the evaluation of the program variables is replaced by $expr$. Of course, the expression $expr$ need be expressed by the combination of the initial values in V_0.

The algorithm to compute the probability of a partial order

1. Label all of initial nodes as *old* and all other nodes as *new*. For each initial node:

   (a) Attach an initial condition (represented as a set of predicates) and an initial DBM (see Section 6.3) to it. The initial DBM is as follows.

   $$\begin{pmatrix} (0, \leq) & \cdots & (0, \leq) \\ \vdots & \vdots & \vdots \\ (0, \leq) & \cdots & (0, \leq) \end{pmatrix}$$

   The initial DBM represents the fact that the reading of every clock is 0 at the time when the system starts.

   (b) Find all outgoing edges from the automata and test their enabledness by the predicates and their enabling conditions. For each

enabled edge, record its enabledness time point as x_0 which is 0.

2. Choose a node a which is labeled as *new* and all of its predecessors labeled as *old*. Label this node as *old*. Let $b_1, \ldots, b_n$ $(n \geq 1)$ be its predecessors. In addition, let τ_k $(1 \leq k \leq n)$ be the transition from b_k to a. For each edge τ_k which is enabled at b_k, do the following:

   (a) Let τ_k be the jth transition from initial states. x_i, the time point when it is enabled, can be found from the information recorded at b_k. l_k and u_k are the time bounds of τ_k. Update the predicates at b_k by the enabling condition and the transformation of τ_k through symbolic execution. Let φ be the DBM at b_k, $I(a)^X$ the assertion on clocks in the location invariant of a, ψ^X the assertion on clocks of the edge τ_k, and λ the set of reset clocks of τ_k. Calculate the DBM from the one at b_k according to the formula defined in [Alu98, Yov98]:

   $$DBM = (((\varphi \wedge I(a)^X) \Uparrow) \wedge I(a)^X \wedge \psi^X)[\lambda := 0], \tag{7.9}$$

   where $\wedge$ is the intersection of two DBMs, $\Uparrow$ is the time elapse operation on DBMs and $[\lambda := 0]$ is the reset operation on DBMs. If both the new predicates and the new DBM are satisfiable, record $x_{j-1} < x_j$ and $l_k \leq x_j - x_i < u_k$ in a.

   (b) For each of enabled edges at b_k except τ_k, if it belongs to the same process that τ_k belongs to, the edge is disabled at a because the control leaves its source location; if it belongs to another process, test its enabledness by the new predicates calculated in step (b).

For each disabled edge τ'_k by τ_k, let l'_k and u'_k be the time bounds of τ'_k and x_i be its enabled time point (which is recorded at b_k). Record $l'_k \leq x'_k - x_i < u'_k$ and $x_j < x'_k$ at a.

(c) Find all of outgoing edges at a from the automata, test their enabledness by the predicates and the enabling condition of the edges. Record the information of all of enabled edges: If an enabled edge is enabled before a (which is recorded at b_k), copy its enabled time point from b_k; otherwise, it is first enabled at a and record it enabled time point as x_j.

3. Repeat step (2) until no node is labeled as *new*. Note that, for a leaf node, we do not need the information of a newly enabled edge at that node, but all transitions that are enabled before reaching the leaf node are supposed to be disabled by the end of a path. Now we have gathered all of time constraints defined by the formulae (7.1)-(7.4) for every path. The final probability is the sum of the probabilities of all paths.

In the above algorithm, testing the enabledness of a transition is similar to symbolic execution of a transition without updating the evaluation of program variables. But when the enabledness of a transition cannot be decided, we need to generate two new accumulated conditions. One represents the transition is disabled and the other represents the transition is enabled. Intersection of two DBMs in forward search is the same as that in backward search in Section 6.3.2. Time elapse on a $k \times k$ DBM D is calculated by setting the entry $D_{i,0}$ to ∞ for each $1 \leq i \leq k$. Reseting the set λ of clocks on D in forward search is done as follows: (1) for $x_i \in \lambda$, set $D_{i,0}$ and

$D_{0,j}$ to $(0, \leq)$; (2) for $x_i, x_j \in \lambda$, set $D_{i,j}$ to $(0, \leq)$; (3) for $x_i \in \lambda$ and $x_j \notin \lambda$, set $D_{i,j}$ to $D_{0,j}$ and $D_{j,i}$ to $D_{j,0}$. The resulting DBMs of these three operations, intersection, time elapse and reset, need be checked for satisfiability and canonicalized if they are satisfiable.

7.3.3 Optimized algorithm for communication transitions

For a system without shared variables, each transition can be locally decided except synchronized communication transitions, which is exactly decided by two participant processes.

The time relation defined by formula (7.1)-(7.4) gives time constraint for a linear sequence. To allow maximum concurrency, we need to relax the relation so that it can describe a partial order. Indeed, if two transitions do not depend on each other, i.e., there is no edge in the partial order connect them, either one can be fired earlier than the other. Thus we remove all inequalities between their execution time points. We need to modify formula (7.1) and (7.2) to reflect this change. For transitions α and β such that there is an edge in the partial order starting at α and pointing to β, we obtain the following formula

$$l_\beta < x_\beta - x_\alpha < u_\beta.$$

In the general case in which β has n predecessors $\alpha_1, \ldots, \alpha_n$,

$$l_\beta < x_\beta - \max\{x_{\alpha_1}, \ldots, x_{\alpha_n}\} < u_\beta. \qquad (7.10)$$

Particularly, if $n = 0$, we have $l_\beta < x_\beta < u_\beta$. The formula (7.10) for every transition can be gained statically from the partial order.

Let ρ_i and ρ_j be projected paths of the partial order on Process i and j. Let x_{n_i} (x_{n_j}) be the end time point on ρ_i (ρ_j) when the last transition on ρ_i (ρ_j) is triggered. For any transition α' belonging to Process i and remaining enabled after x_{n_i} (including the newly enabled transitions at x_{n_i}), we have the following relation in order to ensure α' cannot be triggered earlier than all transitions appearing on ρ_j:

$$(x_{\alpha'} > x_{n_i}) \wedge (x_{\alpha'} > x_{n_j}), \tag{7.11}$$

where $x_{\alpha'}$ is the same time point as the one in formula (7.4).

For Figure 7.5 and the partial order $a \rightarrow b, d \rightarrow g$, we now describe the time constraint for the joint transition c. Let x_a, x_b, x_c, x_d and x_g be the time points of a, b, c, d and g respectively. On the paths $abdg$ and $dgab$, c is not enabled because during the execution of these paths, the system never reaches a state containing the two source location of c in both processes. The key time constraint to describe this situation is

$$(x_a > x_g) \vee (x_d > x_b). \tag{7.12}$$

On paths $adbg, adgb, dabg$ and $dagb$, c is first enabled at time point $\max\{x_a, x_d\}$ and disabled at $\min\{x_b, x_g\}$. The key time constraint is

$$(x_a < x_g \wedge x_d < x_b) \wedge (x_c > \min\{x_b, x_g\} \wedge l_c < x_c - \max\{x_a, x_d\} < u_c), \tag{7.13}$$

where l_c, u_c are the lower and the upper bound of c. However, since

$$(x_a > x_g \vee x_d > x_b) \wedge (x_c > \min\{x_b, x_g\} \wedge l_c < x_c - \max\{x_a, x_d\} < u_c)$$

is simplified to

$$(x_a > x_g \vee x_d > x_b) \wedge (l_c < x_c - \max\{x_a, x_d\} < u_c), \tag{7.14}$$

the integration on x_c is 1 when we integrate on the above formula, which means the integration result on formula (7.14) is the same as the result on formula (7.12). Thus, from the integration point of view, we can use the following formula to describe the constraint for c, no matter c is enabled or not:

$$(x_c > \min\{x_b, x_g\}) \wedge (l_c < x_c - \max\{x_a, x_d\} < u_c). \tag{7.15}$$

Note that x_a and x_d could be the x_0 and x_b, and x_g could be the end time point on other processes if both x_a and x_d are the end time points on the projected paths they belong to respectively.

Now formulae (7.3), (7.4), (7.10), (7.11) and (7.15) define the time constraint for partial orders with synchronized communication transitions. The optimization effect is significant when the partial order allows much concurrency. For example, for the system in Figure 7.5 and the partial order $a \to c, d \to c$, there are two equivalent paths $\langle adc \rangle$ and $\langle dac \rangle$. The number of regions of the path $\langle adc \rangle$ after split is 2 and the number for the path $\langle dac \rangle$ is 1. After optimization being applied, the number of regions is 3, which equals to the sum of regions of the two paths. However, for the partial order $a \to b, d \to g$, the sum of regions of the six paths is 38, while the number after optimization is 19, only a half of the sum. The optimized algorithm for synchronized communication transitions is described as follows:

1. Label all of initial nodes as *old* and all other nodes as *new*. For each initial node:

   (a) Attach an initial condition and an initial DBM to it.

   (b) Find all of outgoing edges and test their enabledness. Record all

of enabled edges and their enabledness time points as x_0.

2. Choose a node a which is labeled as *new* and all of whose predecessors are labeled as *old*. Label this node as *old*. Let $b_1, \ldots, b_n$ $(n \geq 1)$ be its predecessors. For each enabled edge τ_k starting at b_k and pointing to a, do the following:

   (a) Let τ_k be the jth transition from initial states. x_i is the time point when it is enabled and l_k and u_k are the time bounds of τ_k. Update the predicates at b_k through symbolic execution, and calculate the DBM from the one at b_k according to the formula (7.9). If both the new predicates and the new DBM are satisfiable, find the time points $x_{\beta_1}, \ldots, x_{\beta_m}$ of all of its predecessors in the partial order. Record $l_k < x_j - \max\{x_{\beta_1}, \ldots, x_{\beta_m}\} < u_k$.

   (b) For each disabled non-synchronized edge τ'_k by τ_k, let l'_k and u'_k be the time bounds of τ'_k and x_i be its enabled time point. Record $l'_k \leq x'_k - x_i < u'_k$ and $x_j < x'_k$ at a. For each disabled synchronized edge τ'_k by τ_k, we can find the time points when it becomes enabled in both processes such as x_a and x_d in formula (7.15) and the successive time points x_b and x_g when it is disabled in both processes from the partial order. Record $x'_k > \min\{x_b, x_g\} \wedge l'_k < x'_k - \max\{x_a, x_d\} < u'_k$.

   (c) Find all of outgoing edges at a and test their enabledness. Record the information of all of enabled edges of them: If an enabled edge is enabled before a, copy its enabled time point from b_k; otherwise, it is first enabled at a and record it enabled time point

as x_j.

3. Repeat step (2) until no node is labeled as *new*. For each projected path ρ_i with γ_i as its last transition, find the transitions that are remaining enabled after γ_i is triggered. Let β_j^k be such a transition on ρ_k. Add $x_{\gamma_i} < x_{\beta_j^k}$ $(i \neq k)$ to time constraints, where x_{γ_i} and $x_{\beta_j^k}$ are time points for γ_i and β_j^k respectively.

4. After gathering the information for each feasible path which is not prohibited by time constraints, we have the knowledge of all synchronized communication transitions that have been enabled but not executed. Each feasible path has the same time constraint, except that a synchronized communication transition may be enabled on some paths, but disabled on other paths. In either case, we can conjunct the formula (7.15) to the time constraint. Thus we conjunct all such formulae to the time constraint and use it to calculate the probability of the partial order.

7.4 Summary

For a real-time system, where each transition must be enabled for a bounded period before being triggered, we proposed a methodology to compute the probability of an execution. Our methodology is suitable for general continuous probability distributions and it gives the accurate value of the probability. At first, a time constraint of the execution is collected and then the probability is calculated using integration over the region defined by the

constraint. The calculation has been generalized to a partial order as well. When a system only uses synchronized communication transitions and no shared variables, the probability computation for a partial order can be optimized. The general algorithm for calculating the probability of a partial order was implemented in RPET, but the optimized algorithm has not been implemented yet.

Although we use uniform distribution in the example to demonstrate our methodology, the general formula for computing the probabilities holds for arbitrary distributions, as shown for exponential distribution, because of the fact that the time constraint defined by the formulae (7.1)-(7.4) and the Jacobian for the random variable transformation are independent of probability distributions. A potential problem for any other distribution than uniform distribution and exponential distribution is that its density function contains the integration variable so that the calculation of an $(n+m)$-fold multiple integral on this kind of distributions could be much harder than the calculation on uniform distribution.

We also showed that our methodology can be optimized for a partial order. However, due to the complexity introduced by joint transitions, the effect of optimization can be counteracted by extra computation for handling this complexity, in particular, when there are many occurrences of joint transitions involved in the probability calculation.

Chapter 8

Real-time Path Exploration Tool

We have enhanced the untimed version of PET to provide code transformation to implement the methodology in Chapter 4 and extended it to work on real-time systems with respect to the methodologies in Chapter 5, 6 and 7. The implementation details will be explained in this chapter. We shall use the term PET for the untimed version of PET and the term RPET (Real-time PET) for our real-time extension of PET.

8.1 The existing work

We give a short description of the work already done by Elsa Gunter and Doron Peled [GP99, GP00b, GP00a, GP02]. The GUI and the kernel of PET have the following functionalities:

1. Reading and analyzing a PASCAL program. The kernel of PET uses ML_lex and ML_yacc to parse a PASCAL program. ML_lex and ML_yacc report any lexical and syntax errors when parsing a PASCAL program,

according to the grammar specification implemented in PET. If a PASCAL program does not contains any syntax errors, a flow chart would be generated for it. The flow chart is stored in memory for further use and written to disk in the DOT input format for display. In addition, two more files are generated for the parsed program. One of them contains the coordinates of each statement in the parsed PASCAL program; the other describes the successors of each flow chart node. The coordinates of a statement include the starting line number and column number, and the ending line number and column number of the statement in the source code program file.

2. Displaying the flow chart and the source code of a PASCAL program on the screen. The GUI of PET calls DOT to read the file containing a flow chart in order to generate the coordinates of nodes and edges of the flow chart when the flow chart is displayed in a hierarchical graph. Then the GUI displays the flow chart in a window according to the coordinates returned by DOT. Moreover, the source code used to generate the flow chart is displayed in another window.

3. Displaying the highlight of a flow chart node when the mouse cursor is moved into the node. Remember that the GUI of PET was written in Tcl/Tk. When the mouse cursor is moved into a flow chart node, Tcl/Tk window management system generates a mouse event. The GUI intercepts the mouse event and defines the behavior of the event. The event behavior displays the flow chart node in a highlight color. Furthermore, the source code corresponding to this flow chart node

is highlighted. If the flow chart node is an *assignment* node, the corresponding source code is the assignment statement; if the node is a condition node of an `if` statement or a `while` statement, the corresponding source code is the whole statement. The file containing coordinates of statements is used to determine the location of the source code corresponding to each node. When the mouse cursor is moved out of the flow chart node, Tcl/Tk generates another mouse event and the GUI intercepts this event to remove the highlighting on the flow chart node and corresponding source code.

4. Selecting and displaying a path. When a flow chart node is clicked by the mouse, Tcl/Tk generate a mouse-click event. The GUI of PET maintain a queue which records all flow char nodes clicked. Indeed, the queue represents the selected path. The GUI defines the behavior of the event as follows:

   (a) If the node clicked is the first node which is clicked in this flow chart, put this node to the end of the queue and display it in a special color. This special color indicates that this node is the last node being clicked in this flow chart.

   (b) If the node clicked is not the first node being clicked in the flow chart, find the last node being clicked. If there is an edge starting from the last node clicked and pointing to the currently clicked node in the flow chart, put the node currently clicked to the end of the queue, display it in the special color and display the last node clicked in the normal color. If there is no such edge, dis-

play an error message in the main window to tell users that the node clicked cannot be added to the path. The file containing the successors of each flow chart node is used to determine whether there is such an edge.

The selected path is displayed in the path window. Each node in the selected path is displayed as a line of text. When the mouse cursor is moved onto a line of text, the flow chart node corresponding to the line and related source code are displayed in highlight.

5. Calculating the precondition of the selected path. The calculation is done by the kernel of PET. In order to do this, the kernel has the following functions to deal with boolean expressions:

   (a) Preprocess a boolean expression according to a set of rules, such as $expr \wedge true = expr$ and $expr \vee true = true$. Then the kernel calls the HOL90 simplification library to simplify the processed expression. The aim to preprocess an expression is to avoid unnecessary calls since a library call is time-consuming.

   (b) Replace a variable in a boolean expression by another boolean expression. If the variable occurs more than once in the expression, every occurrence is replaced.

   Calculating the precondition of a path is performed backwards from the end of the path to the beginning of the path.

6. Temporally debugging a system [GP02]. This was implemented solely by Elsa Gunter. At the time of thesis writing, it has not been integrated

into the version made available.

7. Miscellaneous functions.

   (a) Clear the selected path, i.e., remove all the nodes from the queue representing the path.

   (b) Remove the last node from the queue.

   (c) Exchange locations of two adjacent nodes in the queue. That is, if node a is the ith node in the queue and node b the $i+1$th node, set a as the $i+1$th node and b the ith.

   (d) Remove from the queue all nodes belonging to a particular flow chart.

   (e) Clear the content of the main window.

8.2 The implementation of code transformation

The following work has been done in order to implement code transformation.

1. Identify shared variables. The original implementation of PET puts all variables from all programs into one set. In order to identify shared variables, the original procedure of parsing PASCAL programs has been modified to put variables belonging to one program into a separate set. Then variables that appear in more than one set are identified as shared variables.

2. Generate the partial order from the selected path. We require that each assignment can reference at most one shared variable. A condition in

an `if` statement or a `while` statement can reference at most one shared variable too. For two nodes a and b in the path, an ordering relation $a \prec b$ is generated between them if (1) a appears earlier in the path than b and (2) they belong to the same flow chart or they reference the same share variable. Redundant relations are removed to obtain the reduced partial order (see Section 2.3). A redundant relation is a relation $a \prec c$ if there exists a node b in the path such that $a \prec b$ and $b \prec c$.

3. Add extra statements defined in Section 4.1.1 according to the partial order. At first, the number of occurrences of each node is inserted into the program. Then, statements of the form of Free_{ij} or Wait_{ji} are inserted into the program. At last, the last node of each program in the path is processed.

8.3 The real-time extension of PET

8.3.1 The real-time features

The real-time extension extends the key function of PET to the real-time programs domain. It has following new features with respect to time:

1. Annotating flow charts with time bounds. After the code is translated into flow charts, the testers need to input time bounds for nodes. According to the rules of translation, only one pair of bounds of each node can be fed, while the other pair of bounds is given by RPET. In general, the *assignment* node has *true* as its guard. Thus the bounds

for the guard of *assignment* nodes are predefined. The bounds of the transformation of *branch* nodes are predefined as well. However, there is an indirect way to change any bounds when necessary. The translated transition systems are stored in files. From the second time the source code is fed to RPET, RPET loads the bounds from these files without checking whether the bounds are changed or not. Hence, the testers could modify the bounds by editing files in order to change the default setting.

2. Computation and display of a partial order. The real-time extension allows the testers to select, modify and clear a path in the same way as the untimed version does. In addition, the selected path is decomposed into a partial order, which is displayed in a window. The displayed partial order consists of flow chart nodes. But the actual partial order which is used to construct a DAG is made up of edges of the extended timed automata. In fact, the selected path composed of flow chart nodes is first translated into the path composed of edges of the extended timed automata. Then the translated path is decomposed into the actual partial order. Finally the actual partial order is converted into the displayed partial order. The reason for this conversion is to provide the possibility to extend RPET further to model timed systems directly by transition systems.

3. Removing time constraints. Testers are allowed to remove time constraints from the location invariants of the ETAs in projected paths in order to calculate the precondition of time unbalanced partial orders.

The testers can choose either the first location or the last location or both in a projected path to remove time constraints. Each projected path is set up separately. This function is enhanced with calculating the maximum and the minimum execution time of each projected path.

4. Computation of the probability of a partial order. The probability of the selected partial order is calculated according to the methodology in Chapter 7. The testers must input the initial precondition of the partial order. If there are any parameters in time bounds, their values must be provided in the initial precondition.

Moreover, the GUI was improved to facilitate testing. The *wait* statement is displayed in the shape of a hexagon and a different color from the colors used for the box and diamond nodes. If a diamond node or a hexagon node appears in the selected partial order and its successor has not been decided, the node is displayed as flashing to remind the testers to choose a successor. Switches are provided to display transition systems or debugging information. The highlighting function is enhanced too.

8.3.2 The structure of the extension

Like the untimed version, the kernel and the GUI are the main components of the extension and they communicate with each other through a bidirectional pipe. The GUI uses DOT to display not only flow charts, but also the selected partial order. Our experience showed that the HOL simplifier does not provide the term in the appropriate format. In the latest version, we em-

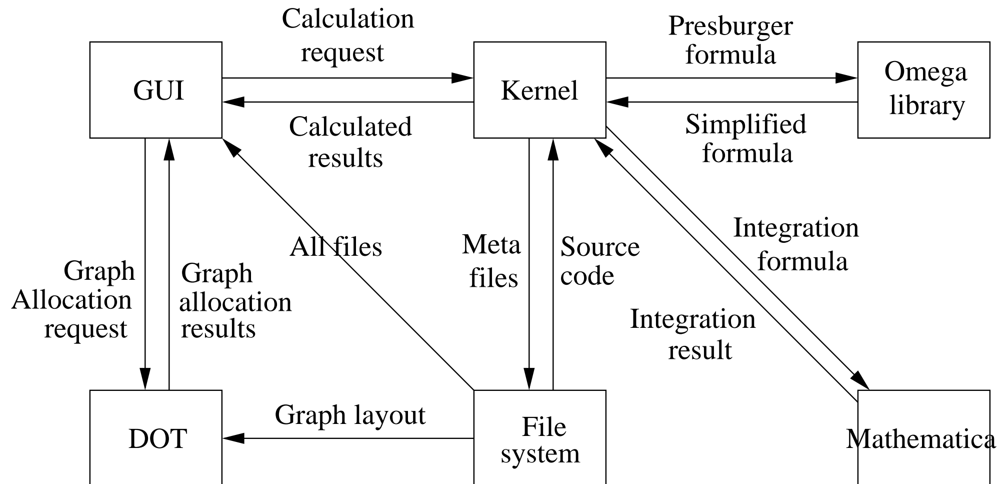

Figure 8.1: The PET structure

ployed the Omega library [KMP+95] for Presburger formula simplification. Figure 8.1 illustrates the structure of the extension.

The GUI includes six kinds of windows: source code windows, flow chart windows, transition system windows, a path window, a partial order window and a main window. Source code windows display the source code of every process. Figure 8.2 shows two source code windows. One (left) is for Process *sn* and the other (right) is for Process *rv*. The source code is written in PASCAL with some extensions.

Figure 8.3 demonstrates the corresponding flow chart windows of Process *sn* and *rv*. All nodes in one process are numbered from 0. The translation from the source code to flow charts is done by the kernel and the graph layout files are generated. The GUI employs the DOT program to calculate the graph allocation information, such as the size and the coordinates of each graph component, e.g., box, diamond, line, text. Then the GUI draws components in flow chart windows using this information.

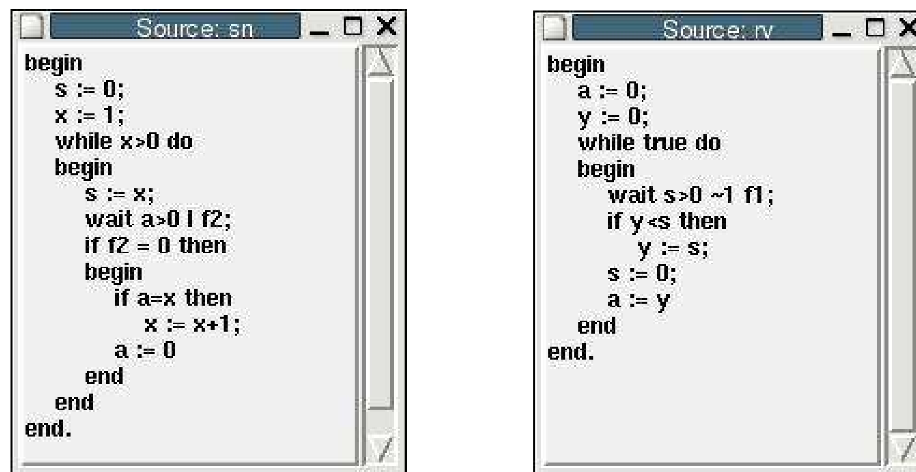

Figure 8.2: The source code windows

Usually the transition system windows are not displayed because the graphs in these windows have many nodes and edges. But RPET provides a switch button to control whether to display these windows. Figure 8.4 illustrates the transition system of Processes *sn* and *rv*. The extended timed automata are not displayed in RPET since they are too complicated and big to be displayed on the screen. However, they are stored in files and thus can be manually displayed in DOT if the testers are interested in them. The product automaton can be displayed in the same way.

The selected path is a total order, which is displayed in the path window as shown on the right of Figure 8.5. Each line in the path window contains three parts, each of which is separated by a colon. The left part is the sequence number; the middle one is the process name, which is represented by source file name; the right one is the number of the flow chart node. The square brackets in blue are used to indicate the node is in the box shape, the angle marks indicate diamond nodes and the square brackets in black indicate hexagon nodes. The partial order window, as shown on the

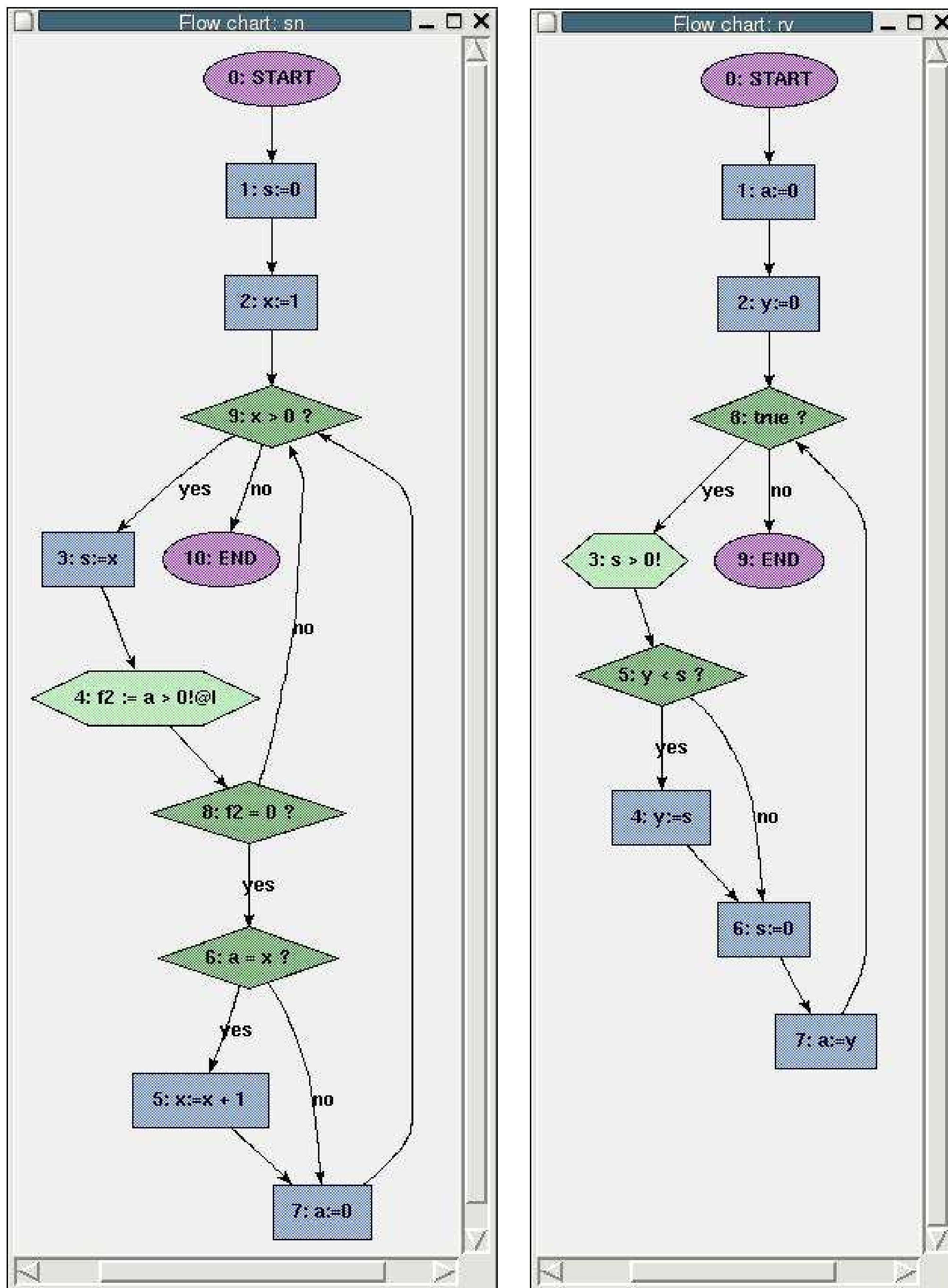

Figure 8.3: The flow chart windows

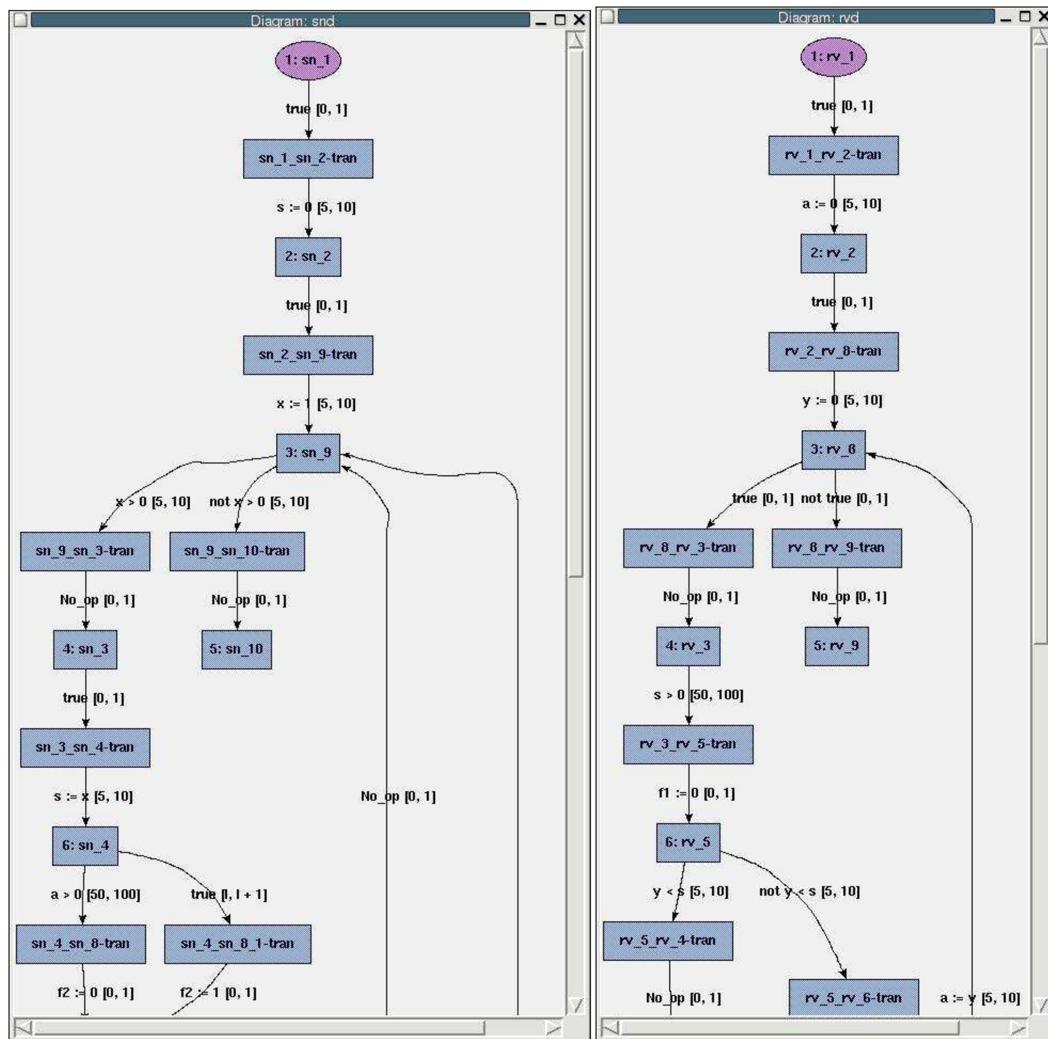

Figure 8.4: The transition system windows

left part of Figure 8.5, displays the partial order decomposed from the selected path. The edges in the partial order denote the execution order of the nodes.

The main window, as shown in Figure 8.5, contains a list of buttons, which define user commands. If a button is in gray, this button cannot be presses since the current environment does not satisfy its requirement. The color of the text in some buttons can be changed in order to represent differ-

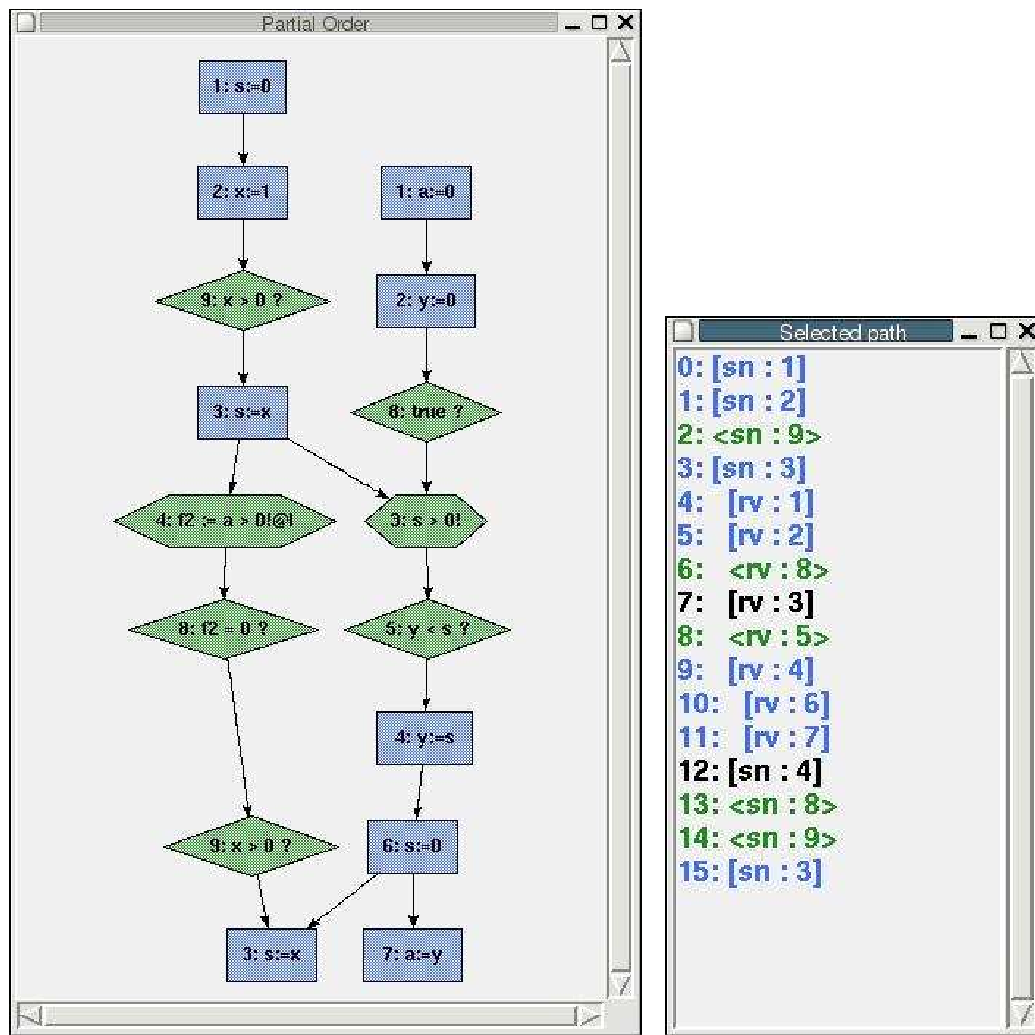

Figure 8.5: The partial order window and the path window

ent states. The results of some operations are displayed in the main window. The precondition of the partial order in Figure 8.5 is displayed in the main window in Figure 8.6.

8.3.3 The implementation details

Now we explain the implementation in detail.

1. Parsing a PASCAL program and generating the flow chart. Parsing a

```
main
Exit  Add  Unselect  Clear Path  Diagram  Debug  RmInv  ExecTime  Save  Calculate  Probability  Clear Pad
precondition=(146 <= I and I <= 244) or (I = 145 or ((66 <= I and I <= 144) or
((61 <= I and I <= 133) or ((56 <= I and I <= 122) or ((46 <= I and
I <= 49) or ((51 <= I and I <= 107) or I = 50)))))
minimum execution time:I + 40
precondition=I = 69 or I = 70
minimum execution time:110
precondition=(60 <= I and I <= 64) or (I = 65 or ((58 <= I and I <= 59) or (I = 69
or I = 70)))
minimum execution time:105
precondition=(61 <= I and I <= 123) or ((56 <= I and I <= 118) or ((51 <= I and
I <= 107) or ((46 <= I and I <= 49) or I = 50)))
minimum execution time:I + 45
precondition=(55 <= I and I <= 59) or ((50 <= I and I <= 54) or ((46 <= I and
I <= 49) or ((59 <= I and I <= 64) or I = 65)))
minimum execution time:100
precondition=(41 <= I and I <= 49) or ((54 <= I and I <= 59) or ((50 <= I and
I <= 58) or ((48 <= I and I <= 54) or I = 60)))
minimum execution time:95
precondition=(51 <= I and I < = 100) or I = 50
minimum execution time:I + 50
precondition=I = 248 or (I = 247 or (I = 246 or (I = 245 or ((51 <= I and I <= 54)
or ((146 <= I and I <= 244) or (I = 145 or ((66 <= I and I <= 144) or
((61 <= I and I <= 133) or ((56 <= I and I <= 122) or I = 55))))))))
minimum execution time:I + 35
precondition=(49 <= I and I <= 54) or ((45 <= I and I <= 53) or ((40 <= I and
I <= 48) or ((36 <= I and I <= 44) or (1 = 55 or (41 <= I and
I <= 49)))))
minimum execution time:90
precondition=(44 <= I and I <= 49) or ((40 <= I and I <= 48) or ((38 <= I and
I <= 44) or ((34 <= I and I <= 42) or ((33 <= I and I <= 41) or
((32 <= I and I <= 40) or ((31 <= I and I <= 39) or I = 50)))))
minimum execution time:85
```

Figure 8.6: The main window

PASCAL program was mostly based on the work of PET, except that the grammar of the `wait` statement has been changed in RPET. Two parameters were added to the statement. Similarly, the work on constructing a flow chart from a PASCAL program was borrowed from implementation of PET with the exception of translating a `wait` statement into a flow chart node.

2. Translating a flow chart into a process in a transition system. This part of work did not exist in the implementation of PET. Translation rules have been explained in Section 5.1 (for *assignment* nodes and *branch*

nodes) and Section 6.5 (for `wait` statements). The default time bounds are suggested by RPET, such as the bound for the condition *true* is chosen as $[0, 1]$. Meanwhile, the description of the process is written to a file.

3. Inputing time bounds for a flow chart node. A transition in a transition system has two pairs of bounds. Although they are both given as default after translating a flow chart into a process, one pair can be modified by users. By clicking a flow chart node using a middle mouse button, a dialog is popped up. Users can input the bounds in the dialog.

4. Saving time bounds. After at least one time bound is modified, users can save those bounds to a file. In fact, a new description of each process is written to a file. The old description in the file is replaced with the new version. At the same time, all extended timed automata and the product automaton are regenerated according to the updated time bounds contained in them.

5. Parsing a description file of a process in a transition system. After a file containing the description of a process has been created, RPET will not translate a flow chart into a process. On the contrary, RPET will load a process from the file. This file name has the same prefix as the PASCAL program file name. The detailed naming rules can be seen in Appendix B. ML_lex and ML_yacc are employed to create a parser for the process description.

6. Displaying flow charts and transition systems. Displaying a flow chart

has been done in PET. It was modified in RPET to display a `wait` statement node. A `wait` node is displayed in a box shape and using the same color as an *assignment* node in PET. We chose a new shape and a new color for this statement. The method to display transition systems is similar to displaying a flow chart. The difference is that a label for a flow chart node or edge always fits in one line, while a label for a node or an edge in the transition system may be split into multiple lines. This difference requires a slightly more complicated control of displaying a transition system compared to displaying flow charts.

7. Translating a process into an extended timed automaton. The translation rules has been explained in Section 5.3. At first, every transition in the process is translated separately. Then these separate pieces of translation are connected together. A node in the process can be the target node of a transition and the source node of another transition, but this node is translated twice when translating those two transitions. The connection is done by merging the two copies of one node together. A source node can be translated into multiple states, while a target node is translated into one state. Thus, the ETA state generated for a target node needs to be mapped to multiple states for a source node, and the edge pointing to the state corresponding to the target node needs to be produced in multiple copies, each copy pointing to one state translated for the source node. During the translation from a process into an ETA, the Omega library is used to simplify combinations of enabling conditions. If one combination is simplified to *false*, the corresponding state is removed. During the implementation of

the translation, we found that HOL90 gave results in an inappropriate format in some cases. This is why we chose the Omega library for the simplification. The communication between RPET and the Omega library is done through a C++ program, which was compiled with the Omega library. RPET writes an expression to a file and calls the C++ program. The C++ program writes the simplified expression to another file. Then RPET parses the file to get the expression. Parsing the file containing the simplified expression was implemented using ML_lex and ML_yacc.

8. Generating the product automaton of a set of extended timed automata. Firstly, we need to label each edge in each ETA according to Section 5.4. Secondly, we generate all the initial states of the product automaton. This is done by making a Cartesian product of initial states of each component ETA. We put these initial states of the product automaton onto a queue. Then the states in the queue are processed in turn. In each step, the first state in the queue is removed from the queue and processed. We generate the successors of the state being processed. If a successor has not been processed before, it is appended to the end of the queue. The processing procedure continues until the queue is empty. During the generation of successors of a state, we need to deal with synchronization of edges according to Section 5.4 and Section 5.5.

9. Selecting and displaying a path. Selecting a path from flow charts in RPET was done in the same way as that in PET. Displaying the se-

lected path was also borrowed from PET with improvements, such that in the path window, `wait` statement nodes are displayed in a different color from colors used for other nodes.

10. Generating and displaying the partial order from the selected path. The selected path is composed of flow chart nodes. In order to generate a partial order, each node in the path is translated into actions defined in Section 5.6. The basic idea to generate the partial order from a sequence of actions is similar to that in the implementation of code transformation in Section 8.2. But the generation procedure here is much more complex since each access to a shared variable is split into two steps: requesting and releasing. The generated partial order is composed of actions, while, when it is displayed on the partial order window, actions are converted to flow chart nodes because the partial order in flow chart nodes gives users more intuition than the partial order in actions. But the latter can be displayed manually (see Appendix B). The way to display a partial order is similar to displaying a flow chart. We improved it for the unique characteristic of partial orders. When a *branch* node is selected, whether the condition of the branch node is satisfied or not will not be decided until another node in the same flow chart is selected. In this case, the *branch* node in the partial order flashes.

11. Handling highlighting. The highlighting mechanism in RPET is the same as in PET. However, the highlighting control in RPET is much more complex than in PET since (1) a flow chart node is related not

only to a piece of source code, but also to one or more transitions in the corresponding process; (2) a text description of a node in the path window is related not only to nodes in flow chart windows and the corresponding source code, but also to nodes in the partial order window and edges in process windows.

12. Generating a DAG for the selected path. The method of generating a DAG has been introduced in Section 5.6. It was implemented in a similar way to constructing a product automaton. We first construct the initial nodes of the DAG. We analyse the partial order to obtain the first action of each process in the partial order and collect translated ETA states that correspond to source nodes of these first actions. Then we build a Cartesian product of these states. Each compound state is an initial state of the DAG and a state in the product automaton. Note that, if a process does not have any transitions participating in the partial order, we use its initial states when carrying out the Cartesian product. We put initial states to a queue. The states in the queue are processed in turn. Each time the head state of the queue is removed from the queue, its successors are constructed. If a successor has not been processed before, put it to the end of the queue. The processing stops when the queue is empty. The generation of successors of a state here is different from that in the generation of the product automaton. Here we use the synchronization of the product automaton and the partial order to determine the successors of a state. We find all edges starting at the state in the product automaton and check each of them whether it is allowed by the partial order. The target nodes of edges

allowed by the partial order are successors of the state.

13. Calculating the precondition of a partial order. The calculation method has been described in Section 6.4. In order to do that, backward DBM operations were implemented. We did not use DBM packages in existing model checkers, such as UPPAAL [BLL+95], because (1) it is difficult to integrate these non-SML packages into RPET, and (2) we need to deal with symbolic values in a DBM operation. After an operation over two DBMs, we may obtain multiple result DBMs, each of which has a different constraint on symbolic values. In the calculation, we need to replace a variable in a boolean expression by another expression. This was borrowed from PET.

14. Calculating estimated execution time of each process in the partial order. The implementation in SML follows the algorithm explained in Section 6.6.1 exactly.

15. Removing time constraints from location invariants. Users are allowed to choose which projected paths need to be applied to the method in Section 6.6.2. They can also choose the first node or the last node of a projected path to remove time constraints.

16. Calculating the probability of a partial order. The DAG generated from the partial order and the product automaton is used to calculate the probability according to the algorithm in Section 7.3.2. The forward DBM operations were implemented for the calculation. Each DAG node has a list to record values of variables. When a transition is symbolically executed, the list belonging to the target state of the

transition is generated by updating the list of the source state by the transformation of the transition. The integrals used to calculate the probability are sent to Mathematica. Mathematica returns the results of these integrations. The communication between RPET and Mathematica is performed through a C program. RPET writes data related to the integrations to a file and calls the C program. The C program reads the data file and calls the Mathematica kernel to compute the integrations. The Mathematica kernel returns the results to the C program and the C program writes the results to a new file. Then RPET parses the file to obtain the results. The parser was constructed using ML_lex and ML_yacc again. To date, the optimization for synchronized communication transitions has not been implemented yet.

17. Miscellaneous functions, such as clearing the selected path, removing the last node from the path, exchanging locations of two adjacent nodes in the path, projecting out from the path against a particular flow chart, and clearing the content of the main window, were basically borrowed from PET. We made changes to them according to the highlighting control in RPET.

8.4 Summary

This chapter introduced the implementation of the methodologies presented in Chapter 4, 5, 6, and 7. The methodology in Chapter 4 was implemented as an extension to PET and the others were implemented as the RPET tool, the real-time extension to PET. Note that the code transformation for distrib-

uted program model in Section 4.3 and the optimized algorithm to calculate the probability of a partial order in Section 7.3.3 has not been implemented yet, because it is limited to synchronized communication transitions, but RPET uses shared variables as the main interprocess communication approach. Most work in the whole implementation focused on RPET. The features, architecture and programming issues of RPET were explained in detail.

RPET supports symbolic values of time bounds, which allows users to obtain the possible range of these values as an expression in the precondition of a partial order. However, when we calculate time constraints symbolically, the initial constraints on time parameters, such as $l_1 \leq u_1$ and $L_1 \leq U_1$, are not always strong enough to guarantee that we can get the correct result of the comparison of two symbolic values, such as l_1 and u_2. We need to add additional assumptions, such as $l_1 < u_2$, $l_1 = u_2$ or $l_1 > u_2$, to the precondition. Thus the number of possible initial preconditions would be increased by a factor of three.

There are two DBM operations that need to be considered carefully during symbolic calculation. One is canonicalization. The other is to check whether a DBM is satisfiable. These two operations have higher complexity than other operations. Canonicalization can be computed by the Floyd-Warshall algorithm, which has $O(n^3)$ complexity. Checking satisfiability can be computed by the Bellman-Ford algorithm [Bel58, LRFF62, Moo59]. The Bellman-Ford algorithm checks a single source vertex to all other vertices and runs in $O(nm)$, which is actually $O(n^2)$ due to $m = n - 1$ in a DBM. Bellman-Ford algorithm has to be applied to every source vertex so

that checking satisfiability also runs in $O(n^3)$. Therefore, both of them generate $O(n^3)$ symbolic comparisons. Since each comparison may generate three new assumptions, the worst case complexity of computing precondition is $O(3^{n^3})$. Although the worst case never occurred in practice, we still experience a very long execution time. Similar result was obtained when handling parametric DBM in forward reachability analysis in [HRSV02].

During the development of RPET, we use the Omega library for the simplification of a Presburger expression. However, the library only works on integers, i.e., it simplifies the expression $l < 3$ into $l \leq 2$. For a variable defined over the real number domain, the simplified expression is not accurate. Another problem about the Omega library is that it cannot simplify the expression $l < 3 \vee l = 3$ into $l \leq 3$. This may result in a long expression for a precondition.

Chapter 9

Conclusion

This thesis focuses on the combination of model checking and testing to provide a practical tool set, which has elaborate theoretical background behind it. In this chapter, we summarize the four methodologies presented in previous chapters. More importantly, we discuss some relevant issues, which suggest directions of future research.

In Chapter 3, we presented a methodology and a syntax for annotating verified programs. In addition to program variables, there are extra variables, which are history and auxiliary variables, in annotations. Those variables record information about the search performed by a model checker. A history variable can be updated by program variables and history variables, and an auxiliary variable can be updated by program, history and auxiliary variables. The value of a history variable is rolled back when the search backtracks, while the value of an auxiliary variable is not. There are four new constructs, `commit`, `halt`, `report` and `annotate`, in annotations. The commit command prevents the search backtracking. The halt command

requires performing backtracking immediately. The report command stops the search and outputs the context of the search. The annotate command is used to collect the information and control the search. The program annotations are used to direct the search and suppress part of the state space, when the state space is too large to complete, though it compromises the exhaustiveness of model checking.

Indeed, there are many ways to use the method of annotating the code for automatic verification. One can use the annotations to extend the capabilities of a simple model checker that searches for deadlocks to one that checks safety properties, given as a deterministic finite automata. In this case, the automaton behavior is encoded within an annotation that is applied to the entire code. Some further constructs are conceivable. For example, we may allow checking whether the current state is already present in the search stack, or permit a nondeterministic choice between annotation actions. With such constructs, we can replace some fixed model checking algorithms, e.g., checking for temporal logic properties and various kinds of reductions, within annotations that are automatically generated. This can allow us an open ended model checker, with flexible capabilities.

In addition to applying the annotation method to verification as proposed above, another direction for future work is to design annotation structure for real-time system, since the proposed methodology works only for untimed systems. A possible way to annotate real-time systems is to augment enabling conditions and transformations of transitions. A potential usage of real-time annotations is to speed up the temporal debugging in real-time systems.

Chapter 4 proposed a code transformation which is used to check whether a suspicious behavior reported by a model checker or a theorem proving effort is indeed faulty. The key idea of the code transformation is to use explicit synchronizations, such as semaphore operations, to enforce the implicit partial order relations, such as references to shared variables, of an execution. Not only can the code transformation be applied to programs using shared variables, but also it can be used for distributed programs, which communicate with each other through communication channels. Althrough the extent that it can be adopted for infinite traces is limited in some cases, it gives useful information about traces in such cases. The transformation can also be used by testers, who need to demonstrate that a discovered error actually occurs. Due to the highly nondeterministic nature of concurrent programs, it may be difficult to enforce a suspicious scenario, which may depend on an infrequent scheduling choices.

The transformation depends on the *granularity* of atomic actions. In the running example, we assumed that the actions are related to the nodes of flow graphs of the processes. However, this is not necessarily the case. For example, it can be that some assignments use several shared variables or the same variable more than once (see a discussion in [OG76]). In this case, it is unlikely that the program will execute such an assignment atomically. We may need to translate this assignment into several actions, with some new variables for carrying temporary values. The interaction between the processes may generate dependencies at this lower level. Since assignments and conditions may appear within other structured code (e.g., a *while* statement containing the condition), other changes may be made to the code

(e.g., calculating the condition before and at the end of each loop, assigning its value to a Boolean variable, and checking that variable as the new loop condition).

The code transformation was developed for discrete systems. If we use it in a timed system, it will change time constraints of the system. Thus it is not appropriate to adopt it to enforce an execution satisfying a partial order in timed systems. However, it is possible to use the code transformation when we simulate a real-time system, since time constraints is under control in this case. Therefore, the code transformation in real-time systems is a future research direction. A possible way is to use the ϵ-transition (i.e., all its bounds are zero) to transform real-time programs.

In Chapter 6, we described a method for calculating the path condition in a timed system. The condition is calculated automatically, then simplified using various heuristics. Of course, we do not assume that the time constraints are given. The actual time for lower and upper bounds on transitions is given symbolically. Then we can make various assumptions about these values, e.g., the relative magnitude of various time constants. Given that we need to guarantee some particular execution and not the other, we may obtain the time constraints as path conditions, including, e.g., some equations, whose solutions provide the appropriate required time constants.

We believe that the constructed theory is helpful in the automatic generation of test cases. The test case construction can also be used to synthesize real-time system time. Another way to use this theory is to extend it to encapsulate temporal specification. This allows verifying a unit of code

in isolation. Instead of verifying each state in separation, one may verify the code according to the program execution paths. This was done for the untimed case in [GP03]. This framework can be extended for the timed case. Such a verification method allows us to handle infinite state systems (although the problem is inherently undecidable, and hence the method is not guaranteed to terminate), and parametric systems e.g., we may verify a procedure with respect to arbitrary allowed input. This is done symbolically, rather than state by state.

In addition, we identified time unbalanced partial orders in timed systems and gave its definition in this chapter. This phenomenon is caused by unbalanced projected path pairs and does not exist in untimed systems. The gap between a pair of unbalanced projected paths might force the system to enter a state from which the final state required by the partial order is unreachable. Due to the existence of time unbalanced partial order, testers may not easily distinguish an extendable partial order from an unextendable one, but these two kinds need different treatments. We proposed an algorithm to check whether a partial order is time unbalanced or not and a remedy method to transform an unbalanced partial order into a balanced one so that we can calculate its underlying precondition. We also applied the remedy method to simplify calculating the maximal and the minimal bounds of a time parameter. Generally speaking, on the one hand, time unbalanced partial order can cause problems for testers so that they must be careful when specifying a partial order; on the other hand, it is helpful to simplify computation in some circumstances.

In Chapter 7, we presented a methodology for calculating the prob-

ability of a path in a real-time system. We assume a uniform distribution, hence we do not need to provide probabilistic parameters of the system explicitly: the probabilities are being fixed by the sequences of enablings, and the time constraints on the individual transitions. The probability of execution depends not only on the occurring actions, but also on these becoming enabled without being executed. The methodology is not limited to a uniform distribution and it gives the exact value of the probability.

Our approach is also useful for optimizing test suites. One aspect is that we always want to run those test cases that have high probability first in order to save on time and resources. Another aspect is that given a choice of test cases, we may use the probability calculation in order to select the more likely cases.

The probability calculation for the uniform distribution case will be simplified if we can find an optimal method to calculated the volume of a polyhedron defined by the time constraint of an execution.

A promising future direction to apply the methodology of computing the probability of an execution is to perform probabilistic reachability analysis, i.e., what the probability of reaching a set of target states from the initial state is. This analysis technique can be extended to model checking a property in a probabilistic real-time system. Model checking techniques for this kind of probabilistic systems have been studied, e.g., in [Spr04]. But they give an approximate result. The exact result will be obtained by applying our methodology. Knowing how to accelerate the integration computation for a group of executions is the main problem when applying our methodology to reachability analysis and model checking.

In Chapter 8, we introduced the details of our implementation of code transformation and the Real-time PET system, which calculates the weakest precondition and the probability of a partial order in timed systems. The development of RPET is an ongoing topic. There are several directions for further development:

- RPET can handle symbolic parameters appearing on time bounds in timed systems. However, its runs slowly when dealing with symbolic values. There are several ways to accelerate the calculation in terms of symbolic parameters. An intuitive way is to use a parallel computer to do the calculation. Since the operations on one DBM does not communicate with operations on other DBMs, the current sequential algorithm can be modified to a parallel or distributed algorithm without theoretical difficulty. A second way is to apply partial order reduction to the current algorithm. Partial order reduction on model checking timed automata has been studied in [BJLY98, Min99]. But their results need modification before they can be applied. Furthermore, both ways can be combined together. Optimizing the procedure of handling symbolic values carefully is another way. But its effect on speed of calculation is not as prominent as that of other methods.

- Current functionalities of RPET are very restricted. The code transformation for programs using communication channels and the optimized probability computation for a partial order have not been implemented yet. This will be done in the near future. Temporal debugging for real-time systems can be implemented as well.

- The usability of RPET can be improved, e.g., allowing users to customize the default values for time bounds, allowing them to modify the source code and on-the-fly update system models, which are the flow charts, the transition system and the ETAs.

In conclusion, this thesis is dedicated to assisting software testing with model checking techniques. It brings new ideas into software testing and goes beyond traditional testing approaches to software development. One problem related to these techniques is that some of them require expertises to use and are not suitable for inexperienced users. More research needs to be done in order to make them easy to use. The presented techniques have been implemented into practical tools to demonstrate their merits. They can be integrated into standard tools in commercial software development as well. The research in this thesis provides an open view for future research on applying model checking, or broadly speaking, formal methods to meet the challenging demands in different domains of software testing.

Appendix A

BNF grammar of the annotation

%nonassoc LOWER_THAN_ELSE

program: MAIN '(' ')' '{' global_defi_list thread_list '}'

;

global_defi_list: definition

| global_defi_list definition

;

definition: dtype var_list ';'

;

thread_list: thread

| thread_list thread

;

thread: THREAD VARIABLE '{' stmt_list '}'

;

stmt_list: stmt

| stmt_list stmt

;

```
assignment: VARIABLE '=' expr
        ;
var_list: var
        | var_list ',' var
        ;
var: VARIABLE
        | VARIABLE '=' expr
        ;
dtype: standard_type
        | modifier standard_type
        ;
modifier: HISTORY
        | AUXILIARY
        ;
standard_type: INT
        | SEMAPHORE
        ;
expr: relexpr1
        | expr OR relexpr1
        ;
relexpr1: relexpr1 AND relexpr2
        | relexpr2
        ;
relexpr2: relexpr
        | '!' relexpr2
        ;
```

```
relexpr: mathexpr
        | relexpr relop mathexpr
        ;
mathexpr: mathexpr ’+’ mulexpr
        | mathexpr ’-’ mulexpr
        | mulexpr
        ;
mulexpr: mulexpr ’*’ primary
        | mulexpr ’/’ primary
        | mulexpr ’%’ primary
        | primary
        ;
primary: ’-’ primary
        | ’(’ expr ’)’
        | INTEGER
        | VARIABLE
        ;
relop: EQU
        | ’<’
        | ’>’
        | LE
        | SE
        | NEQ
        ;
stmt: annotated
        | basic_stmt
```

| sema_stmt

;

basic_stmt: ';'

| INT var_list ';'

| assignment ';'

| FOR '(' assignment ';' expr ';' assignment ')' basic_stmt

| WHILE '(' expr ')' stmt

| DO stmt WHILE '(' expr ')' ';'

| IF '(' expr ')' stmt %prec LOWER_THAN_ELSE

| IF '(' expr ')' stmt ELSE stmt

| '{' list_basic_stmt '}'

;

list_basic_stmt: basic stmt

| list basic stmt ';' basic stmt

;

annotated: WITH '{' stmt '}' ANNOTATE '{' annotation '}'

| ANNOTATE '{' annotation '}'

;

sema_stmt: P '(' SEMA_VARIABLE ')' ';'

| V '(' SEMA_VARIABLE ')' ';'

;

annotation: basic_stmt_plus

| WHEN '(' expr ')' basic_stmt_plus

| WHEN '(' expr ')'

;

basic_stmt_plus: basic_stmt

| COMMIT ';'

| HALT ';'

| REPORT '(' STRING ')' ';'

;

Appendix B

Real-time PET user manual

Real-time PET can be started by the command `'tester'`. The syntax of this command is

`tester -N` *filename1* [*filename2 filename3* ...],

where filename1, filename2 and filename3 are file names. Each file contains a program. At least one file must be specified. After RPET has been started, flow charts of programs specified by file names are displayed in separate windows.

For each program, RPET generates a flow chart, a process in the transition system, and an extended timed automaton. The flow chart will be saved to a new file whose name is the concatenation of the name of the program file and the suffix ".dot", the text description of the process in the transition system will be saved to a file whose name is the concatenation of the program file name and the suffix "d", and the graphical description is saved to a file whose name is the concatenation of the text description file name and the suffix ".dot". The extended timed automaton will be saved to a new file with the name of the concatenation of the text de-

scription file name and the suffix ".auto.dot". For example, suppose we feed two files "sn" and "rn" to RPET. The flow charts are stored in files "sn.dot" and "rv.dot". The text descriptions of processes in the transition system are stored in files "snd" and "rvd". The graphical descriptions are stored in "snd.dot" and "rvd.dot". The ETAs for the processes are stored in "snd.auto.dot" and "rvd.auto.dot".

RPET also generates the product automaton of extended timed automata. The graphical description of the product automaton is stored in a file whose name is the concatenation of the text description file names of all component processes and the suffix ".product.dot". A simplified version of the graphical description is store in a file whose name is the concatenation of the component processes' names and the suffix ".product.simp.dot". The simplified version does not describe the detail of each transition, such as clocks and transformations, and is used to illustrate the structure of the product automaton. For the example of programs "sn" and "rv", the product automaton is saved in "snd_rvd.product.dot" and the simplified version is in "snd_rvd.product.simp.dot".

When a partial order is given, RPET generates the DAG from the product automaton and the partial order. The partial order is stored to a file whose name is ".temppath.dot". The DAG is stored to a file whose name is the concatenation of the component processes' name and the suffix ".dag.dot". For the above example, the DAG is stored in "snd_rvd.dag.dot".

Those new files with suffix ".dot" can be converted by DOT to common graphic file format, such as JPG and FIG. Many figures in Chapters 5, 6 and 7 were obtained in this way.

RPET supports mouse operations in the following ways:

1. Specifying an execution in RPET is done by using a left mouse button clicking nodes of flow charts one by one.

2. Modifying time bounds of a node (which is translated into a transition in the transition system) by using a middle mouse button clicking the node.

3. Moving the mouse cursor into a node to check the source code related to it.

Users can also use the tool bar in the main window to manipulate an execution. Commands on the tool bar are explained in detail from the left of the tool bar to the right.

1. *Exit* button. Exit Real-time PET.

2. *Add* button. Open a new file and import the program in it. After clicking this button, a dialog will be popped up. It requires the user to input the file name. The program in this file will be added to the system, which is composed of programs in files already opened.

3. *Unselect* button. This command removes the last transition (a flow chart node) in the specified execution. If the execution is empty, this command is disabled. It is enabled as soon as the first transition of the execution is selected.

4. *Clear Path* button. It deletes the selected execution. In the meantime, it disables the *Unselect* button.

5. *Diagram* button. When clicking this button once, the transition system generated from the flow charts are displayed in new windows. These windows will be closed when clicking the button again.

6. *Debug* button. If clicked once, it will cause RPET to output debugging information when calculating the precondition or the probability of the partial order generated from the selected execution. The output of debugging information will be disabled when clicking the button once more.

7. *RmInv* button. This command is used to apply the remedy method in Section 6.6.2, i.e., removing time constraints from location invariants. Users can choose whether the first or the last location of a specified projected path or both should be applied. It will pop up a dialog to allow the user to choose projected paths and locations. Each projected path has two check boxes on the dialog, one for the first location and the other for the last location. If a check box is selected, the corresponding location will be applied to the remedy method. This command is disabled if the selected execution is empty.

8. *Exec Time* button. This command displays the execution time of each projected path in the selected execution using the algorithm in Section 6.6.1.

9. *Save* button. This command saves the time bounds of all transitions to the files. Since each program is contained in a separate file, the bounds for transitions belonging to one program is saved to one file, whose name is the concatenation of the name of the program file and the string “d”. If no bound has been modified since RPET started, this command is disabled. Only after at least one bound is modified is it enabled.

10. *Calculate* button. This command calculates the precondition of the partial order generated from the selected execution. If the selected execution is empty, the command is disabled, and is enabled otherwise.

11. *Probability* button. It calculates the probability of the partial order generated from the selected execution. Before calculation, it will pop up a dialog to require the user to input the initial condition of the partial order. The initial condition is in the form of a boolean expression. If there are any parameterized time bounds, the user needs to provide their exact values. The values are input as a boolean expression, which is the conjunction of equations. Each equation is of the form $v = expr$, where v is a parameter and $expr$ is an arithmetic expression. After all fields in the dialog has been filled completely, the probability is computed and displayed in the main window. The command is disabled if the selected execution is empty.

12. *Clear Pad* button. This command removes all the content displayed in the main window.

13. *Help* button. It is displayed as a human head with a question mark in it. The command shows a simple help information to users.

14. *Info* button. It is the last button on the right. It displays information about Real-PET, such as the copyright and the version information.

Bibliography

[ACD91] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for probabilistic real-time systems. In *Automata, Languages and Programming: Proceedings of the 18th ICALP*, pages 115–136. LNCS 510, 1991.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[Alu98] Rajeev Alur. Timed automata. In *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998.

[AMT94] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for standard ml. URL: http://www.smlnj.org/doc/ML-Lex/manual.html, 1994.

[APP95] Rajeev Alur, Doron Peled, and Wojciech Penczek. Model-checking of causality properties. In *the* 10th *Symposium on Logic in Computer Science*, pages 90–100. IEEE, 1995.

[BA90] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[BBD03] Jeremy Bryans, Howard Bowman, and John Derrick. Model checking stochastic automata. *ACM Transactions on Computational Logic*, 4(4):452–492, 2003.

[BD04] Mario Bravetti and Pedro R. D'Argenio. Tutte le algebre insieme: Concepts, discussions and relations of stochastic process algebras with general distributions. In *GI/Dagstuhl Research Seminar Validation of Stochastic Systems*, pages 44–88. LNCS 2925, 2004.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BHHK03] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *the* 9th *International Conference on Concurrency Theory*, pages 485–500. LNCS 1466, 1998.

[BLL$^+$95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *the* 9th *DIMACS Workshop on*

Verification and Control of Hybrid Systems, pages 232–243. LNCS 1066, 1995.

[BMS91] Navin Budhiraja, Keith Marzullo, and Fred B. Schneider. Derivation of sequential, real-time process-control programs. *Foundations of Real-Time Computing: Formal Specifications and Methods*, pages 39–54, 1991.

[Bor98] Aleksandr Alekseevich Borovkov. *Probability theory*. Gordon and Breach, 1998.

[BPQT04] Saddek Bensalem, Doron Peled, Hongyang Qu, and Stavros Tripakis. Automatic generation of path conditions for concurrent timed systems. In *the* 1st *International Symposium on Leveraging Applications of Formal Methods*, 2004.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *the Workshop on Logics of Programs*, pages 52–71. LNCS 131, 1981.

[CEJS98] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *the* 10th *International Conference on Computer Aided Verification*, pages 147–158. LNCS 1427, 1998.

[CFJ93] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *the* 5th *International Workshop on Computer Aided Verification*, pages 450–462. LNCS 697, 1993.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGMP99] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.

[DF88] Martin E. Dyer and Alan M. Frieze. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing*, 17(5):967–974, 1988.

[Dij68] Edsger W. Dijkstra. The structure of the “the”-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212. LNCS 407, 1989.

[DPCPG04] Giovanni Denaro, Mauro Pezzè, Alberto Coen-Porisini, and Carlo Ghezzi. A symbolic execution based approach for verifying safety-critical systems. *URL:*

http://www.lta.disco.unimib.it/homepage/giovanni.denaro/papers /TSE(submitted)2004.pdf, 2004.

[DR95] Volker Diekert and Grzegorz Rozenberg. *The Book of Traces*. World Scientific, 1995.

[ELLL04] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.

[EP02] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software systems. In *SPIN 2002*, pages 230–239. LNCS 2318, 2002.

[ES93] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In *the* 5th *International Workshop on Computer Aided Verification*, pages 463–478. LNCS 697, 1993.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[Fra86] Nissim Francez. *Fairness*. Springer-Verlag, 1986.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1991.

[Gly89] Peter W. Glynn. A gsmp formalism for discrete-event systems. *Proceedings of the IEEE*, 77:14–23, 1989.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[GP99] Elsa L. Gunter and Doron Peled. Path exploration tool. In *the* 5th *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 405–419. LNCS 1579, 1999.

[GP00a] Elsa L. Gunter and Doron Peled. Pet: an interactive software testing tool. In *the* 12th *International Conference on Computer Aided Verification*, pages 552–556. LNCS 1855, 2000.

[GP00b] Elsa L. Gunter and Doron Peled. Using a mix of languages in formal methods: the pet system. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2981–2986, 2000.

[GP02] Elsa L. Gunter and Doron Peled. Temporal debugging for concurrent systems. In *the* 8th *International Conference on Tools and*

Algorithms for Construction and Analysis of Systems, pages 431–444. LNCS 2280, 2002.

[GP03] Elsa L. Gunter and Doron Peled. Unit checking: Symbolic model checking for a unit of code. In *Verification: Theory and Practice 2003, Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, pages 548–567. LNCS 2772, 2003.

[Har00] Mary J. Harrold. Testing: a roadmap. In *Future of Software Engineering, the* 22nd *International Conference on Software Engineering*, pages 61–72. ACM Press, 2000.

[HMP94] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112:273–337, 1994.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[Hoa85] C.A.R. Hoare. *Communication Sequential Processes*. Prentice Hall, 1985.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[HP94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *the* 7th *IFIP International Conference on Formal Description Techniques*, pages 197–211, 1994.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), 2000.

[HRSV02] Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002.

[JPQ05] Marcin Jurdziński, Doron Peled, and Hongyang Qu. Calculating probabilities of real-time executions. In *the* 5th *International Workshop on Formal Approaches to Testing of Software*, 2005.

[JWMM91] Kathleen Jensen, Niklaus Wirth, Andrew B. Mickel, and James F. Miner. *Pascal User Manual and Report: ISO Pascal Standard, Fourth Edition*. Springer, 1991.

[KD01] Joost-Pieter Katoen and Pedro R. D'Argenio. General distributions in process algebra. In *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, pages 375–430. LNCS 2090, 2001.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KK67] Georg Kreisel and Jean-Louis Krivine. *Elements of mathematical logic (model theory)*. North Holland Pub. Co, 1967.

[KM89] Robert P. Kurshan and Kenneth L. McMillan. A structural induction theorem for processes. In *the* 8th *annual ACM Sym-*

posium on Principles of Distributed Computing, pages 239–247. ACM Press, 1989.

[KMP+95] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical report cs-tr-3445, CS Dept., University of Maryland at College Park, March 1995.

[KNSS00] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In *11th International Conference on Concurrency Theory*, pages 123–137. LNCS 1877, 2000.

[KNSS02] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.

[KP90] Shmuel Katz and Doron Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75(3):263–287, 1990.

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[Kur95] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LHK01] Gabriel G. Infante López, Holger Hermanns, and Joost-Pieter Katoen. Beyond memoryless distributions: Model checking semi-markov chains. In *Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pages 57–70. LNCS 2165, 2001.

[LMB92] John Levine, Tony Mason, and Doug Brown. *lex & yacc, Second Edition*. O'Reilly, 1992.

[LN00] Ranko Lazic and David Nowak. A unifying approach to data-independence. In *the* 11th *International Conference on Concurrent Theory*, pages 581–595. LNCS 1877, 2000.

[LRFF62] Jr. Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[LV03] László. Lovász and Santosh Vempala. Simulated annealing in convex bodies and an $o^*(n^4)$ volume algorithm. In *the* 44th *Annual IEEE Symposium on Foundations of Computer Science*, pages 650–659. IEEE, 2003.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324. LNCS 255, 1986.

[McM92] Kenneth L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[McM93] Kenneth L. McMillan. *The Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Min99] Marius Minea. Partial order reduction for model checking of timed automata. In *the* 10th *International Conference on Concurrency Theory*, pages 431–446. LNCS 1664, 1999.

[Moo59] Edward F. Moore. The shortest path through a maze. In *the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *the* 10th *ACM Symposium on Principles on Programming Languages*, pages 141–151. ACM, 1983.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag, 1992.

[MPS98] Anca Muscholl, Doron Peled, and Zhendong Su. Deciding properties for message sequence charts. In *the First International Conference on Foundations of Software Science and Computation Structure*, pages 226–242. LNCS 1378, 1998.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MTM97] Robin Milner, Mads Tofte, and Robert Harperand David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[Nag03] Trygve Nagell. *Introduction to Number Theory, second edition*. Chelsea Publishing Company, 2003.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13:85–108, 1981.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[Pax95] Vern Paxson. Flex, version 2.5. URL: http://www.gnu.org/software/flex/manual/html_mono/flex.html, 1995.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *the* 6th *International Conference on Computer Aided Verification*, pages 377–390. LNCS 818, 1994.

[Pel02] Doron Peled. *Software Reliability Methods*. Springer-Verlag, 2002.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *the* 18th *Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pnu81] Amir Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[PQ03] Doron Peled and Hongyang Qu. Automatic verification of annotated code. In *the* 23rd *IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, pages 127–143. LNCS 2767, 2003.

[PQ05a] Doron Peled and Hongyang Qu. Enforcing concurrent temporal behaviors. *Electronic Notes in Theoretical Computer Science*, 113:65–83, 2005.

[PQ05b] Doron Peled and Hongyang Qu. Timed unbalanced partial order. In *the* 5th *International Workshop on Formal Approaches to Testing of Software*, 2005.

[Pra86] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[PWW98] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *the* 5th *Colloquium on International Symposium on Programming*, pages 337–351. LNCS 137, 1982.

[RLK$^+$01] Theo C. Ruys, Rom Langerak, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. First passage time analysis of stochastic process algebra using partial orders. In *the* 7th *Inter-*

national Conference on Tools and Algorithms for Construction and Analysis of Systems, pages 220–235. LNCS 2031, 2001.

[Sch98] Murray Schechter. Integration over a polyhedron: an application of the fourier-motzkin elimination method. *The American Mathematical Monthly*, 105(3):246–251, 1998.

[Seg95] Roberto Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[Spr04] Jeremy Sproston. Model checking for probabilistic timed systems. In *Validation of Stochastic Systems*, pages 189–229. LNCS 2925, 2004.

[SZ92] David J. Scholefield and Hussein S. M. Zedan. Weakest precondition semantics for time and concurrency. *Information Processing Letters*, 43:301–308, 1992.

[TA00] David R. Tarditi and Andrew W. Appel. Ml-yacc user's manual. URL: http://www.smlnj.org/doc/ML-Yacc/index.html, 2000.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 133–191. MIT Press/Elsevier, 1990.

[TW97] P. S. Thiagarajan and Igor Walukiewicz. An expressively complete linear time temporal logic for mazurkiewicz traces. In *the* 12th *Annual Symposium on Logic in Computer Science*, pages 183–194. IEEE, 1997.

[Vog93] Walter Vogler. Bisimulation and action refinement. *Theoretical Computer Science*, 114(1):173–200, 1993.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *the* 1st *Annual Symposium on Logic in Computer Science*, pages 332–344. IEEE, 1986.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[Whi80] Ward Whitt. Continuity of generalized semi markov processes. *Mathematics of Operation Research*, 5(4):494–501, 1980.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *the* 13th *Annual Symposium on Principles of Programming Languages*, pages 184–193. ACM, 1986.

[Wol03] Stephen Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, 2003.

[WvH03] Kurt Wall and William von Hagen. *The definitive guide to GCC*. Apress, 2003.

[Yov98] Sergio Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, pages 114–152. LNCS 1494, 1998.

[YS04] Håkan L. S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type

distributions. In *Proc. Nineteenth National Conference on Artificial Intelligence*, pages 742–748. AAAI Press, 2004.