Design and Implementation of an Analyzer for a Timed π-calculus

by

Md Rezoanoor Rahman

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
September 2010

Copyright © Md Rezoanoor Rahman, 2010

Abstract

In this thesis, we design and implement an analysis tool for a language called kiltera which is a timed extension of the π-calculus. kiltera allows the modeling of concurrent, mobile, real-time and distributed systems. Our analyzer takes a kiltera model as input and performs analyses such as detection of deadlock states, stable states etc. To improve performance, the analyzer uses some transformation techniques to simplify the input kiltera model without changing its behavior. We provide a detailed description of the design and implementation of the analyzer and discuss some performance optimizations. Finally, we present some case studies to illustrate the capabilities of our analyzer.

Acknowledgments

I would like to thank my supervisor Juergen Dingel for his guidance, his help and his financial support through out last two years. I also like to thank the School of Computing for providing everything I needed to complete my tenure. The staffs at School of Computing were so helpful that made my graduate life easy. I would also like to thank the committee members Karen Rudie and Bob Tennent for their comments and critiques. I extend my gratitude to Ernesto Posse at the School of Computing at Queen's University for helping me through out the last two years about the research, writing my thesis and providing me the technical directions about the implementation of the thesis. I take the opportunity to thank Karolina Zurowska and Eyrak Paen for their help in the last two years. My special thank goes to Salimur Choudhury and Tawhid Bin Waez for their continuous inspirations. I thank all of my friends in facebook for accompanying me while away from them in last two years. Special thank goes to Google for their wonderful products to help me in doing my research. Finally I want to thank my parents and my sister for everything they did in my life. Thanks to Almighty who helped me to remain calm in my tough time, and helped me to have patience.

Contents

Abstract . . . . **i**

Acknowledgments . . . . **ii**

Contents . . . . **iii**

List of Tables . . . . **viii**

List of Figures . . . . **ix**

Chapter 1:

Introduction **1**

1.1 Summary of contribution 4

1.2 Organization of thesis 5

Chapter 2:

Background **6**

2.1 kiltera 6

2.1.1 Process terms 7

2.1.1.1 Termination 7

2.1.1.2 Events 7

2.1.1.3 Sequential and parallel compositions 9

2.1.1.4 Process definitions 9

2.1.1.5 State variables 12

2.1.1.6 Time 13

2.1.1.7 Conditionals and pattern matching 14
2.1.2 Example 14
2.1.2.1 Problem description 15
2.2 Design patterns 22
2.2.1 Composite pattern 22
2.2.2 Visitor pattern 24
2.3 π_{klt} syntax 28

Chapter 3:

Analyses 31

3.1 π_{klt} analyzer: how does it work? 31
3.2 Pre-processor 33
3.2.1 Preliminary definitions 33
3.2.2 Deep simplification 34
3.2.3 Renaming 35
3.2.4 Scope extrusion 36
3.3 Analyzer core 38
3.3.1 Shallow simplification 39
3.3.2 π_{klt} state 40
3.4 Analyses modes 41
3.4.1 Interactive mode 41
3.4.2 Exhaustive mode 42
3.4.3 Bounded mode 42
3.5 Open and closed systems analysis 43
3.6 Kinds of analyses 44
3.6.1 Stable states 44
3.6.2 Deadlock states 45
3.6.3 Reachability analysis 45
3.6.4 Searching for triggers and receivers 45
3.6.5 Search all events 46

3.6.6 Search any events 46
3.6.7 Longest and shortest execution time 46
3.6.8 Equivalence frequency 47
3.6.9 Specific kiltera process invocation 47
3.6.10 Searching for dropped signals 47

Chapter 4:

Implementation of the π_{klt} analyzer 48

4.1 Abstract syntax 48
4.1.1 Expressions 49
4.1.2 Patterns 50
4.1.3 Process terms 50
4.1.4 Process definition and function definition 54
4.1.5 Values 55
4.1.6 Actions and symbolic actions 55
4.2 Visitors 56
4.2.1 Expression visitor 56
4.2.2 Pattern visitor 58
4.2.3 ProcTermVisitor and ProcDefVisitor 59
4.2.3.1 Free names and bound names 61
4.2.3.2 Substitution 61
4.2.3.3 Concrete time advance and evolution 62
4.2.3.4 Deep and shallow simplification 62
4.2.3.5 Renaming 63
4.2.3.6 Scope extrusion 63
4.2.3.7 Length of a π_{klt} term 64
4.2.3.8 Sorting parallel terms 65
4.2.3.9 Alpha-equivalence checking 65
4.2.3.10 First actions 66
4.2.3.11 Next state set 69

4.2.4 State visitor . . . 72
4.2.5 Symbolic action visitors . . . 73
4.2.6 Action visitor . . . 74
4.2.7 Values visitor . . . 74
4.3 Performance optimization of the analyzer . . . 74
4.3.1 Improvement of alpha-equivalence checking . . . 74
4.3.2 Get rid of redundant renaming . . . 76
4.3.3 Implementing the seen set . . . 77

Chapter 5:

Evaluation of the π_{klt} analyzer . . . 79
5.1 Case study: finding a deadlock . . . 79
5.1.1 General description: death model . . . 79
5.1.2 Analysis question: death model . . . 80
5.1.3 Analysis output: death model . . . 81
5.2 Case study: token ring algorithm . . . 85
5.2.1 General description: token ring model . . . 85
5.2.2 Analysis questions: token ring model . . . 87
5.2.3 Analysis output: token ring model . . . 88
5.2.4 Simple scalability analysis of π_{klt} analyzer . . . 90
5.3 Case study: dropped signal analysis . . . 91
5.3.1 General description: model for dropped signal analysis . . . 92
5.3.2 Analysis question: dropped signal analysis . . . 93
5.3.3 Analysis output: dropped signal analysis . . . 94

Chapter 6:

Related work . . . 97
6.1 Language perspective . . . 97
6.1.1 CCS . . . 98
6.1.1.1 The π-calculus . . . 98

6.1.2 CSP . . . 98
6.1.2.1 LOTOS . . . 99
6.1.2.2 Timed CSP . . . 99
6.1.3 π_{klt} . . . 100
6.1.4 Other timed extensions of CCS and CSP . . . 100
6.2 Tools perspective . . . 101
6.2.1 ProBE . . . 101
6.2.2 FDR2 . . . 101
6.2.3 The concurrency workbench . . . 101
6.2.4 The mobility workbench . . . 102
6.2.5 Construction and analysis of distributed processes . . . 102
6.2.6 Other tools . . . 102

Chapter 7:

Conclusion and future work . . . 105
7.1 Our contribution . . . 105
7.1.1 π_{klt} analyzer . . . 105
7.1.2 Transformation of models . . . 106
7.1.3 Removing redundant renaming . . . 106
7.2 Limitations . . . 107
7.3 Future work . . . 107
7.4 Conclusion . . . 108

Bibliography . . . 110

Appendix A:

kiltera **model and UML-RT state machines . . . 117**

List of Tables

2.1 Classes and their attributes and operation 23

2.2 π_{klt} syntax 30

4.1 Classes and π_{klt} constructs 51

5.1 Performance evaluation of the π_{klt} analyzer 91

List of Figures

2.1 Server and client example 11
2.2 Widget production control system 15
2.3 Trace output for widget production controller system 22
2.4 Abstract syntax tree for example expression 24
2.5 Composite pattern for example BNF 25
2.6 Composite pattern for example BNF 25
2.7 Visitor pattern structure for example BNF 26
2.8 Control flow for function invocations 29

3.1 π_{klt} analyzer general framework 32

4.1 π_{klt} expressions 49
4.2 π_{klt} patterns 50
4.3 Process terms class diagram 52
4.4 π_{klt} values 55
4.5 π_{klt} Actions and symbolic actions 56
4.6 π_{klt} expression visitor 57
4.7 π_{klt} pattern visitor 58
4.8 π_{klt} process terms visitor 60
4.9 π_{klt} process definition visitor 61
4.10 π_{klt} state visitor 72
4.11 π_{klt} symbolic action visitor 73
4.12 Hash table 78

5.1 Death: an example of deadlock . . . 80
5.2 Portion of the state space tree for the Death model . . . 82
5.3 Deadlock state report screen shot . . . 83
5.4 Interactive mode screen shots . . . 84
5.5 Token ring example . . . 85
5.6 Trace to a stable state . . . 89
5.7 Process vs time . . . 92
5.8 State machine of the UML-RT model . . . 93
5.9 Dropped signal analysis . . . 96

A.1 UML-RT state machine . . . 118

Chapter 1

Introduction

Modeling is a proven and well-accepted engineering technique. For instance, it is common practise to construct architectural models for visualization and analysis of a building that we are going to build. Consequently, the design of a good model is often crucial and a correctly designed model can save time and money. Engineering disciplines today rely heavily on modeling techniques and do analysis on the models prior to the development of the product.

Roughly speaking, a model describes an entity in some abstract, that is, somehow simplified, way. Models typically have a purpose that allows some details of the entity to be ignored and excluded from the models. For example, we could build a useful model for a car without caring much about the functionality of its engine.

The situation is similar with models used in a software development process. Behavioral software models are often used to describe behaviors (executions) that the system to be built is expected to have. The model may not be complete but it should support the purpose for which the model has been created. Just like models in general, software models can help us understand a complex system and its potential solutions through abstraction and correctly designed models can facilitate the efficient construction of correct software.

Due to the increased level of abstraction, it is easier to implement an algorithm in Java than in

machine language. The level of abstraction can be increased even more if we use a general purpose modeling language such as UML [19] or a domain-specific modeling language supported by a graphical modeling tool. If the tool also allows the automatic generation of executable code from the models, the two central ingredients for so-called *Model Driven Development* (MDD) [62] are in place. The idea of MDD is to use models as primary development artifacts which can be analyzed for desirable properties and successively refined as the development progresses. Once models contain a sufficient amount of detail, complete executable code is generated automatically. MDD has been used successfully in, e.g., the telecommunication and automotive domains. An example of a commercial MDD tool is IBM's RSA-RTE (a previous version was called IBM RoseRT) which uses UML-RT [61], a variant of UML specifically designed to allow the modeling of real-time, embedded systems, as a modeling language.

Although MDD has already led to some productivity gains, current MDD tools are lacking in their support for sophisticated model analyses that allow developers to analyze models directly with respect to relevant properties. For instance, in IBM RSA-RTE models cannot be analyzed directly for behavioral properties such as the reachability of user-specified states or the dropping of messages. Moreover, models cannot even be executed directly which, e.g., complicates model debugging. Instead, any debugging and analysis activity in RSA-RTE requires the generation of code from the models which slows down debugging and unnecessarily complicates the implementation of analyses meant to establish model-level properties.

The context of this thesis is an ongoing project with IBM Ottawa to improve the model analysis capabilities of RSA-RTE. Unfortunately, this task is complicated by the fact that UML-RT currently does not have a formal semantics. To formalize UML-RT and enable model analysis at the same time, the project proposes to

1. automatically translate UML-RT models into an intermediate language that allows easy expression of UML-RT models, has a formal semantics, is substantially more high-level than C/C++ or Java, and enables model analyses, and to

2. devise and implement non-trivial analyses of UML-RT models expressed in this intermediate language [15].

As intermediate language, a language called kiltera [53] has been chosen. kiltera is based on a real-time extension of the π-calculus [41]. kiltera can model real-time and distributed systems. A kiltera model is a complete executable program written in kiltera. The reasons kiltera has been chosen include:

- kiltera supports concurrent, real-time, mobile and distributed computation [53, 56].
- kiltera models are executable.
- kiltera has a formal semantics, which makes the kiltera models analyzable.

The purpose of this thesis is to perform step 2) of the project and to describe the design and implementation of suitable analyses on kiltera models. To do analysis on kiltera models, we have developed an analyzer called π_{klt} *analyzer*. We will present an analyzer that takes a kiltera model as an input, performs various analyses on the model and then reports the output of the analysis.

There has been a lot of research related to process algebras. Mathematical frameworks to reason about concurrent programs were invented independently by C.A.R. Hoare and R. Milner. Hoare came up with the calculus of *Communicating Sequential Processes* (CSP) [25] which offers a mathematical foundation to develop concurrent systems. Milner's *Calculus of Communicating Systems* (CCS) [39] is similar to CSP but focuses on the operational semantics of the language. These two developments have been regarded as one of the foundations of analysis of concurrent systems. These calculi are process-based, where a process is a system with some behavior. Researchers all around the world have extended both CCS and CSP and developed programming languages with suitable syntaxes based on these. Later, Milner also developed the π-calculus, an extension of CCS which supports channel-mobility. A real-time extension to the π-calculus was proposed in [53] and which led to the development of kiltera. The core of kiltera is called the π_{klt} calculus [15]. Our analyzer works on π_{klt} and thus performs analysis on the core of kiltera.

We did not choose any of the other existing process algebras because they lacked at least one of the features we needed. For example, UML-RT supports the dynamic binding of ports to channels. In order to faithfully describe this, the target language should support channel mobility, in addition to real-time behavior. None of the alternatives considered provided suitable support for these features.

On the tools side, there exist other tools to perform analysis on CSP or CCS models. In our work, inspiration was provided by the following analysis tools.

Process Behavior Explorer (ProBE) [16] is an interactive tool offered by Formal Systems (Europe) Limited [18]. The input process is written in CSP and ProBE allows the user to choose the next action of the input process. It then generates the next states. Users can watch the process and its evolution. The interactive mode of our π_{klt} analyzer is similar. However, a problem with ProBE is that the input program can only be written in CSP. Since CSP does not support channel mobility, the expression of UML-RT models becomes cumbersome. Moreover, the tool only provides support to view the trace. It can not perform any analysis on the trace. In contrast, our π_{klt} analyzer supports reachability and timing analysis based on bounded or exhaustive search.

Failures-Divergences Refinement (FDR) [59] and subsequently FDR2 are two other tools from Formal Systems (Europe) Ltd [18]. FDR and FDR2 are tools for refinement checking of models expressed in CSP. FDR2 is a model checker which allows the verification of properties using refinement. A refinement is a relation between two models. Instead, our approach to analysis is based on first creating the state space of a process and then examining it with respect to certain properties.

Perhaps the most closely related tool is the Concurrency WorkBench (CWB) [11, 12]. This is a tool for verifying systems written in CCS. CWB supports a wide range of analysis on the input processes. However, CWB cannot be used to analyze kiltera models.

More related work is discussed in Chapter 6.

1.1 Summary of contribution

The main contribution of this thesis is the design and implementation of an analyzer for kiltera models. The analyzer supports several kinds of analysis, such as:

- finding deadlocks
- finding stable states (i.e., states which do not have an internal action)

- finding states that match a user-specified description which either describes the state fully or just mentions triggers and receivers
- timing analysis
- analysis of open and closed systems

Other contributions include:

- the identification and implementation of certain transformations (the simplification of models both in pre-processing and during analysis) of models which speed up the analysis. These transformations are standard in the process algebra literature, but they have not been applied to the automated analysis of kiltera.
- the removal of redundant renaming of a process to avoid the capture of free variables
- the analyzer's applicability to do analysis of a subset of UML-RT models.

1.2 Organization of thesis

The rest of the thesis is organized as follows. We begin by providing some basic background on kiltera and design patterns with suitable examples in Chapter 2. Chapter 3 provides the detailed description of our analyzer and Chapter 4 describes the implementation. We provide case studies on suitable models and show the analyses output of our π_{klt} analyzer in Chapter 5. Chapter 6 describes the related work. Finally, in Chapter 7 we discuss the limitations of our analyzer and provide some directions to extend the analyzer by improving the performance and capabilities.

Chapter 2

Background

In this chapter we will go through some topics which are useful for understanding the thesis. The readers are required to gather some knowledge about the constructs of kiltera and have some knowledge of composite and visitor patterns. The thesis is based on the core of kiltera and implemented using the above mentioned patterns. In the last section of this chapter we present a mapping of kiltera syntax to the syntax of the π_{klt} calculus.

2.1 kiltera

kiltera is a language based on the π-calculus. It can model complex, interactive, reactive, timed systems. Execution of a kiltera program using the kiltera simulator will produce a trace containing the interactions between the processes. In this section we will describe some of the core constructs of kiltera that are required to understand the semantics of π_{klt}, which is the calculus that kiltera is based on.

A note about kiltera's syntax: kiltera's syntax is indentation sensitive like Python. This means, to run the example kiltera code in this chapter, indentation is necessary. In contrast, π_{klt}'s syntax is not indentation sensitive.

2.1.1 Process terms

A kiltera system consists of some processes. A process is a component that has behavior. All the processes can have some possible actions, by means of which a process can interact with other processes. The interaction can be both internal or external.

2.1.1.1 Termination

The simplest process is `done`. This process does nothing. It represents a terminated process.

2.1.1.2 Events

Two processes can communicate with each other via `events`. An event is a piece of information that can be triggered (or sent). A process can trigger an `event` and can also react to an `event`. To trigger an `event` in kiltera we write:

```
trigger u
```

The process reacting to the triggering event `u` has the following form:

```
when u →
  P
```

where `P` is the continuation process. That means, after `u` has occurred, `P` is the new state of the process. Triggering event `u` can also be viewed as sending a message over channel `u`. In the case of `trigger u` the message is empty. To send a message with data, we use

```
trigger u with data
```

The `data` can be received using the `when` construct as follows:

```
when u with x →
  print x
```

This statement causes `x` to be bound to the `data` that is sent over channel `u` and then prints it. `data` can be a basic value (such as numbers, boolean or strings) or tuples, e.g.,

```
trigger u with ("price", 140)
```

When the event `u` is received by the following process it will try to match the received `data` against the pattern. If the match is successful, `value` is bound to 140 and then printed.

```
when u with ("price", value)→
  print value
```

Pattern matching will be described in more detail later in the section. In general, a listener construct has the following form:

```
when channel with data after variable →
  P
```

where

channel is the name of the channel, *data* is the message received over the *channel*, *variable* is the elapsed time variable and `P` is the process to be executed after the *data* has been received.

In kiltera's operational semantics, the sending of an event (possibly with data) and a matching reception give a rise to a so called τ-action. τ- actions represent internal actions that no process in the environment can participate in.

2.1.1.3 Sequential and parallel compositions

In kiltera, processes can execute sequentially or in parallel. The sequential composition has the form:

```
seq
  P1
  P2
  ⋮
  Pn
```

where P1, P2, ... , Pn are processes. Here processes will execute in order, meaning P1 will always execute before P2, P2 before P3 and so on. P2 will not be executed if the execution of P1 is not finished.

Parallel composition has the following form:

```
par
  P1
  P2
  ⋮
  Pn
```

where P1, P2, ... , Pn are processes. Here, execution of the processes is independent of the order in which they are declared.

2.1.1.4 Process definitions

In kiltera a process definition has the following structure:

```
process A[x1,x2,..., xn]:
  P
```

where A is the name of the process, x1,x2,...,xn are the parameters and P is the body of the process. The parameters can be thought of *ports* of a process, by which the process can interact with other processes. A process definition is valid within some scope. Thus, a valid process definition has the form:

```
process A[x1,x2,...,xn]:
  P
in
  ...
  A[u1,u2,...,un]
  ...
```

Here, we define a process named A with some parameters (parameter list can be empty). Within the process we define the body, which in this case is P. The body is also a process. We are calling process A with some actual parameters (u1,u2,...,un) which will be bound to the formal parameters (x1,x2,...,xn). This can also be referred to as process instantiation.

Let us look at a small example of how two processes can interact. The processes are represented as the round cornered rectangular boxes. Each of the processes has its own ports located on the boundary. Assume a process named Server. This process sends a message to another process named Client. In kiltera the processes can be written as:

```
1 process Server[x]:
2   trigger x
3   done
4 process Client[y]:
5   when y →
```

```
6       done
7   in
8     event z in
9       par
10        Server[z]
11        Client[z]
```

Here we define two processes named `Server` and `Client` with a single parameter. In the scope of the process definitions, we create a new channel `z` and call the defined processes (`Server` and `Client`) with `z` as parameter in parallel. Figure 2.1 illustrates the resulting system.

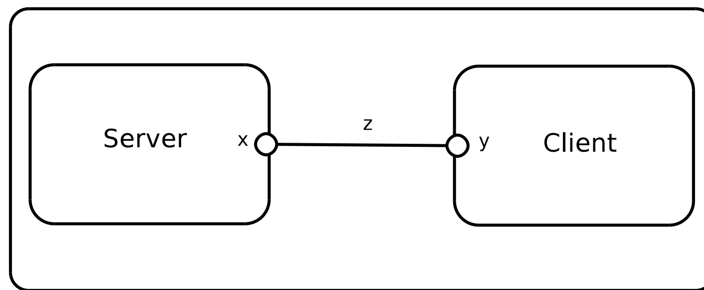


Figure 2.1: Server and client example

The port of the `Server` process (`x`) and the port of the `Client` process (`y`) are now hooked up with the new channel `z`. If the `Server` process triggers the channel with some data (or without any data), the `Client` process can receive the information through the channel `z`.

Process definitions can be nested, meaning, we can declare a new process within a process. An example of a nested process definition is the following:

```
process A[x1]:
  process B[y1]:
    process C[]:
      P3
  in
```

```
    P2
in
  P1
```

Here, we declared process `B` in process `A` and process `C` in process `B`. The child process (the process declared under a process) can have its own ports. It can also access the ports of all ancestor processes. In the above example process `B` has access to `x1` and `y1`. Process `C` also has access to both `x1` and `y1`.

2.1.1.5 State variables

While defining any process we can also declare a special variable called *State Variable*. This is similar to the parameter passing construct of function definitions in programming languages. A state variable can be a number, a string or an arithmetic expression. A process definition with state variables has the following form:

```
process A[x1,x2,...,xn](a1,a2,...,am):
  P
```

where `a1`,`a2`,...,`am` are the state variables. Consider an example where a server waits for an specific amount of time received in a state variable:

```
1 process Server[x](delay_time):
2   wait delay_time →
3     trigger x
4       done
5 in
6   event u1 in
7     Server[u1](10.0)
```

Here we declare a process named `Server`, and in the scope of the process we create a channel named `u1`. Now, we create an instance of the process causing `x` to be bound to `u1`. We set the delay to be 10.0 seconds. When we create the instance of the process `Server`, `delay_time` is bounded to that initial value. Now, the `Server` process will wait for 10 seconds and then trigger the channel `x`.

2.1.1.6 Time

Time plays an important role in kiltera. In kiltera every event and action occurs at some point of time. This time can be real time or simulation time. The kiltera construct to let the time pass is:

```
wait e →
  P
```

This process will wait for `e` amount of time (simulation or real time, based on modes selected by user), and then proceed to `P`. In kiltera's operational semantics, the passage of time is expressed by an *evolution step*. It is also possible to trigger an event after a certain amount of time:

```
schedule a after e
```

which is the short-hand for:

```
wait e →
  trigger a
```

The other important time construct is associated with a listener :

```
when u after e →
  P
```

Once `u` is triggered, the variable `e` is bound to the elapsed time (elapsed time between the beginning of the execution of the listener and the occurrence of a matching trigger). `e` can also occur anywhere in `P`. This construct allows one to measure the passage of time and determines the behavior of the process `P` based on the elapsed time `e`.

2.1.1.7 Conditionals and pattern matching

Like other programming languages, kiltera also supports conditionals, and a special construct called pattern matching. Conditionals have the following form:

```
if condition then P1 else P2
```

where `condition` is a boolean expression and `P1` and `P2` are processes. If the expression evaluates to True, `P1` is executed, otherwise `P2` is executed.

Pattern matching is a special construct in kiltera that has the following form:

```
match expr with
  F1 →
    P1
| F2 →
    P2
⋮
| Fn →
    Pn
```

where `expr` is an expression, `F1`,`F2`,...,`Fn` are patterns and `P1`, `P2`, ..., `Pn` are processes. Evaluation of `expr` will result in a value, and that value is matched with the patterns. If there is any successful match, the corresponding process will execute.

2.1.2 Example

We are going to describe a detailed example implemented in kiltera. This example will give the reader a partial overview of the language.

2.1.2.1 Problem description

The widget production control system has three components: the controller, the workstation and the robot. The controller starts up the workstation and the robot and shuts down after some time. After starting up, the controller orders the workstation to produce the widget. The workstation replies back to the controller when the widget is built. Then the controller orders the robot to deliver the widget, and the robot acknowledges when it has done so. The entire process continues for about a minute. Let us look in Figure 2.2 to understand the structure of the system.

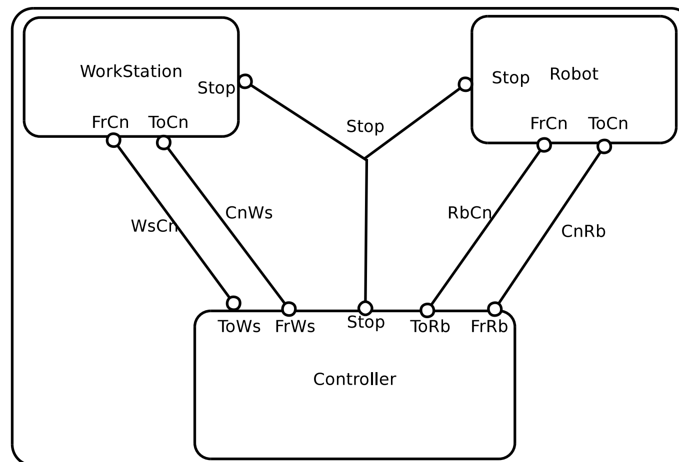


Figure 2.2: Widget production control system

There are three processes in the Widget Production Control System. The `Workstation` process has three ports: `FrCn`, `ToCn` and `Stop`. To communicate with the `Workstation`, the `Controller` has three ports: `ToWs`, `FrWs` and `Stop`. In the scope of the process, we create three channels named `WsCn`, `CnWs` and `Stop`. When we create the instances of the `Workstation` process and the `Controller` process with the new channels as the ports, the ports of the `Workstation` and the `Controller` are connected. In kiltera the processes would look like the following code segment:

```
1  process Controller[ToWs,FrWs,Stop]:
2    P //(Description of the process)
3  process Workstation[FrCn,ToCn,Stop]:
4    Q //(Description of the process)
```

```
5  in
6    event WsCn,CnWs,Stop in
7      par
8        Workstation[WsCn,CnWs,Stop]
9        Controller[WsCn,CnWs,Stop]
```

The ToWs port of Workstation and the FrCn port of Controller are now hooked up through the channel named WsCn. Similarly, we connect the other two ports. The connection between the Robot and the Controller are the same as the connection between the Workstation and the Controller. The Robot and the Workstation can not communicate with each other. So, the whole structure would look like:

```
1  process Controller[ToWs,FrWs,Stop]:
2    P //(Description of the process)
3  process Workstation[FrCn,ToCn,Stop]:
4    Q //(Description of the process)
5  process Robot[FrCn,ToCn,Stop]
6    R //(Description of the process)
7  in
8    event WsCn,CnWs,RbCn,CnRb,Stop in
9      par
10       Workstation[WsCn,CnWs,Stop]
11       Robot[RbCn,CnRb,Stop]
12       Controller[WsCn,CnWs,RbCn,CnRb,Stop]
```

The Controller now starts up the Workstation and the Robot (this is described later as it is done in the scope of the process). Then the Controller sends an instruction to the Workstation to produce a widget. The code segment for this action would be (in the description of controller process):

```
1 //(waits 1 unit of time before producing a widget)
2 process Controller[ToWs,FrWs,Stop]
3   wait 1.0 →
4     trigger ToWs with "Produce Widget"
```

When the `Workstation` process receives the instruction through the `WsCn` channel with the message `"Produce Widget"`, it produces the widget, and acknowledges receipt to the `Controller` with another message `"Widget Produced"` . It then creates an instance of its own process. The purpose of this instantiation is to prepare the `Workstation` for receiving further instruction from the `Controller`.

```
1 //in the description of Workstation process
2 process Workstation[FrCn,ToCn,Stop]:
3   when FrCn with Produce Widget →
4     print "Producing widget...."
5     trigger ToCn with "Widget Produced"
6     Workstation[FrCn,ToCn,Stop]
```

When the `Controller` receives the message from `Workstation`, it sends an instruction to the `Robot` process to deliver the widget.

```
1 process Controller[ToWs,FrWs,ToRb,FrRb,Stop]:
2   wait 1.0 →
3     trigger ToWs with Produce Widget
4     when FrWs with Widget Produced →
5       print "Widget built..."
6       trigger ToRb with "Deliver Widget"
```

The `Robot` process will wait until it receives the instruction from the `Controller` to deliver the widget. Then, it sends a message back to the `Controller` saying that the widget has been delivered,

and goes back to the initial state to receive further instruction. The following code segment would do that.

```
1 process Robot[FrCn,ToCn,Stop]:
2   when FrCn with "Deliver Widget" →
3     print "Delivering widget...."
4     trigger ToCn with "Widget Delivered"
5     Robot[FrCn,ToCn,Stop]
```

The `Controller` now receives the message from the `Robot`, and it goes back to the initial state to get another widget.

```
1 process Controller[ToWs,FrWs,ToRb,FrRb,Stop]:
2   wait 1.0 →
3     trigger ToWs with "Produce Widget"
4     when FrWs with "Widget Produced" →
5       print "Widget built...."
6       trigger ToRb with "Deliver Widget"
7       when FrRb with "Widget Delivered" →
8         print "Widget received..."
9         Controller[ToWs,FrWs,ToRb,FrRb,Stop]
```

The whole module continues forever. So, to stop the processes after a certain amount of time, we will now use the `Stop` channel to send the special instruction to the `Robot` and the `Controller`. Before doing that, let us create a container process inside the `Controller`, so that the `Controller` runs for a certain amount of time and then sends the `Stop` instruction to the `Workstation` and the `Robot`.

```
1  process Controller[ToWs,FrWs,ToRb,FrRb,Stop]:
2    process Container[ ]:
3      wait 1.0 →
4        trigger ToWs with "Produce Widget"
5        when
6          FrWs with "Widget Produced" →
7            print "Widget built...."
8            trigger ToRb with "Deliver Widget"
9            when FrRb with "Widget Delivered" →
10             print "Widget received...."
11             Container[ ]
12        | Stop →
13         done
14   in
15 // this is the start up time for the Workstation and the Robot
16     wait 2.0 →
17       par
18         Container[ ]
19         schedule Stop after 10.0.
```

In the scope of the process we first start up the `Workstation` and the `Robot`. Then in parallel, we create the instance of the `Container` process (which has access to all the ports of the `Controller`), and trigger the `Stop` channel after 10.0 seconds. The `Container` waits in parallel to receive a message from `Workstation` and waits for a trigger on the `Stop` channel (which is triggered after 10 seconds). Similarly we update the `Workstation` and the `Controller` processes to receive the special instruction `Stop` from the `Controller`. So, the whole program segment would look like:

```
1  module WidgetProductionSystem:
2  process Controller[ToWs,FrWs,ToRb,FrRb,Stop]:
```

```
3      process Container[ ]:
4        wait 1.0 →
5          trigger ToWs with "Produce Widget"
6          when
7            FrWs with "Widget Produced" →
8              print "Widget built..."
9              trigger ToRb with "Deliver Widget"
10             when FrRb with "Widget Delivered" →
11               print "Widget received..."
12               Container[ ]
13          | Stop →
14             done
15     in
16 // this is the start up time for the Workstation and the Robot
17       wait 2.0 →
18         par
19           Container[ ]
20           schedule Stop after 10.0.
21   process Workstation[FrCn,ToCn,Stop]:
22     when
23       FrCn with "Produce Widget" →
24         print "Producing widget...."
25         trigger ToCn with "Widget Produced"
26         Workstation[FrCn,ToCn,Stop]
27    | Stop →
28        done
29   process Robot[FrCn,ToCn,Stop]:
30     when
31       FrCn with "Deliver Widget" →
```

```
32          print "Delivering widget...."
33          trigger ToCn with "Widget Delivered"
34          Robot[FrCn,ToCn,Stop]
35      | Stop →
36          done
37  in
38    event WsCn,CnWs,RbCn,CnRb,Stop in
39      par
40          Workstation[WsCn,CnWs,Stop]
41          Robot[RbCn,CnRb,Stop]
42          Controller[WsCn,CnWs,RbCn,CnRb,Stop]
```

If we run the program, the output will:

```
Producing widget....
Widget built...
Delivering widget....
Widget received...
Producing widget....
    ⋮
```

kiltera can also save the trace of the events in a separate file. The trace contains details of the time, action, location, event, data and ports. Some portion of the trace output can be viewed in Figure 2.3.

kiltera's operational semantics defines the mathematical object: a *labelled transition system* (LTS). This object is defined inductively by using inference rules. The description of the object can be found in [53].

time	location	action	port	event	data	position	site	id
3.000	WidgetProductionSystem.Controller.Contai	trigger	ToWs	WsCn	Produce Widget	(9,6)	local	1
3.000	WidgetProductionSystem.Workstation	reaction	FrCn	WsCn	Produce Widget	(42,4)	local	1
3.000	WidgetProductionSystem.Workstation	print	None	None	"Producing widget...."	(44,6)	local	1
3.000	WidgetProductionSystem.Workstation	trigger	ToCn	CnWs	Widget Produced	(46,6)	local	1
3.000	WidgetProductionSystem.Controller.Contai	reaction	FrWs	CnWs	Widget Produced	(13,8)	local	1
3.000	WidgetProductionSystem.Controller.Contai	print	None	None	"Widget built..."	(15,10)	local	1
3.000	WidgetProductionSystem.Controller.Contai	trigger	ToRb	RbCn	Deliver Widget	(17,10)	local	1
3.000	WidgetProductionSystem.Robot	reaction	FrCn	RbCn	Deliver Widget	(58,4)	local	1
3.000	WidgetProductionSystem.Robot	print	None	None	"Delivering widget...."	(60,6)	local	1
3.000	WidgetProductionSystem.Robot	trigger	ToCn	CnRb	Widget Delivered	(62,6)	local	1

Figure 2.3: Trace output for widget production controller system

The kiltera simulator can execute the model described in kiltera. The execution of the model is based on event-scheduling. There are two simulation modes in kiltera's simulator: logical time mode and real time mode. More details of the kiltera language can be found in [53, 56].

2.2 Design patterns

In the previous section, we discussed the kiltera language constructs with examples. Now we will discuss the design concepts behind the implementation of kiltera's analyzer. The implementation structure uses several software Design Patterns [20].

Design patterns are solutions for common, recurring problems in software design. They are not a finished product or code that can be used directly. Design patterns are actually templates for solving problems and can be applied in different situations. There are varieties of patterns developed for different purposes. The implementation of the core of kiltera uses a few design patterns. In this section we will briefly describe the composite and the visitor design pattern to give an overview of how the patterns work.

2.2.1 Composite pattern

The Composite pattern is a Structural Pattern that allows the user to treat an individual object and the composition of objects uniformly. The intent of composition is to create a tree structure of objects to represent part-whole hierarchies. In object-oriented programming, a composite object is

an object designed as a composition of one or more similar objects, all exhibiting similar functionality. As the instances have similar behavior, the programmer can ignore the difference between the single object and the composition of objects. This pattern is particularly useful when the program has a tree structure of objects, and has to deal with different branches or leaves of the tree.

To clarify of how the composite pattern works, let us look at a small example. Let us think of a simple expression that can be any integer or a binary operation (addition in this case) or a function application. The BNF for the expression would be:

$$
\begin{array}{lcll}
E & ::= & n & \text{Number} \\
 & | & E + E & \text{Binary Operation} \\
 & | & f(E) & \text{Function Application}
\end{array}
$$

We now want to evaluate an expression of the form:

```
E = 1 + f(2+3)
```

The Abstract Syntax Tree (AST) for the expression can be seen in Figure 2.4. We can use the Composite pattern to represent any expression generated using the BNF above.

We create an abstract class `Expr` that has an operation called `eval()`. This abstract class does nothing, but this will be the parent of the other classes named as `BinOp` (for binary operation), `Num` (for integer numbers) and `FuncApp` (for function application). The summary of the classes and their attributes and operations can be seen in Table 2.1.

Name of the Class	Attributes	Operations
`Expr`		`eval()`
`Num`	`value`	`eval()`
`BinOp`	`left,right`	`eval()`
`FuncApp`	`name,argument`	`eval()`

Table 2.1: Classes and their attributes and operation

The application of the composite pattern for the BNF can be seen in the UML class diagram in Figure 2.5. From the composite pattern we can see that a `BinOp` class can have two expressions which are stored in attributes `left` and `right` respectively. `FuncApp` can have a function name and arguments which are expressions as well.

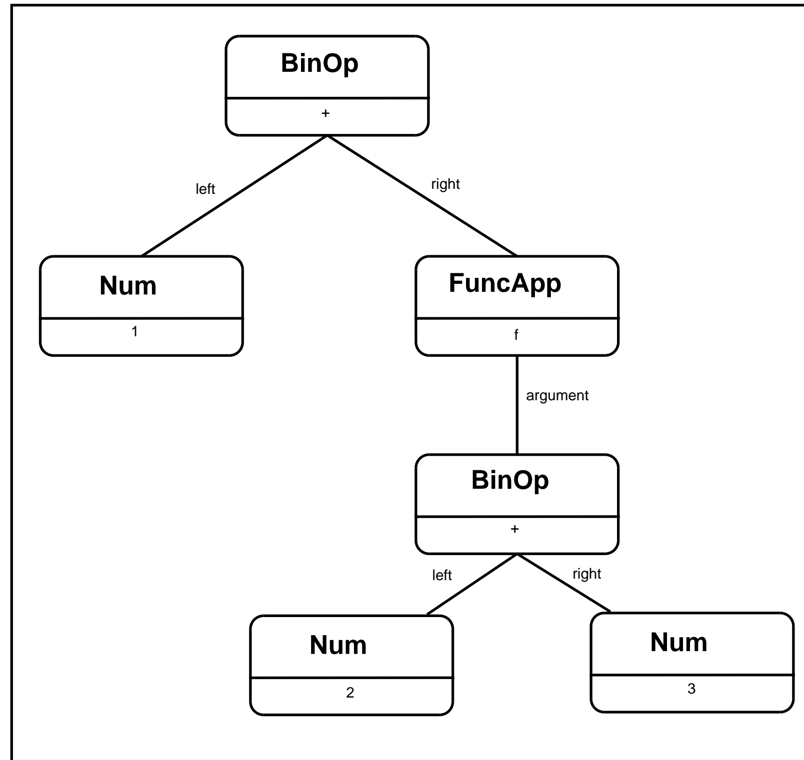


Figure 2.4: Abstract syntax tree for example expression

2.2.2 Visitor pattern

The visitor pattern provides a way to traverse composite structures. This pattern is particularly useful if we want to extend the functionality of any composite structure. If we look at the example BNF described in Subsection 2.2.1, we can see that the abstract syntax was created to evaluate the expression generated by the BNF. Say, we now want to print the expression first as output (pretty print) and then evaluate the expression to show the result to the user. If we want to do that differently, we will simply need two abstract syntaxes. But it would be redundant to create a new syntax just for different functionalities of the same syntax. And this cost will become huge if the functionalities grows in number. The most obvious way to solve this problem is to use the visitor pattern. If we have an abstract class for the visitor we can derive each visitor from the base class for different functionality. To use the same composite structure for different visitors (different functionality), we need to create a way that each visitor can use the same abstract syntax. Therefore,

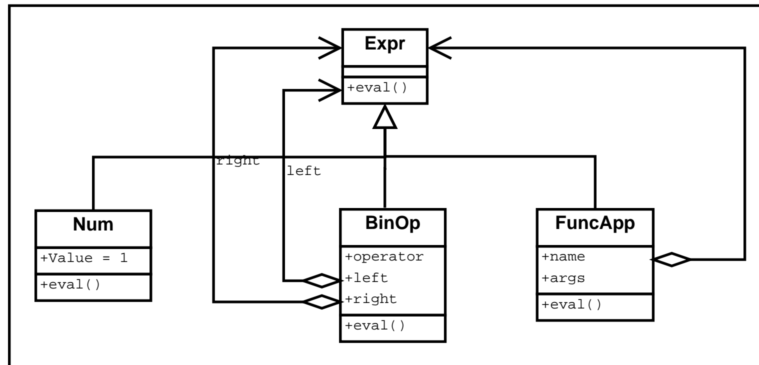


Figure 2.5: Composite pattern for example BNF

in the abstract syntax we have an `accept()` method that receives the visitor as an argument. Each element in the abstract syntax then calls the visitor from which it was invoked. Each of the visitor has several `visit()` functions based on the abstract syntax to implement the different functionality.

In our example, we have two different visitors. Each of the visitors (two in this case) is derived from a base visitor class. Before going into the implementation of two above-mentioned visitors, let us change the composite pattern as depicted in Figure 2.6.

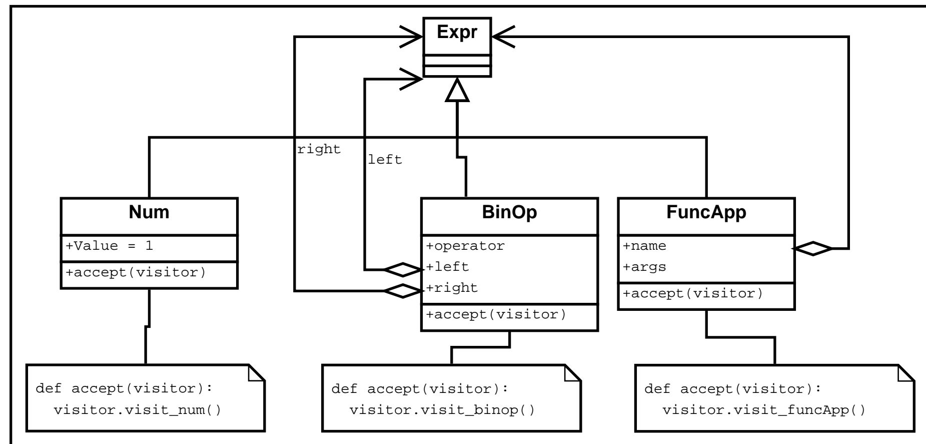


Figure 2.6: Composite pattern for example BNF

Earlier we had only one function in each of the classes called `eval()` that could evaluate the expression for the example BNF. As we want to use the composite structure for two different visitors (pretty print and evaluate), we create the function `accept()` that will direct control towards the

visitor from which the `accept()` function is invoked (the visitor is received as an argument).

We use two different visitors: one is used to pretty print the expression and the other visitor is used to evaluate the expression. The visitor pattern structure can be viewed in Figure 2.7.

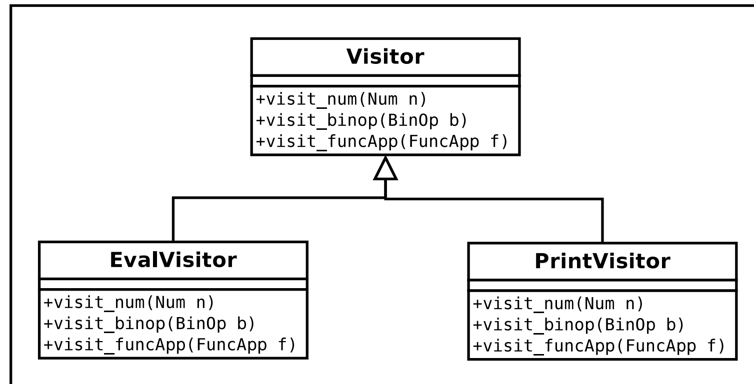


Figure 2.7: Visitor pattern structure for example BNF

We can use the `EvalVisitor` to evaluate the expression and `PrintVisitor` to print the expression. Each of the visitors has three visit functions as we have three subclasses in the abstract syntax. The function `visit_num(Num n)` in `EvalVisitor` will evaluate the number and in `PrintVisitor` will print the number. `visit_binop(BinOp b)` will take two arguments of the `BinOp` instance and add them (`EvalVisitor`) or print them (`PrintVisitor`). Let us look at the Python snippets to clarify the composite pattern and the visitor pattern.

The composite pattern for the example BNF can be represented in Python2.6 as:

```
1 # Abstract syntax (composite pattern) for example BNF
2 class Expr: # this is the abstract class
3   pass # pass statement does nothing
4
5 class Num(Expr): # inherited from Expr class
6   def __init__(self, num): # constructor function
7         self.num = num
8   def accept(self, visitor):
9         return visitor.visit_num(self)
10
11 class BinOp(Expr): # class for binary operation
```

```
12   def __init__(self,left,right):# left and right are also expressions
13         self.left = left
14         self.right = right
15   def accept(self, visitor):
16         return visitor.visit_binop(self)
17     ⋮
```

We create the abstract visitor first and then we inherit the two visitors from the base class:

```
1 class Visitor: # abstract visitor class
2   def visit_num(self):
3         pass
4   def visit_binop(self):
5         pass
6   def visit_funcapp(self):
7         pass
```

We can now implement two different functionalities using two different visitors. The `EvalVisitor` evaluates the expression:

```
1 class EvalVisitor(Visitor): # inherited from Visitor class
2   def eval(self, expr): # the expression we want to evaluate
3     return expr.accept(self)
4   def visit_num(self,expr):
5     return expr.num # returns the number
6   def visit_binop(self, expr):
7         return expr.left.accept(self)\
8          + expr.right.accept(self) #left and right are both expressions
9     ⋮
```

Similarly for pretty printing the expression, we can implement the `PrintVisitor` without any modification to our existing abstract syntax:

```
1 class PrintVisitor(Visitor):
2   def pprint(self,expr): # the expression we want to print
3         return expr.accept(self)
4   def visit_num(self,expr):
5         print expr.num
6   def visit_binop(self,expr):
7         print expr.left.accept(self)+ '+' + expr.right.accept(self)
8     ⋮
```

If we now create an expression and want to prettyprint and evaluate it, we can do the following:

```
1 number1 = Num(5) #An instance of Num class with the value
2 number2 = Num(6)
3 expression = BinOp(number1,number2) #An instance of BinOp class
4 #with the values we want to add
5 expr_print = PrintVisitor() # An instance of PrintVisitor class
6 expr_eval = EvalVisitor() # An instance of EvalVisitor class
7 print expr_print.pprint(expression) # Print the Expression
8 print expr_eval.eval(expression) # Evaluate the Expression
```

If we look at Figure 2.8, we will understand how the entire function invocations works. The numbers on the arrows indicate the order of the interaction between the functions. Details about patterns with examples can be found in [20].

2.3 π_{klt} syntax

π_{klt} is the core of kiltera. The π_{klt} syntax is more compact than that of kiltera which increases readability and it is also closer to the syntax of the π-calculus. Table 2.2 summarizes the mapping of kiltera terms into π_{klt} terms. Details of the mapping can be found in [55, 54].

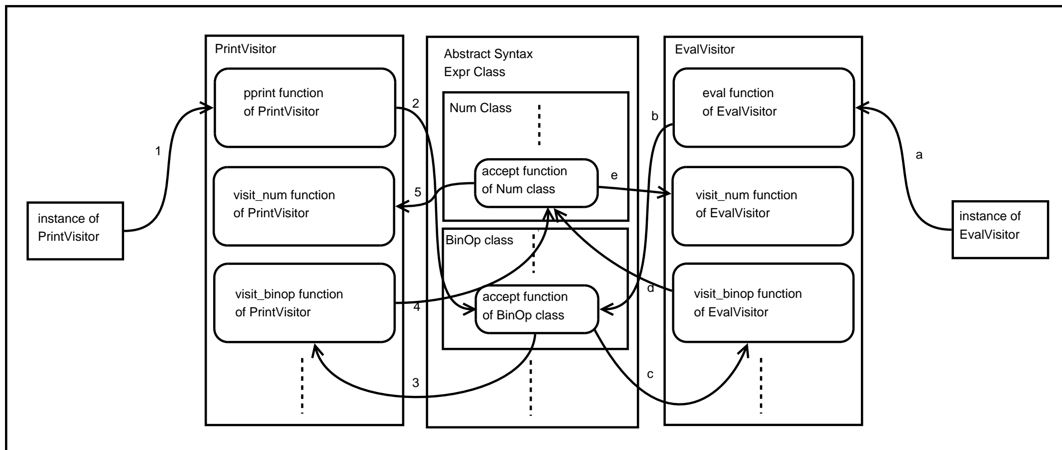


Figure 2.8: Control flow for function invocations

<table>
<tr><th>kiltera Syntax</th><th>π_{klt} Syntax</th></tr>
<tr><td>done</td><td>√ or nil</td></tr>
<tr><td>trigger x</td><td>x!</td></tr>
<tr><td>trigger x with E</td><td>x!E</td></tr>
<tr><td>wait E →P</td><td>ΔE →P
or
wait E →P</td></tr>
<tr><td>event x in P
or
channel x in P</td><td>νx.P
or
new x in P</td></tr>
<tr><td>par
P1
P2
⋮
Pn</td><td>P1 || P2 || . . . || Pn</td></tr>
<tr><td>seq
P1
P2
⋮
Pn</td><td>P1; P2; . . . ;Pn</td></tr>
<tr><td>when x1 with F1 after t1 ->
P1
| x2 with F2 after t2 -> ...
| xn with Fn after tn ->
Pn</td><td>when{
x1?F1@t1→P1
| x2?F2@t2→P2
⋮
| xn?Fn@tn→Pn}</td></tr>
<tr><td>A[x1,x2, . . . xn](s1,s2,. . . sm)</td><td>A(x1,x2, . . . xn,s1,s2, . . . sm)</td></tr>
<tr><td>process A[x1,x2,. . . ,xn]:
P1
process B[y1,y2,. . . ,yn]:
P2
in
P</td><td>def{d1,d2,. . . ,dn}in P
where,
$d_1 \stackrel{def}{=}$ proc A(x1,x2,. . . ,xn) = P1
$d_2 \stackrel{def}{=}$ proc B(x1,x2,. . . ,xn) = P2
etc.</td></tr>
</table>

Table 2.2: π_{klt} syntax

Chapter 3

Analyses

In this chapter we are going to describe the analysis framework of the π_{klt} analyzer. The next section contains a detailed description of the π_{klt} analyzer framework. As the purpose of the thesis is to analyze kiltera models, we will describe how the analyzer works on these models.

Our π_{klt} analyzer takes kiltera models as input. The Aperiot [52] parser then generates the *Abstract Syntax Tree* (AST). Our analyzer generates the state space of the input model, and performs several analyses based on user choices. Users can specify a logical time or depth bound as well as other input parameters as an external initialization script to the analyzer. Figure 3.1 shows the general analysis architecture of our π_{klt} analyzer.

3.1 π_{klt} analyzer: how does it work?

Given a kiltera model, the parser generates the AST, where each node is an instance of the `ProcTerm` class, i.e, a π_{klt} term. Upon receiving the AST, the analyzer does some pre-processing on the AST and generates the state space tree. The pre-processing step includes several operations on the AST namely, *Deep Simplification, Renaming* and *Scope Extrusion. Deep Simplification* recursively applies some simplification rules on the π_{klt} terms. *Renaming* renames the bound names of a process by fresh names that are unique. *Scope Extrusion* safely enlarges the scope of the declaration of a

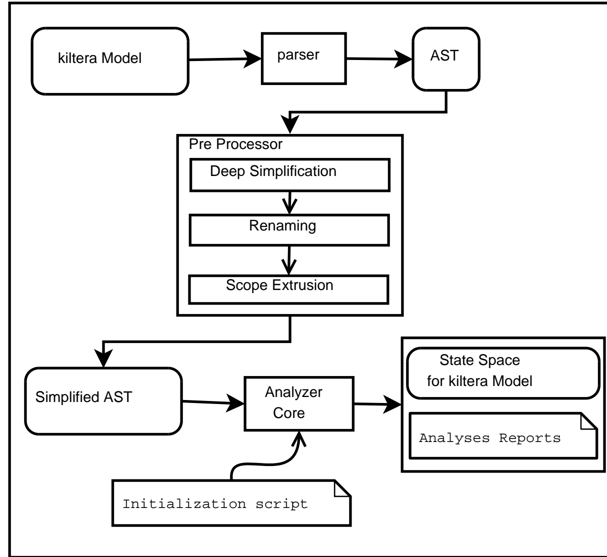


Figure 3.1: π_{klt} analyzer general framework

name using new by moving the declaration to the beginning of the term. These transformations are guaranteed to preserve the semantics of the input term. More specifically, the resulting term is equivalent to the original term according to the rules of Structural Congruence defined in [15].

After the pre-processing is done, the analyzer creates an initial *state* to be included in the *State Space Tree*. A π_{klt} *state* consists of a π_{klt} term and an *environment*. An *environment* is a map of process names to process values (process values contain list of parameters and body of the process). A *State Space Tree* is a data structure that holds the reachable π_{klt} states of a kiltera model. The first state is the *root* of the tree. From the root, the analyzer generates all the *first actions* of a π_{klt} term. *First Actions* are actions that a π_{klt} term can immediately perform. From a π_{klt} state and a first action we will generate the *next set* of π_{klt} states. *Next Set* is a set that holds π_{klt} processes which are generated from a process and a first action. The analyzer creates new states with each of the π_{klt} terms in the next set, and includes them in the tree. Then the analyzer explores the next unvisited state in the tree using standard Breadth First Search [28]. This way, the analyzer generates the state space tree of the kiltera model, which is used to perform various analyses. We can perform analyses such as detection of deadlocks, stable states, specific triggers or receivers, specific set of triggers or receivers, user specified states etc. Most of the analyses are performed during

the generation of the state space tree. Others are performed once the state space tree has been generated. Figure 3.1 provides an overview of how our π_{klt} analyzer works.

3.2 Pre-processor

Upon receiving the AST of a kiltera model, our pre-processor performs some operations on the AST to improve the analyzer performance. The pre-processing step includes three operations: ***Deep Simplification***, ***Renaming*** and ***Scope Extrusion***. The purpose of the pre-processing step is to simplify the AST, so that the state-space generation can be done efficiently.

3.2.1 Preliminary definitions

$\mathcal{T}_{\pi_{klt}}$ is the set of π_{klt} terms or processes. *fn(P)* is a function to calculate the set of free names of a process P. Substitution σ is a function that contains an assignment of names, terms or values to names. To avoid the accidental capture of free variables of a process, we rename the bound variables of a process with *fresh* variables. A variable is *fresh* if that variable is not used in any other process. We write $\{^{u_1}/_{x_1}, \ldots, ^{u_n}/_{x_n}\}$ for the substitution σ, where $\sigma(x_1) = u1$, $\ldots$, $\sigma(x_n) = u_n$ and $\sigma(z) = z$ for all $z \notin \{x_1, \ldots, x_n\}$. A variable is *free* if it is not *bound*. A variable is a *bound* variable if it is within the scope of a `new`, `when` or `proc` construct. For example, if `P` = `new x in x!5` is a process, the variable `x` in the scope of `new` is *bound*. The *model time* of a state `S` in the state space tree is the sum of the times of the evolution steps along the path from the initial state `S`. In other words, it is the amount of time that needs to pass before state `S` can be reached. Two terms are *alpha-equivalent* if they are equal or one term can be obtained from the other term by substituting the bound names of the first term by those of the second term. If `P` and `Q` are two different π_{klt} terms:

P = νx.x!5

Q = νy.y!5

`P` and `Q` are alpha-equivalent. `Q` can be obtained from `P` by substituting the bound names of `P` by those in `Q` or vice versa. The way to do it is to create a substitution $\sigma = \{^{y}/_{x}\}$ or $\sigma = \{^{x}/_{y}\}$ and

apply the substitution on P$\{^y/_x\}$(substitute `x` by `y`) or Q$\{^x/_y\}$(substitute `y` by `x`) and check if they are equal.

3.2.2 Deep simplification

Deep Simplification is a step to simplify π_{klt} terms. We remove the unnecessary `nil` terms and `new` operators from a π_{klt} term recursively in this step. More precisely, Deep Simplification is a function $ds : \mathcal{T}_{\pi_{klt}} \rightarrow \mathcal{T}_{\pi_{klt}}$ defined as:

$$
\begin{array}{rcl}
ds(P \mid\mid \texttt{nil}) & \stackrel{def}{=} & ds(P) \\
ds(\texttt{nil} \mid\mid P) & \stackrel{def}{=} & ds(P) \\
ds(\texttt{nil}\mid\mid\texttt{nil}) & \stackrel{def}{=} & \texttt{nil} \\
ds(x!E) & \stackrel{def}{=} & x!E \\
ds(\nu x.P) & \stackrel{def}{=} & \begin{cases} ds(P) & x \notin fn(P) \\ \nu x.ds(P) & \text{otherwise} \end{cases} \\
ds(\texttt{when}\{\ldots|x?F@y.P|\ldots\}) & \stackrel{def}{=} & \texttt{when}\{\ldots|x?F@y.ds(P)|\ldots\} \\
ds(P;\texttt{nil}) & \stackrel{def}{=} & ds(P) \\
ds(\texttt{nil};P) & \stackrel{def}{=} & ds(P) \\
ds(\texttt{nil};\texttt{nil}) & \stackrel{def}{=} & \texttt{nil} \\
ds(\texttt{wait}\ E \rightarrow P) & \stackrel{def}{=} & \texttt{wait}\ E \rightarrow ds(P) \\
ds(\texttt{def}\{d_1, d_2, \ldots, d_n\}\ \texttt{in}\ P) & \stackrel{def}{=} & \texttt{def}\{d_1', d_2', \ldots, d_n'\}\ \texttt{in}\ ds(P) \\
\text{where} & & \text{where} \\
d_1 \stackrel{def}{=} \texttt{proc}\ A(x1, x2, \ldots, xn) = P_1 & & d_1' \stackrel{def}{=} \texttt{proc}\ A(x1, x2, \ldots, xn) = ds(P_1) \\
d_2 \stackrel{def}{=} \texttt{proc}\ B(y1, y2, \ldots, yn) = P_2 & & d_2' \stackrel{def}{=} \texttt{proc}\ B(y1, y2, \ldots, yn) = ds(P_2) \\
\text{etc.} & & \text{etc.}
\end{array}
$$

For example, let us consider the following parallel composition of π_{klt} processes:

```
P = x!"data" || nil || y!"data"
```

```
Q = x!"data" || y!"data"
```

Here, the behavior of process P is actually the same as that of process Q. We can simplify process P and represent it as process Q.

3.2.3 Renaming

Renaming is a function $rn(P)$: $\mathcal{T}_{\pi_{klt}} \rightarrow \mathcal{T}_{\pi_{klt}}$ and is defined as follows:

$$
\begin{aligned}
rn(\texttt{nil}) &\stackrel{def}{=} \texttt{nil} \\
rn(x!E) &\stackrel{def}{=} x!E \\
rn(\nu x.P) &\stackrel{def}{=} \nu x'.rn(P\{^{x'}/_{x}\}) \text{ with } x' \text{ fresh} \\
rn(\texttt{when}\{\ldots|x?F@y.P|\ldots\}) &\stackrel{def}{=} \texttt{when}\{\ldots|x'?F\{^{x'}/_{x}\} \\
& \quad @y\{^{x'}/_{x}\}.rn(P\{^{x'}/_{x}\})|\ldots) \text{ with } x' \\
& \quad \text{fresh} \\
rn(P \mid\mid Q) &\stackrel{def}{=} rn(P) \mid\mid rn(Q) \\
rn(P\ ;\ Q) &\stackrel{def}{=} rn(P)\ ;\ rn(Q) \\
rn(\texttt{wait } E \rightarrow P) &\stackrel{def}{=} \texttt{wait } E \rightarrow rn(P) \\
rn(\texttt{def}\{d_1, d_2, \ldots, d_n\} \texttt{ in } P) &\stackrel{def}{=} \texttt{def}\{d_1', d_2', \ldots, d_n'\} \texttt{ in } rn(P) \\
\text{where} & \quad \text{where} \\
d_1 \stackrel{def}{=} \texttt{proc } A(x1, x2, \ldots, xn) = P_1 & \quad d_1' \stackrel{def}{=} \texttt{proc } A(x1, x2, \ldots, xn) = rn(P_1) \\
d_2 \stackrel{def}{=} \texttt{proc } B(y1, y2, \ldots, yn) = P_2 & \quad d_2' \stackrel{def}{=} \texttt{proc } B(y1, y2, \ldots, yn) = rn(P_2) \\
\text{etc.} & \quad \text{etc.}
\end{aligned}
$$

To view the effect of renaming a process, consider process P for example:

```
P = νx.x!"data" || when{u?y→ y!5}
```

P is a parallel composition of two processes. Here x and y are bound and thus hidden from the environment. We can rename the process as:

```
P = νx1.x1!"data" || when{u?y1→y1!5}
```

where, `x1` and `y1` are fresh names (unique names and cannot be used in any other processes). The benefit of this renaming is the avoidance of accidentally capturing free names in a process through substitution. We need renaming for two reasons: a) to implement the semantics of interaction (when an event is received by a listener) and b) to perform transformations like scope extrusion. To illustrate the avoidance of capturing free names, consider a process `P` and two substitutions σ and σ_1 where substitution is a function that contains an assignment of names, terms or values to names:

$$P = \textbf{when}\{u?x \rightarrow (y! \ || \ x!)\}$$
$$\sigma = \{^z/_y\}$$
$$\sigma_1 = \{^x/_y\}$$

Applying substitution σ to `P`, we will get a resulting process `P'`:

$$P' = \textbf{when}\{u?x \rightarrow (z! \ || \ x!)\}$$

where `P` and `P'` contain the same free names. If we apply σ_1 naively to `P'`, we would get:

$$P'' = \textbf{when}\{u?x \rightarrow (x! \ || \ x!)\}$$

If we receive any data over channel `u`, `x` is bound to that data. After receiving the name, we trigger the events twice and thus the behavior of the process `P'` differs from that of process `P"`. We can see that the *free variable* `x` in `P` has been captured by the substitution. The capture of free names can be avoided if we rename the bound names of process `P` with fresh names. Let us create a substitution $\sigma_2 = \{^{x'}/_x\}$ of x with a fresh name x' and apply the substitution σ_2 to `P`. The resulting process will be $P' = \textbf{when}\{u?x' \rightarrow y! \ || \ x'!\}$. Now if we apply σ_1 to `P"` we will get $R = \textbf{when}\{u?x' \rightarrow x! \ || \ x'!\}$ where we avoided the variable `y` from being captured.

3.2.4 Scope extrusion

Scope Extrusion is the last of the pre-processing steps. This step is used to safely enlarge the scope of a name declaration. We apply scope extrusion after applying the renaming on the input model. The renaming guarantees that the extracted variables will not cause free variables of the

other processes to be captured. For example, `P` $=$ `x!5` $||$ ν`x.Q` is a process. If we extract the ν**x** operator from `P`, the free variables of `x!5` (`x`) will be captured. But, renaming guarantees that the free variable `x` in `x!5` will be renamed to a new fresh name. If we then apply the scope extrusion, it is now safe to do so. Formally, Scope Extrusion is a function $sc(P)$: $\mathcal{T}_{\pi_{klt}} \rightarrow \mathcal{T}_{\pi_{klt}}$ is defined as follows:

$$
\begin{array}{rcl}
sc(\texttt{nil}) & \stackrel{def}{=} & \texttt{nil} \\
sc(x!E) & \stackrel{def}{=} & x!E \\
sc(\nu x.P) & \stackrel{def}{=} & \nu x.sc(P) \\
sc(\texttt{when}\{\ldots|x?F@y.(\nu z.P)|\ldots\}) & \stackrel{def}{=} & \nu z.\texttt{when}\{\ldots|x?F@y.sc(P\{^{z'}/_{z}\})|\ldots) \\
sc(P \mid\mid \nu z.Q) & \stackrel{def}{=} & \nu z.sc(P \mid\mid Q) \\
sc(P\ ;\ \nu z.Q) & \stackrel{def}{=} & \nu z.sc(P\ ;\ Q) \\
sc(\texttt{wait}\ E \rightarrow \nu z.P) & \stackrel{def}{=} & \nu z.sc(\texttt{wait}\ E \rightarrow P) \\
sc(\texttt{def}\{d_1, d_2, \ldots, d_n\}\ \texttt{in}\ P) & \stackrel{def}{=} & \texttt{def}\{d_1', d_2', \ldots, d_n'\}\ \texttt{in}\ sc(P) \\
\text{where,} & & \text{where,} \\
d_1 \stackrel{def}{=} \texttt{proc}\ A(x1, x2, \ldots, xn) = P_1 & & d_1' \stackrel{def}{=} \texttt{proc}\ A(x1, x2, \ldots, xn) = sc(P_1) \\
d_2 \stackrel{def}{=} \texttt{proc}\ B(y1, y2, \ldots, yn) = P_2 & & d_2' \stackrel{def}{=} \texttt{proc}\ B(y1, y2, \ldots, yn) = sc(P_2) \\
\text{etc.} & & \text{etc.}
\end{array}
$$

For example, let us look at process `P`:

```
P = νx.x!"data" || νy.y!"data"
```

If we apply scope extrusion to `P`, we would obtain:

```
P = νx.νy.(x!"data" || y!"data")
```

We apply scope extrusion so that all processes are of the form ν`x.P` whenever possible. This scope extrusion can be helpful if we combine it with the alpha-equivalence checking. For example, let us consider two processes `A` and `B`:

```
A = νx.z!5 || z!6
B = νy.(z!5 || z!6)
```

If we do not apply scope extrusion on process `A`, process `A` and process `B` will never be found equivalent. But, actually they are equivalent, and this equivalence can only be achieved if we apply scope extrusion on process `A` before checking for alpha-equivalence. `A` will be $\mathtt{A}' = \nu\mathtt{x}'.\mathtt{(z!6||z!5)}$ after applying scope extrusion. We can now compare between the processes $\mathtt{A}'$ and `B`, and they will be found alpha-equivalent.

This scope extrusion can be applied only up-to process definition. A process definition may invoke itself for recursive definition. In that case, the actual definition would be changed if we apply the scope extrusion on a process definition.

3.3 Analyzer core

The analyzer core is responsible for the creation of state space tree, traversing the tree using standard Breadth First Search (BFS), and perform the *online* or *offline* analysis. We call an analysis *online* if the analysis is done during the generation of the state space tree. Otherwise we call the analysis *offline*.

After receiving the simplified AST, the analyzer first creates a π_{klt} state and puts that state into the state space tree. A π_{klt} state consists of a π_{klt} term and an environment. Given a π_{klt} term, the analyzer computes the set of first actions. There are three possible first actions of any input π_{klt} process: Input actions, Output actions and τ (Internal) actions. The term along with a first action will generate a new next set of π_{klt} terms. We denote the method of generating next states as $\mathtt{next}(term, action)$, where *term* is π_{klt} process and *action* is the first action of that process. The analyzer repeats the process for all the actions in the first actions set. Then, the analyzer creates a state for each term in the next set and adds it to the state space tree.

For example, let `P` be a process which contains a parallel composition of two interactive processes `Q` and `R`:

```
P = Q || R
```

where

```
Q = x!5
R = when{x?data → y!data}
```

`Q` is the process that triggers the event `x` with a numeric value of 5. `R` is another process that listens to the event `x` and then triggers another event `y` with the received data. The first action set of `P` would be {`x!5`, `x?data`, τ} where τ is the internal action (interaction between `Q` and `R`). Now, `P` will generate three new sets of next terms for each of the actions in the first actions set. For example, `P` along with the action τ would generate the following set of next terms:

```
next(P,τ) = {(nil || y!5)}
```

BFS is used to explore each new state and its successor states. The process continues until all the states in the state space tree have been visited. Algorithm 3.1 shows the pseudo code for the interactive mode and Algorithm 3.2 shows the pseudo code for the exhaustive mode. Before presenting these algorithms we introduce the shallow simplification of processes and the π_{klt} state.

3.3.1 Shallow simplification

As soon as a set of next states has been created, we perform a transformation operation called ***Shallow Simplification*** to simplify the terms in the set. Formally, Shallow Simplification is a function $ss(P)$: $\mathcal{T}_{\pi_{klt}} \rightarrow \mathcal{T}_{\pi_{klt}}$ defined as:

$$
\begin{aligned}
ss(\texttt{nil}) &\stackrel{def}{=} \texttt{nil} \\
ss(x!E) &\stackrel{def}{=} x!E \\
ss(\nu x.P) &\stackrel{def}{=} \begin{cases} P & x \notin fn(P) \\ \nu x.P & \text{otherwise} \end{cases} \\
ss(\texttt{when}\{\ldots|x?F@y.P|\ldots\}) &\stackrel{def}{=} \texttt{when}\{\ldots|x?F@y.P|\ldots\} \\
ss(P \mid\mid \texttt{nil}) &\stackrel{def}{=} P
\end{aligned}
$$

$$
\begin{array}{rcl}
ss(\texttt{nil} \mid\mid P) & \stackrel{def}{=} & P \\
ss(\texttt{nil} \mid\mid \texttt{nil}) & \stackrel{def}{=} & \texttt{nil} \\
ss(P \; ; \texttt{nil}) & \stackrel{def}{=} & P \\
ss(\texttt{nil} \; ; P) & \stackrel{def}{=} & P \\
ss(\texttt{nil} \; ; \texttt{nil}) & \stackrel{def}{=} & \texttt{nil} \\
ss(\texttt{wait } E \rightarrow P) & \stackrel{def}{=} & \texttt{wait } E \rightarrow P \\
ss(\texttt{def}\{d_1, d_2, \ldots, d_n\} \texttt{ in } P) & \stackrel{def}{=} & \texttt{def}\{d_1, d_2, \ldots, d_n\} \texttt{ in } P \\
d_1 \stackrel{def}{=} \texttt{proc } A(x1, x2, \ldots, xn) = P_1 & & d_1 \stackrel{def}{=} \texttt{proc } A(x1, x2, \ldots, xn) = P_1 \\
d_2 \stackrel{def}{=} \texttt{proc } B(y1, y2, \ldots, yn) = P_2 & & d_2 \stackrel{def}{=} \texttt{proc } B(y1, y2, \ldots, yn) = P_2 \\
\text{etc.} & & \text{etc.}
\end{array}
$$

Shallow simplification differs from deep simplification in that shallow simplification does not remove the `nil` terms or `new` terms recursively. After simplifying a set of terms with the *shallow simplification* we create a π_{klt} state for each term in the set, and add them to the state space tree.

3.3.2 π_{klt} state

A π_{klt} state consists of a π_{klt} term and an environment. An *environment* is a map from process names to process values. A process value is an object that contains:

- a possibly empty list of parameters of the process
- the body (a process term)

For instance, let us consider a process `P` which is a process definition:

```
P = def{proc A(x) = x!5} in A(y)
```

We create the first state which has the term `P` and an environment ρ. ρ is initially empty, so we represent is as ρ_0. The next state would be:

$\texttt{next((P,}\rho_0\texttt{),}\tau\texttt{)} = \{\texttt{A(y)},\ \rho_1\}$

where

$\rho_1 = \{\texttt{A} \mapsto \lambda x.x!5\}$

where, `x` is the parameter and `x!5` is the body. The environment ρ_0 is extended to ρ_1 with the definition of process `A`. The next state of the newly generated state is:

$\texttt{next((A(y),}\rho_1\texttt{),}\tau\texttt{)} = \{\texttt{(y!5},\ \rho_1\texttt{)}\}$

As we do not have any process definition in this step, the environment remains the same. This way the analyzer generates the complete state space tree for a given kiltera model. After the state space has been created, the analyzer performs several analyses and produces reports. During the generation of the state space tree we perform some analyses such as searching for a specific trigger and receiver, timing analysis, etc. We describe the analyses in the Subsection 3.4 and mention if the analysis has been done *online* or *offline*.

3.4 Analyses modes

Our analyzer has three modes of analyses:

- Interactive mode
- Exhaustive mode
- Bounded mode

3.4.1 Interactive mode

Interactive mode lets the user choose an action from the set of first actions of the input process and generates the next set of states. A π_{klt} term with an action may generate more than one next state. The user can choose any of the next states from the set and follow the evolution of the input process. Algorithm 3.1 describes the interactive mode of the analyzer.

Algorithm 3.1 Interactive mode

```
1  P:= input model, done = False
2  while(not done)
3    first_action_set := firstAction(P)
4    if first_action_set is empty:
5       done := True
6    else: #first_action_set is not empty
7      user_chosen_action := prompt(first_action_set)
8      next_state_set := next(P, user_chosen_action)
9      if next_state_set is empty:
10       done := True
11     else: #next_state_set is not empty
12       user_chosen_term := prompt(next_state_set)
13       P:= user_chosen_term
14     end if
15   end if
16 end while
```

3.4.2 Exhaustive mode

Exhaustive mode of analysis runs until all the states have been visited in the tree in BFS manner. There is no termination condition in exhaustive mode except that we check if all the states generated have been visited. As soon as a new state is generated it is compared with those have been seen before. If the state is equal or alpha-equivalent [54] to any of the states in the seen set, it is not added to the state space tree. This equivalence checking guarantees the termination of the search when all the states have been visited. Algorithm 3.2 describes the Exhaustive mode of the analyzer.

The state contains a term and an environment. The queue is used to traverse the state space tree using standard BFS. The *seen_before* table is a hash table that is used to store the states that have already been visited. The hash table uses the length of the process (in the state) as hash key.

3.4.3 Bounded mode

The ***Bounded mode*** of analysis is a variant of the *Exhaustive mode* of analysis. There are two termination conditions in this mode: Time-bounded or Depth-Bounded or Both. In time-bounded mode, the model time of each explored state is compared with the bound set by the user in the initialization script. If the model time of the current state exceeds this bound, the algorithm terminates. In depth-bounded mode, the depth of each explored state is compared with the depth bound set by the user in the initialization script. If the depth of the current state exceeds this bound,

Algorithm 3.2 Exhaustive mode

```
1  P:= input model, tree := empty tree, seen_before := empty set, queue := empty
2  env := empty
3  state: = (P, env)
4  tree.add(state)
5  queue.add(state)
6  seen_before.insert(state)
7  while(queue not empty and no termination condition):
8    current_state = queue.pop()
9    first_action_set := firstAction(current_state.term)
10   for action in first_action_set:
11     next_state_set := next(current_state, action)
12     for state in next_state_set:
13       if state %is% in seen_before: # alpha-equivalence checking
14           discard state
15       else:
16           tree.add(state)
17           queue.add(state)
18           seen_before.insert(state)
19       end if
20     end for
21   end for
22 end while
```

the algorithm terminates. If the user sets both modes as the terminating conditions, the algorithm terminates as soon as it satisfies any of the conditions.

3.5 Open and closed systems analysis

In all the analysis modes the user can choose to perform on open or a closed system analysis. The open system analysis considers all the first actions of a input process whereas the closed system takes into account the τ actions only. τ actions represent internal actions, so in a closed system analysis we are considering the system's behavior independently from its environment. Both modes have their own benefits: The open system shows the user all the states that a kiltera model has. This mode is able to perform a truly exhaustive search, but may take a long time to complete. Closed system analysis takes less time because it typically generates smaller numbers of states.

For example, let us define a process P as the parallel composition of three π_{klt} processes:

```
P = Q || R || S
```

where

```
Q = x!5
R = y!6
S = when{x?data → z!data | y?data → z!data}
```

In open system analysis, the set of first actions of P is {x!5, y!6, τ_1 , τ_2, x?data, y?data} where τ_1 represents the interaction between Q and S and τ_2 represents the interaction between R and S. These first actions will generate six new sets of next states. For action x!5 next set is {(R || S)}, for action y!6 the next set is {(Q || S)}, for action x?data next set is {(Q || R || z!data)}, for action y?data next set is {(Q || R || z!data)}, for action τ_1 next set is {(R || z!data)}, and for action τ_2 next set is {(Q || z!data)}. If the analysis is a closed system analysis for the same example, the first actions set will be {τ_1 (interaction between Q and S), τ_2 (interaction between R and S)}. This set will produce two sets of next states.

The user can choose to perform any of the open or closed system analyses by setting a parameter in the initialization script.

3.6 Kinds of analyses

We can perform the analysis online or offline. Most of the analyses are performed during the generation of the state space. Others are performed after the generation of the state space tree.

After the generation of the state space, the analyzer can be used to compute and output all the paths from the root (i.e., the initial state) to states satisfying certain properties. The searching is optional. For example, if the user does not want to search stable states, he can turn off the searching by setting the parameter to False in the initialization script.

In this section we will briefly describe the analysis techniques.

3.6.1 Stable states

This analysis is performed online. If a state does not have any τ (internal) actions, that state is a stable state. Consider the process P for an example:

```
P = x!5 || y!"abc" || nil
```

The set of first actions of `P` is {`x!5`, `y!"abc"`}. As `P` does not have any τ action, `P` is a stable state. Thus a process is in a stable state if its only possible behaviors depend on the interaction with it's environment and it does not have any internal behavior. We add this state to the stable state set.

3.6.2 Deadlock states

This analysis is performed online. A state is a deadlock state if it does not have any first action. In the following example, both process `P` and process `Q` do not have any first action. Thus they are deadlock states.

```
P = nil
Q = νx.nil
```

3.6.3 Reachability analysis

This analysis is performed offline. The analyzer can also look for all the paths from the root to a specific π_{klt} state if there are any. The state to be searched for is given by the user as an external input to the analyzer. Each of the states in the state space tree is compared against the input specific state, and if they are alpha-equivalent, the path from the root to that state in the tree is returned. Our analyzer will report all the possible paths from the root to the alpha-equivalent states of the specific state for bounded analysis.

3.6.4 Searching for triggers and receivers

Sometimes it is useful to find out if a specific event is triggered or received by the system. Our π_{klt} analyzer enables this search by looking at the states as soon as they are generated (online analysis). Our analyzer compares the first actions of that state with the specific triggers we are searching for. The name of the specific trigger is taken as an input from the user in the initialization script. In a similar way we search for the name of the receiver channel that the user wants to search. There is

a latent problem in this type of analysis, and its solution is not straightforward. Due to renaming of the processes, the name of the actual trigger or receiver may be changed during the execution. Thus we used a data dictionary that holds a record of the renaming history. While searching for a specific trigger or receiver, our analyzer goes through the chain of names in the dictionary to find the actual trigger or receiver.

3.6.5 Search all events

Searching all events is similar to the searching for triggers and receivers. This analysis is performed online. Instead of searching for one specific event, we can search for a set of events. The events we want to search are given as input in the initialization script as a set by the user. We look for all the events in a state and include the path from the root to the state in the output if all the events are possible first actions of the process in that state (events can be triggering or receiving). One of the case studies in Chapter 5 shows that this search can be used to determine if there is more than one process in a critical section or not.

3.6.6 Search any events

Searching any events is similar to the searching for all events (online analysis). In this case we search for any event in the input set (provided by the user) and show the path from the root to a state where in which at least one event occurs. The search thus tells us if at least one of the given events is used in a state. In the token ring example in Chapter 5 for instance, we can use this analysis to determine that at least one event is in the critical section all the time.

3.6.7 Longest and shortest execution time

While our analyzer generates a new state, it keeps a record of the *model time* at that point. *Model time* is the sum of the time along a path in the model taken by the processes to evolve into other processes. It gives the user an idea how much time has passed in the model to reach a state. After the traversal, the analyzer uses this information to determine the state with the longest or shortest execution time (model time). This analysis is performed offline.

3.6.8 Equivalence frequency

While attempting to find a newly generated state in the seen set during the generation of state space, our analyzer also counts the number of times a state has been compared against and found equivalent. During the report generation, the frequency of successful matches for a state is printed. This gives the user an idea which states occur most frequently during execution. A state may be compared many times but that does not necessarily mean that it is actually visited all those times.

3.6.9 Specific kiltera process invocation

This analysis is performed online. Similar to the way we find a specific state, we can also search for specific process invocations, this is, given a process name A, return whether A is invoked or not. This search is especially useful to filter the reports and present the outputs to the user in a more readable way. This filtering is very helpful while analyzing the translated kiltera model of a UML-RT state machine. Using this filtering, we can show the user the path that corresponds to a specific execution of the UML-RT state machine that the kiltera process model represents.

3.6.10 Searching for dropped signals

This is another useful online analysis for the kiltera models that represent a UML-RT state machine. If a signal is triggered from the environment and the currently active UML-RT state rejects the signal, that signal is reported as dropped. Our analyzer reports the total number of dropped signals and shows the paths from the root to the π_{klt} states where the signal has been dropped. From the output our analyzer can filter the corresponding UML-RT state names and show the corresponding UML-RT state machine execution paths.

In general, given a kiltera model, if a signal is dropped anywhere in the model, our π_{klt} analyzer would show the path from the root to that state where the signal has been dropped. To map with the corresponding UML-RT state machine that the kiltera model represents, we can filter the output to show the path that corresponds to the path in a UML-RT state machine.

Chapter 4

Implementation of the π_{klt} analyzer

This chapter describes the abstract syntax used to represent π_{klt} models and visitors of the π_{klt} analyzer. The next section explores the implementation techniques which are used in the thesis. We implemented our analyzer using Python. We provide the class diagrams and small code snippets in Python for illustration. To keep the description simple we avoid a detailed description of each of the attributes and methods of the visitors and only provide the necessary details about them. In the last section, we described the techniques that we used in implementation to improve the performance of our π_{klt} analyzer.

4.1 Abstract syntax

The π_{klt} abstract syntax is composed of four different kinds of classes. They are: `Expr`, `Patt`, `ProcTerm` and `Def`. `ProcTerm` is the generalization of other π_{klt} constructs. Similarly, `Expr` and `Patt` are the generalization of the expressions and patterns to be matched. Some of the classes in `ProcTerm` contain other classes in `Expr` and `Patt`. `Def` is the generalization for process definition and function definition. `Def` contains `ProcTerm` and one specialization of `ProcTerm`(`PDef`) can have many process definitions in it. The abstract syntax is implemented following the composite pattern. The composite patterns are described in Chapter 2.

4.1.1 Expressions

A π_{klt} expression can be a constant, a name, an arithmetic expression, a Boolean expression or a conditional expression. The expression can even be a function application or a tuple of expressions. A constant can be a null constant, a number, a Boolean constant or a string constant. Arithmetic and Boolean expressions can be binary or unary expressions. The `Expr` class is the abstract class for all the π_{klt} expressions. All forms of expressions are classes themselves and `Expr` is the superclass of the other classes. Figure 4.1 shows the class diagram for π_{klt} expressions.

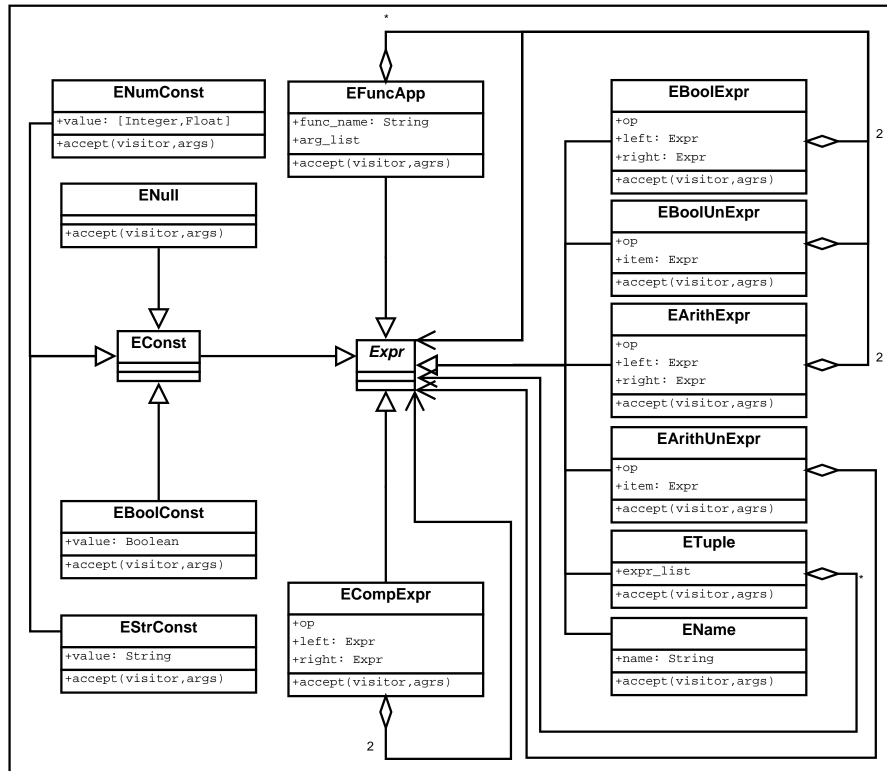


Figure 4.1: π_{klt} expressions

4.1.2 Patterns

π_{klt} patterns are similar to expressions. A pattern of data in π_{klt} is like an expression to be matched against a value. The `Patt` class is the superclass of π_{klt} patterns. `FName` (variable name), `FTuple` (tuples of patterns), `FConst` (Null constant, Boolean constant, Number constant and String constant are the specialization of pattern constants) are the subclasses of `Patt`. Figure 4.2 displays the class diagram for π_{klt} `Patt`. The main difference with expressions is that patterns do not allow function applications.

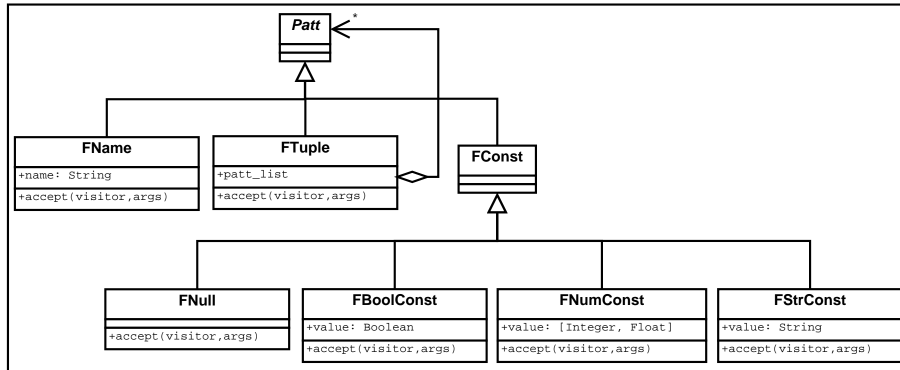


Figure 4.2: π_{klt} patterns

4.1.3 Process terms

`ProcTerm` is the abstract class to express π_{klt} processes. Each of the classes specialized from the superclass `ProcTerm` is a π_{klt} construct. Table 4.1 shows the name of the classes and the corresponding π_{klt} constructs they represent.

Figure 4.3 shows the `ProcTerm` class diagram. Some of the classes in the diagram are associated to other classes. This class relationship represents how terms can be contained in other terms, according to the composite pattern. We will briefly describe the relationship among the classes and provide some examples.

The class `PDef` in Table 2.1 is a collection of definitions (process definitions or function definitions). Each D_i is a `ProcDef` or a `FuncDef`.

Name of the Class	π_{klt} Constructs
PDone	nil
PTrig	x!E
PListener	when{x1?F1@y1 -> P1 |x2?F2@y2 -> P2 ⋮ |xn?Fn@yn -> Pn }
PNew	new x in P
PDelay	wait E -> P
PPar	P1 || P2 || . . . || Pn
PSeq	P1 ; P2 ; . . . ; Pn
PDef	def{D1, D2, . . . Dn}in P
PAssert	assert E -> P
PMatch	match E with { F1 -> P1 | F2 -> P2 . . . | Fn -> Pn}
PCond	if E then P1 else P2

Table 4.1: Classes and π_{klt} constructs

We can also see the direction of the relationship between the classes and the multiplicity in Figure 4.3. If there is no specific number for multiplicity is mentioned, it is assumed that the multiplicity is one.

To understand the implementation of the abstract syntax, let us look at the process P:

P = x!5

The abstract syntax of P is

P = PTrig('x',ENumConst(5))

P is a process that sends a message (numeric value of 5) over channel x. The implementation of the class PTrig in Python is stated below:

```
1 class PTrig(ProcTerm): #Abstract class is ProcTerm
2   #built-in function init():Constructor
3   def __init__(self, name, expr):
4     assert type(name) is str and isinstance(expr, Expr)
5     self.name = name # name = 'x'
6     self.expr = expr # expr = ENumConst(5) is an expression
7   #built-in function dir():To find out which names a module defines
```

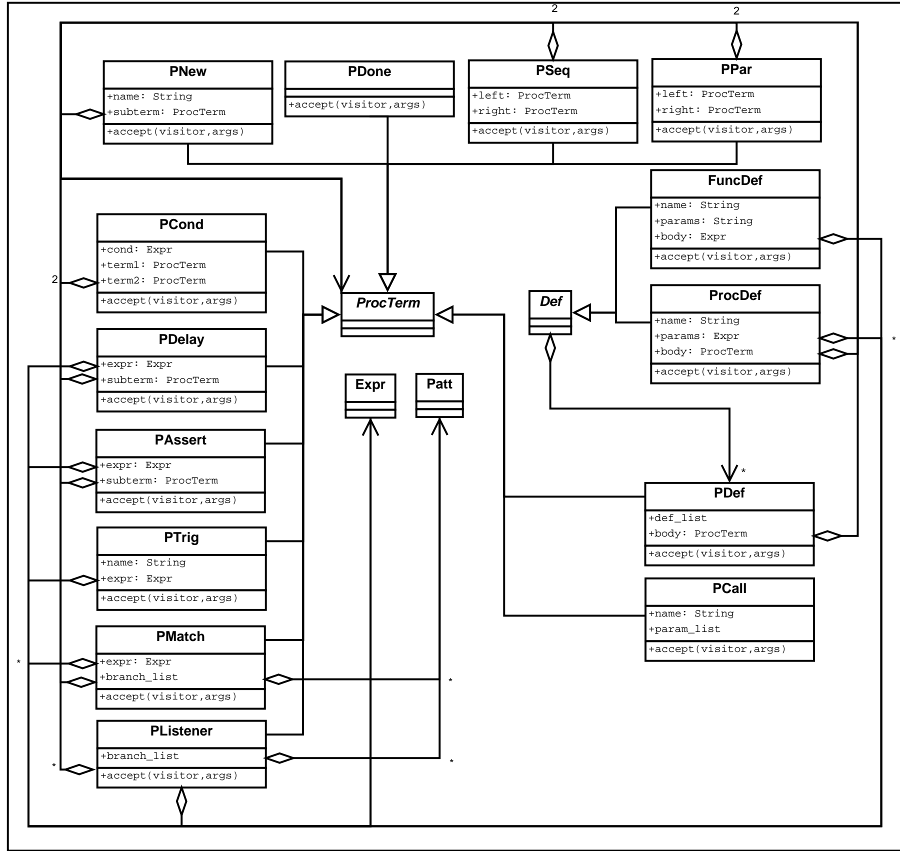


Figure 4.3: Process terms class diagram

```
8   def __dir__(self):
9     return ['name','expr']
10  #built-in function eq():Checking equality
11  def __eq__(self,other):
12    return self.__class__==other.__class__\
13    and self.name==other.name and self.expr==other.expr
14  #built-in function hash():To find out the hash value of the object
15  def __hash__(self):
16    return hash((hash(self.name),hash(self.expr)))
17  #user defined function accept():To implement the visitor
18  def accept(self, visitor, *args):
19    return visitor.visit_trigger(self, *args)
```

An instance of class `PTrig` represents a trigger construct in π_{klt}, where a trigger has a channel name

(string) and has an expression (an instance of class `Expr`). The implementation contains some built in functions and a user defined function. The `__init()__` function is the constructor in Python. We have used `__dir()__` function to find out which names a module defines. The `__eq()__` function checks the equality of two objects. `__hash()__` function is used to find the hash value of the object. To increase the readability and compactness, we will omit the description of the built in functions in the next example. The `accept()` function receives a `visitor` object as a parameter, and returns to the implementation of the function of that `visitor`. For instance, the `visitor` object implements prettyprinting. The `accept()` function then returns the implementation of the `visit_trigger()` method of the `visitor` that implements the prettyprinting.

Let us define a process `Q` which is a π_{klt} listener.

```
Q = when{x?3->z!4 | x?(2,data)-> z!5}
```

The abstract syntax of `Q` is

```
Q = PListener([
  ('x',FNumConst(3),'dummy',PTrig('z',ENumConst(4))),
  ('x',FTuple(FNumConst(5),FName('data')),'dummy',
    PTrig('z',ENumConst(5)))
])
```

A listener receives data over a channel, and while doing so, it matches the pattern of the data. If the matching is successful and the guard is true, the listener evolves to the next process in one of its branches. In our example, process `Q` receives data over channel `x`. There are two branches in the listener. The first branch checks the channel `x` and matches the received data (3). The second branch checks the same channel, but matches with a different pattern (a tuple of a number and a string).

Each branch in a listener (`x?F@y.P`) has four different variables:

`x` = name of the channel (`x` in the example)

`F` = Pattern (`FNumConst(3)` in the first branch,

`FTuple(FNumConst(5),FName('data'))` in the second branch)

`y` = Elapsed time (`dummy`)

`P` = Process (`PTrig('z',ENumConst(4))` in the first branch, `PTrig('z',ENumConst(5))` in the second branch)

The Python implementation of the listener is as follows:

```
1 class PListener(ProcTerm):
2   def __init__(self, branch_list):
3     assert type(branch_list) in [list, tuple] \
4       and all(type(branch) is tuple and len(branch) is 4 and \
5         type(branch[0]) is str and isinstance(branch[1], Patt) and \
6         type(branch[2]) is str and isinstance(branch[3], ProcTerm) \
7         for branch in branch_list)
8     self.branch_list = branch_list
9   def accept(self, visitor, *args):
10    return visitor.visit_listener(self, *args)
```

Other classes for different π_{klt} constructs are implemented in a similar way.

4.1.4 Process definition and function definition

Class `Def` is the abstract class for the definitions in π_{klt}. There are two types of definitions: Process Definitions and Function Definitions. Figure 4.3 shows the class diagram for π_{klt} definitions. A process definition is of the form:

```
proc A[x1,x2, ... , xn] = P
```

where `proc` is the keyword for process, `A` is the name of the process definition, `x1,x2,...,xn` are expressions and `P` is the body of the process. The function definition has the form:

```
func A[x1,x2, ... , xn] = E
```

where `func` is the keyword for function, `A` is the name of the function definition, `x1,x2,...,xn` are strings and `E` is the expression of the function. A π_{klt} process `def{d1,d2,..,dn} in P` can

have many process definitions (`di` are process definitions). The class that implements π_{klt} process definition is `PDef`. `PDef` can have as many process definitions (instances of `ProcDef`) as required as parameters.

4.1.5 Values

π_{klt} values are concrete values which are implemented by the class `Val`. A value can be any variable name or constant (null constant, Boolean constant, number constant or string constant) or a tuple of values. Figure 4.4 shows the class diagram for π_{klt} values. The structure of values is the same as patterns but without variables, or the same as expressions but without variables or function applications.

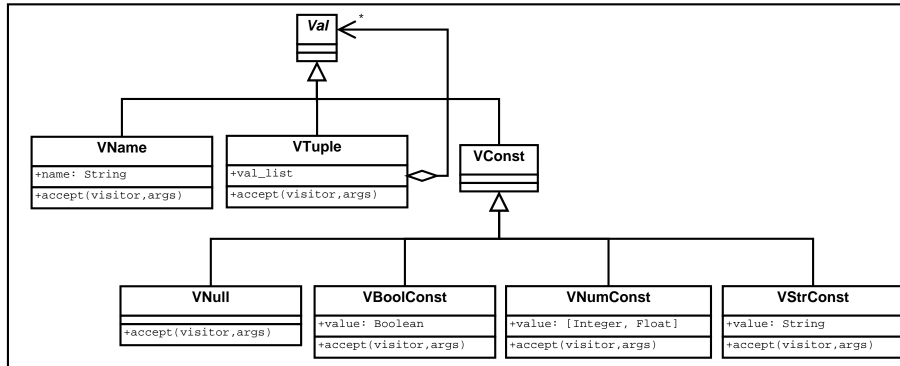
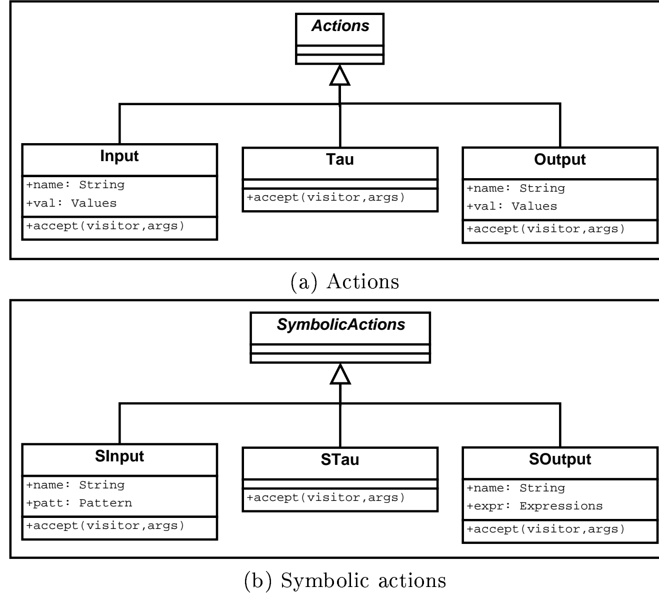


Figure 4.4: π_{klt} values

4.1.6 Actions and symbolic actions

Any π_{klt} process can perform three basic types of actions: input actions, output actions and internal or silent actions (represented as τ actions). There are two different kinds of actions: `Actions` (concrete) and `Symbolic Actions`. `Actions` are concrete actions whose arguments are fully evaluated and represented as values. `Symbolic Actions` are actions whose arguments are not evaluated and expressed using expressions and patterns. For instance, consider a process `P` = `x!(2+3)`. The first action set of `P` is {`x!(2+3)`} which is symbolic (the expression is not evaluated). To generate the next set of states of `P`, we need to evaluate the expression of the first action of `P`. This evaluation

would result in {`x!5`} which is concrete. With this concrete action and the process `P` we can now generate the set of next terms. Figure 4.5a and Figure 4.5b show the class diagrams for Action and Symbolic Actions.



(a) Actions

(b) Symbolic actions

Figure 4.5: π_{klt} Actions and symbolic actions

4.2 Visitors

Visitors are used to implement different functionalities of π_{klt}. We will discuss the implementation of different visitors in this section. We will discuss the techniques and present small code snippets with examples. However we will not go through the details of the implementation of all the functions in visitors, rather we will only describe the most intricate functions.

4.2.1 Expression visitor

Expression visitors are used to implement operations on π_{klt} expressions. Figure 4.6 shows the class diagram for the π_{klt} expression visitor.

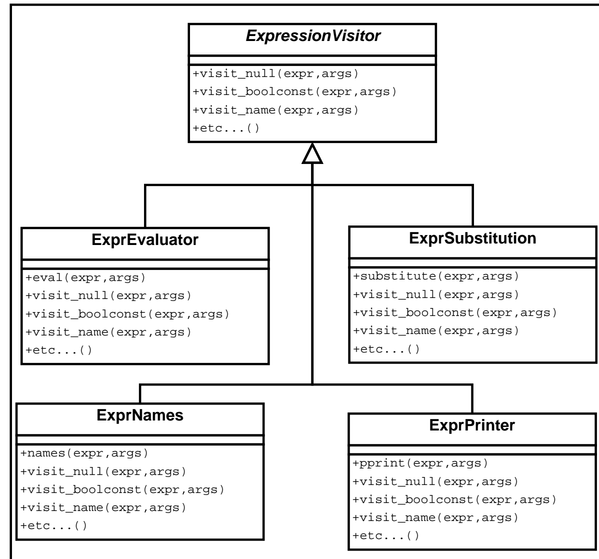


Figure 4.6: π_{klt} expression visitor

The `ExpressionVisitor` is the superclass for the expression visitors. The class `ExprNames` calculates the names of an expression. For example, let `e` be a π_{klt} expression:

`e` = `ETuple(EName('x'),EName('y'),EName('z'))`

where the `ExpressionVisitor` will return the set `{x, y, z}`, if we want to find the set of names of variables of the expression `e`. The Python snippet to invoke the `ExprNames` for the example would be:

```
expr_names = ExprNames() #create an instance of the visitor
set_of_names = expr_names.names(e)
```

The `visit_tuple` function of `ExprNames` is implemented as:

```
1 class ExprNames(ExpressionVisitor):
2   def names(self, expr):
3     return expr.accept(self)
4   def visit_names(self, expr):
5     return set([expr.name])
6   ⋮
7   def visit_tuple(self, expr):
```

```
8     result = set([]) // empty set
9     for subexpr in expr.expr_list:
10      names_of_subexpr = subexpr.accept(self)   # this returns a set
11      result = result.union(names_of_subexpr)    # we compute the union
12    return result
```

The visitor `ExprEvaluator` evaluates a π_{klt} expression and returns the result. Let us consider the π_{klt} expression `e`:

`e` = `EArithExpr(plus,ENumConst(5),ENumConst(3))`

the visitor would return the concrete value of `VNumConst(8)`. The `ExprSubstitution` visitor updates an expression by applying a substitution to it. If `e` = `EName('x')` is an expression and the substitution is $\sigma = \{y/x\}$, the visitor `ExprSubstitution` would return `EName('y')`. The visitor `ExprPrinter` is used to prettyprint the expression.

4.2.2 Pattern visitor

The `PatternVisitor` visitor is used to implement operations on a π_{klt} patterns. Figure 4.7 shows the class diagram for the π_{klt} pattern visitor.

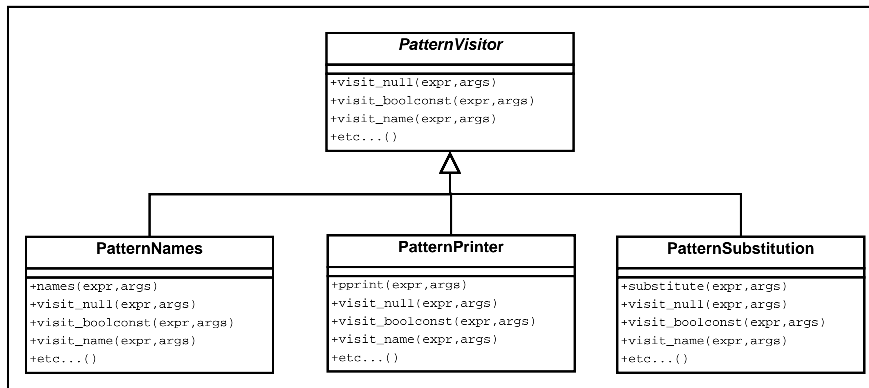


Figure 4.7: π_{klt} pattern visitor

The visitor `PatternNames` is used to figure out the names of variables in a π_{klt} pattern. If `f` = `FTuple(FName('x'),FName('y'))` is a pattern, the visitor `PatternNames` would return the set

of names `{x,y}`. The `PatternPrinter` is another visitor used to prettyprint the patterns. The `PatternSubstitution` visitor performs the substitution on a π_{klt} pattern.

4.2.3 ProcTermVisitor and ProcDefVisitor

As `ProcTermVisitor` implements operations on π_{klt} processes, we will describe all the visitors specialized from the `ProcTermVisitor` visitor in brief with suitable examples and Python code snippets. Figure 4.8 shows the class diagram of the `ProcTermVisitor` visitor. Almost all the visitors specialized from `ProcTermVisitor` have a relationship with the `ProcDefVisitor`. The `ProcDefVisitor` implements the operations for process definitions (`PDef` which is a specialization of `ProcTerm`). Process definitions must have a process (`ProcTerm`) in its body. This is why most of the visitors in `ProcTermVisitor` are related to the visitors of `ProcDefVisitor`. For example, let us consider a process `P`:

```
def{proc A(x)= x!5, proc B(y)= y!2} in A(2.00)
```

The abstract syntax for `P` is:

```
P = PDef(
  [
    ProcDef('A', [EName('x')], PTrig('x', ENumConst(5))),
    ProcDef('B', [EName('y')], PTrig('y', ENumConst(2)))
  ], PCall('A', [ENumConst(2.0)]))
```

The class `PDef` contains the instances of the class `ProcDef` as parameters and the instances of the class `ProcDef` contains the instances of the class `ProcTerm`. If we want to prettyprint `P`, we have to visit both the `ProcTermPrinter` and `ProcDefPrinter`. For this reason, we will describe the `ProcTermVisitor` and `ProcDefVisitor` together here. Figure 4.9 shows the class diagram of `ProcDefVisitor`.

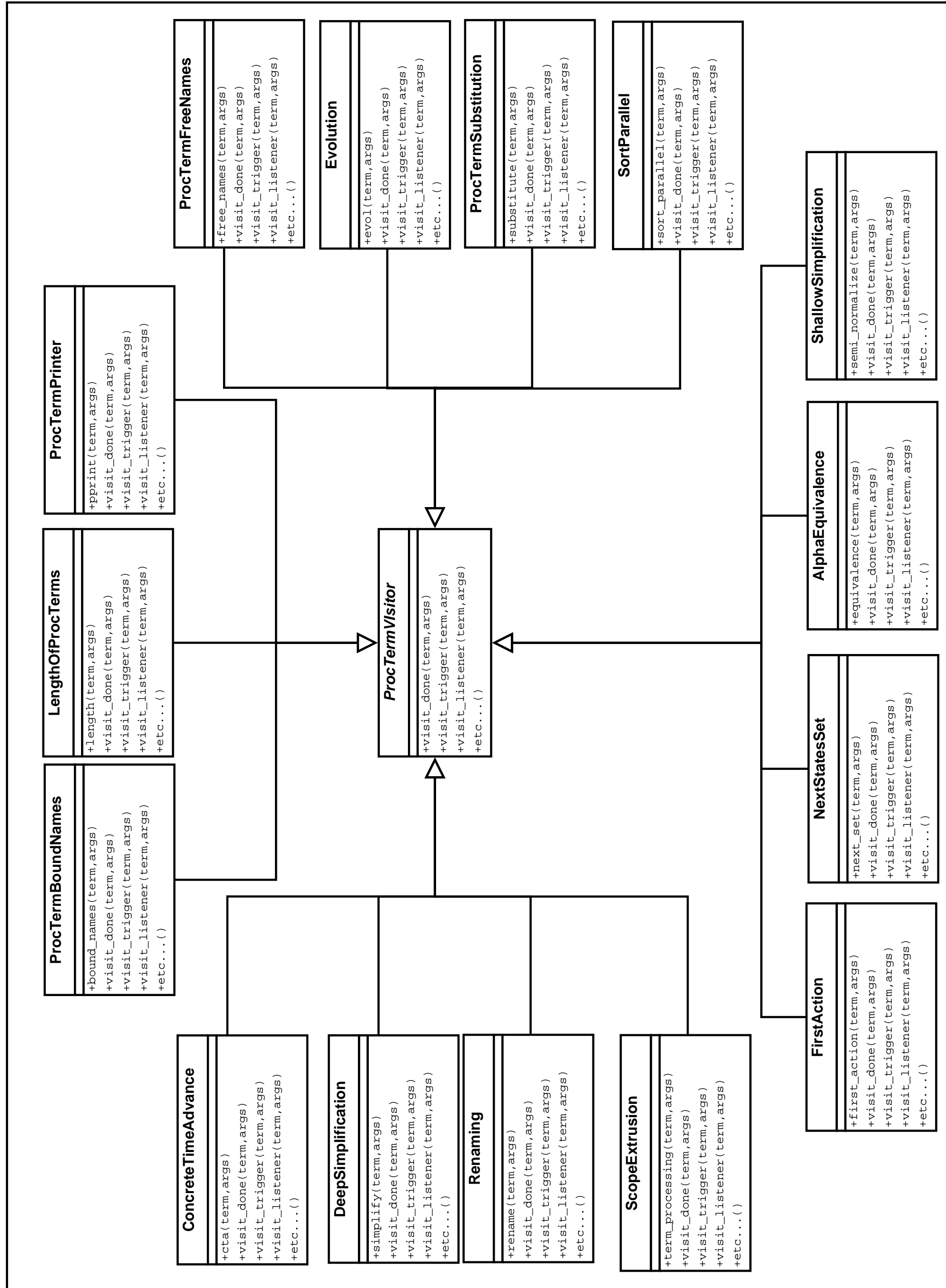


Figure 4.8: π_{klt} process terms visitor

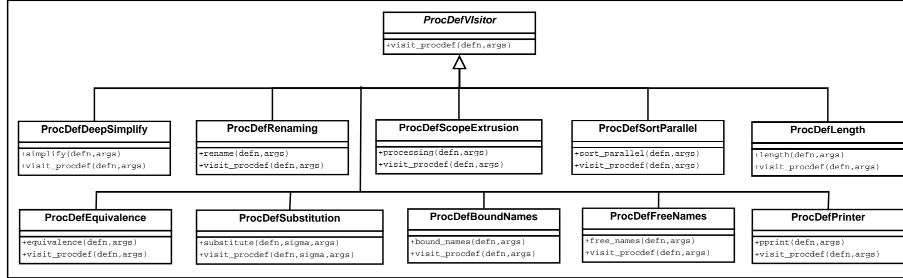


Figure 4.9: π_{klt} process definition visitor

4.2.3.1 Free names and bound names

The `ProcTermFreeNames` and the `ProcDefFreeNames` are the visitors to calculate the free names in a process. If `P = new x in (x!y||z!y)` is a process, the set of free names is `{y,z}`. `x` is declared as a channel name and is hidden from the environment in the sub term `(x!y||z!y)`. `ProcTermBoundNames` and `ProcDefBoundNames` are the visitors to find the bound names in a process. The set of bound names for the process `P` is `{x}`. `ProcTermPrinter` and `ProcDefPrinter` visitors are used to prettyprint the process.

4.2.3.2 Substitution

`ProcTermSubstitution` and `ProcDefSubstitution` are the visitors which perform the substitution of names in a process. To understand the substitution, let us consider a process `Q = new y in y!x` and the substitution function is $\sigma = \{^{y}/_{x}\}$. If we apply σ to `Q` naively, we would get the resulting process `Q = new y in y!y`. The problem is that, before applying σ to `Q`, the variable `x` was free in `Q`. But, after the substitution, the variable is captured the variable `y`. To avoid this capturing of free variables, we rename the bound names of `Q` with a fresh name. Now, if we apply the substitution to `Q`, the variable `y` avoids the capturing. If `y′` is a fresh name, the correct substitution would be: `Q = new y′ in (y!x)`$\{^{y'}/_{y}\}\{^{y}/_{x}\}$. We get the resulting process `Q = new y′ in y′!x`. The Python code snippet for `ProctermSubstitution` visitor illustrates how it is implemented.

```
1 def fresh(): # function to create fresh names
2   global counter
3   counter = counter+1
```

```
4    return str('_X_'+str(counter))
5
6  class ProcTermSubstitution(ProcTermVisitor):
7    def substitute(self, procterm, sigma):
8      assert isinstance(procterm, ProcTerm)
9      return procterm.accept(self, sigma)
10   ⋮
11  def visit_new(self, procterm, sigma):
12          fresh_name = fresh()
13          sigma[procterm.name] = EName(fresh_name) #create substitution
14          new_term = PNew(fresh_name, procterm.subterm.accept(self, sigma))
15          del sigma[procterm.name] # deleting used recent substitution
16          return new_term
```

4.2.3.3 Concrete time advance and evolution

The `ConcreteTimeAdvance` visitor finds the amount of time that a process may delay during the execution of that process over time. Let `P = wait E`$\rightarrow$`P`$'$ be a π_{klt} process. Process `P` must wait `E` amount of time before the execution of process `P`$'$. The `ConcreteTimeAdvance` visitor calculates the delay time by evaluating `E` using the `ExprEvaluator` visitor and returns its value. An `Evolution` visitor outputs the process term that has been changed with the time. The `Evolution` visitor advances time by a given parameter `d` and updates a given process appropriately. For instance, the process `P` will evolve to process `Q = wait(E-d)`$\rightarrow$`P`$'$ after `d` amount of time.

4.2.3.4 Deep and shallow simplification

The `DeepSimplification` and the `ProcDefDeepSimplify` visitors remove the redundant name declarations and the redundant `nil` term from a process such that the behavior of that process is unchanged. They remove the `nil` process recursively from a process. The `ShallowSimplification` visitor also removes the `nil` term and name declarations from a process, but not recursively. Both the visitors follow the same rules for simplification as defined in the Subsection 3.2.2 and Subsection 3.3.1 of Chapter 3. The reasons for using two different kinds of simplification is to reduce the

time complexity. `DeepSimplification` and `ProcDefSimplify` are used only once during the pre-processing step. The visitor `ShallowSimplification` is used in every step of the analysis. Let us consider a process `P = when{x?y→P′||nil}`. `ShallowSimplification` visitor will output `P` without removing the `nil` term, but `DeepSimplification` visitor will return the process `Q = when{x?y → P′}`. We omitted the recursive removal of `nil` term in the `ShallowSimplification` visitor because we will remove the `nil` process from the process `(P′||nil)` when we receive data on the channel `x` of the process `P`. The other reason for employing two different kinds of simplification is that we cannot determine from earlier how a process would evolve. Let us consider another process `R = x !5 || y!6`. The first action set of `R` is `{x!5, y!6}`. So, the first action `x!5` along with process `R` will generate a new process `R′ = nil || y!6`. We could never remove the `nil` term from `R′` in the pre-processing step. But, this `nil` term will be removed in the shallow simplification step by the visitor `ShallowSimplification`.

4.2.3.5 Renaming

The `Renaming` and `ProcDefRenaming` visitor rename the bound names of a process with fresh names in the pre-processing step. The steps for renaming are:

1. create a fresh name for each of the bound names
2. create a substitution
3. apply the substitution to the subterm of that process.

This renaming follows the definitions stated in the Subsection 3.2.3 of the Chapter 3.

4.2.3.6 Scope extrusion

The `ScopeExtrusion` and `ProcDefScopeExtrusion` visitors are used to extract the `new` operators from a process and put them at the front of the process. Let us define a process `P = (new x in Q) || R`. Applying scope extrusion on `P` will result in a process `P′ = new _X_1 in (Q{_X_1/x} || R)`, where `_X_1` is the fresh name for x and in `Q{_X_1/x}` each free occurrence of `x` is replaced by `_X_1`.

The following Python code snippet will illustrate the implementation of the `ScopeExtrusion` visitor. We show the implementation of the `visit_parallel` function only.

```
1 class ScopeExtrusion(ProcTermVisitor):
2  def process_term(self, term, *args):
3     assert isinstance(term, ProcTerm)
4     return term.accept(self, *args)
5   ⋮
6  def visit_parallel(self, term, *args):
7    l = term.left.accept(self,*args)
8    r = term.right.accept(self,*args)
9    if not isinstance(l,PNew) and not isinstance(r,PNew):
10     return PPar(l,r)
11   elif isinstance(l,PNew) and not isinstance(r,PNew):
12     return PNew(l.name,PPar(l.subterm,r).accept(self,*args))
13   elif not isinstance(l,PNew) and isinstance(r,PNew):
14     return PNew(r.name,PPar(l,r.subterm).accept(self,*args))
15   else:
16     return PNew(l.name,PNew(r.name,\
17     PPar(l.subterm,r.subterm).accept(self,*args)))
```

A parallel composition of processes has at least two subprocesses: `left` and `right`. In the implementation we check for all the combinations and extract the `new` operators from the process.

4.2.3.7 Length of a π_{klt} term

The `LengthOfProcTerms` and `ProcDefLength` visitors are used to calculate the number of nodes in the abstract syntax tree of the process. For example, let us consider a process `P`:

`P = new x in x!y || z!5`

The abstract syntax is:

`P = PPar(PNew('x',PTrig('x',EName('y'))),PTrig('z',ENumConst(5)))`

The total number of nodes in `P` is 4. So, the length of `P` is 4.

4.2.3.8 Sorting parallel terms

`SortParallel` and `ProcDefSortParallel` are the visitors used to sort the π_{klt} parallel terms. Sorting means to sort the parallel terms based on their length. This sorting is useful when we compare two similar terms for alpha-equivalence. For example, let `P = Q ||( R || S)` be a process where the lengths of `Q`, `R` and `S` are assumed to be 20, 15 and 25 respectively. After the term `P` is sorted, the resulting output would be `P = R || (Q || S)`. Let us consider another process $\texttt{P}' = \texttt{R}'$ `||` $(\texttt{Q}'$ `||` $\texttt{S}')$, where the lengths of $\texttt{R}'$, $\texttt{Q}'$ and $\texttt{S}'$ are 14, 20 and 25. Process `P` and $\texttt{P}'$ cannot be alpha-equivalent as the lengths of `R` and $\texttt{R}'$ are not same. We can determine that the terms are not alpha-equivalent by just observing that the corresponding subterms have different lengths.

4.2.3.9 Alpha-equivalence checking

The `AlphaEquivalence` and `ProcDefEquivalence` visitors are used to compare two processes and determine if they are alpha-equivalent. The following Python code snippet shows the implementation of this test for π_{klt} `new` terms.

```
1  class AlphaEquivalence(ProcTermVisitor):
2    def equivalence(self, term, other, *args):
3      assert isinstance(term, ProcTerm) and isinstance(other, ProcTerm)
4      return term.accept(self, other, *args)
5    ⋮
6    def visit_new(self, term, other, *args):
7      if term.__class__ != other.__class__: #term and other are of different class
8        return False
9      elif term == other: #term and other are equal
10       return True
11     elif term.name == other.name \ # names are same; check for subterm
12       and self.equivalence(term.subterm, other.subterm, *args):
13       return True
14     elif term.name != other.name: #names are not same
15       sigma[other.name] = EName(term.name) #create substitution
16       #~Apply the substitution on the subterm of other process ~#
17       renamed_proc = proc_substitution_visitor.substitute(other.subterm, sigma)
18       #~check for equivalence ~#
```

```
19        if self.equivalence(term.subterm, renamed_proc, *args):
20          return True
21        else:
22          return False
23      else:
24        return False
```

The implementation is straightforward. If we create the the substitution for the bound names of one process, and apply the substitution to the other process then the resulting subterms are alpha-equivalent. If the subterms are alpha-equivalent, the two processes are also alpha-equivalent.

The implementation of the comparison is recursive in the structure of the processes. If the two top nodes on the AST processes have different types, the processes are not alpha-equivalent. Otherwise, the subterms are compared recursively for alpha-equivalence. The implementation leverages the fact that the processes in the parallel composition have been sorted by default, the listeners in a `when` clause are compared in the order they are listed. However, an implementation that considers every permutation is also available.

4.2.3.10 First actions

A first action of a π_{klt} process is an action that the process can perform immediately. The `FirstAction` visitor takes a π_{klt} process as an input and returns the set of first actions of that input. The implementation of the `FirstAction` is straightforward according to the definitions [54]. The most interesting part of the implementation is the first action of a parallel process.

In a parallel process, there might be processes that can interact with each other. If two processes can interact, we will get an action called a τ action or silent action or internal action. To find out the τ actions of interacting processes we define a function `compairs` which takes two processes as input, and returns the set of pairs of actions of interacting processes. The actions in a pair will produce a τ action only if the actions are complementary. To determine if two actions are complementary, we define another function called `complement`. Two actions are said to be complementary if one of the action is an input action, the other action is an output action, their channel names are the same, and the `Expression` of the output action matches the `Pattern` of the input action. For example,

let α_1= `x!E` and α_2= `x?F` then α_1 is an output action, α_2 is an input action, `x` is the name of the channel, `E` is an expression and `F` is a pattern. The actions α_1 and α_2 are complementary if `match(expr_eval(E),F,`$\emptyset$`)` $\neq \bot$, where `match` is the function that takes a pattern and a value as input, and returns the substitution if there is any match or a mismatch otherwise [54]. `expr_eval(E)` evaluates the expression, and returns the value. $\bot$ is the symbol used for mismatch.

The Python snippet for the implementation of `FirstAction` visitor is shown below. We present the implementation for the case of parallel process only.

```
1 class FirstAction(ProcTermVisitor):
2   def f_action(self, term):
3     assert isinstance(term, ProcTerm)
4     return term.accept(self)
5   ⋮
6   def visit_parallel(self, term, *args):
7     if compairs(term.left,term.right) != set([]): # if left and right may interact
8       return term.left.accept(self,*args)| \ # set Union
9              term.right.accept(self,*args)| \
10             set([STau(term)])
11    else: # processes are not interacting
12      return term.left.accept(self,*args)\
13             |term.right.accept(self,*args)
14
15 def compairs(left, right): # left and right are processes
16   action_set1=first_action_visitor.f_action(left)
17   action_set2=first_action_visitor.f_action(right)
18   return set([(x1,x2) for x1 in action_set1 \ # returns the set of pairs of actions
19                       for x2 in action_set2 \ # (x1,x2) from the first action sets
20                       if complement(x1,x2)])  # of processes "left" and "right" which
21                                               # are complementary of each other.
22 #complement function implementation
23 def complement(action1, action2):
24   if isinstance(action1,SInput) and isinstance(action2,SOutput):
25     temp = action1
26     action1 = action2
27     action2 = temp
28     s = match(action2.patt, expr_eval_visitor.eval(action1.expr),sigma)
```

```
29    if not isinstance(s,MisMatch) \
30      and (isinstance(action1,SOutput)and isinstance(action2,SInput))\
31      and action1.name == action2.name:
32      return True
33    else:
34      return False
35  elif isinstance(action1,SOutput) and isinstance(action2,SInput):
36      s = match(action2.patt, expr_eval_visitor.eval(action1.expr),sigma)
37      if not isinstance(s,MisMatch) \
38         and (isinstance(action1,SOutput)and isinstance(action2,SInput))\
39         and action1.name == action2.name:
40         return True
41      else:
42         return False
43  else:
44      return False
45
46 def match(patt, value, sigma):
47    if isinstance(patt,FTuple) and isinstance(value,VTuple):
48        sigma_new = copy.deepcopy(sigma)
49        for p,v in zip(patt.patt_list,value.val_list):
50            sigmaP = match(p,v,sigma_new)
51            if not isinstance(sigmaP,MisMatch):
52                sigma_new = sigmaP
53                returnValue = sigmaP
54            else:
55                returnValue = mis_match
56                break
57        return returnValue
58
59    elif isinstance(patt,FConst) and \
60         (isinstance(value,VConst) or \
61          isinstance(value,EConst)):
62        if patt.value == value.value:
63            return sigma
64        else:
65            return mis_match
```

```
66    elif isinstance(patt,FName) and patt.name in sigma and sigma[patt.name] == value :
67        return sigma
68    elif isinstance(patt,FName)and not(patt.name in sigma):
69        sigma_new = copy.deepcopy(sigma)
70        sigma_new[patt.name] = value
71        return sigma_new
72    else:
73        return mis_match
```

4.2.3.11 Next state set

The visitor `NextStateSet` takes a process and an action as input, and generates the set of next states. The definitions of generating this set can be found in the [54]. The implementation is straightforward according to the definitions. To illustrate the implementation of this visitor, we will provide an example for the case of parallel processes only.

The visitor `NextStateSet` takes a process, an action and an environment as input. The environment variable is discussed in the Chapter 3 with example. Given a parallel process

```
P = x!y || when{x?y1→y1!5} || when{x?y2→y2!100} .
```

the set of first actions of the process `P` is `{x!y, x?y1, x?y2,` τ`}`. If we consider the process `P` and the action τ , the set of next states is

```
{(nil || when{x?y1→y1!5} || y!100),
 (nil || y!5 || when{x?y2→y2!100})}.
```

After the shallow simplification step, the set is

```
{(when{x?y1→y1!5} || y!100), (y!5 || when{x?y2→y2!100})}.
```

Here the processes in `P` are interacting. To find the processes that can interact, we define a function called `interaction`. The function takes two processes of a parallel process and an action as input, and determines whether the processes can interact or not only if the action is a τ action. To determine the interaction, we defined another helper function called `compairs_ws` (read as "compairs with substitution"). This function takes two processes as input and returns the pair of interacting actions

if any, along with the created substitution or the mismatch. The rules for match or mismatch are the same as described for the `FirstAction` visitor except that the complement function is redefined as `complement_ws` (read as "complement with substitution") which returns a substitution in case of a match or a mismatch otherwise, instead of a Boolean value. The following Python snippet illustrates the implementation of the visitor in case of parallel processes only.

```
1 class NextStatesSet(ProcTermVisitor):
2   def next_set(self, term, env, action, *args):
3     isinstance(term,ProcTerm)
4     return term.accept(self, env, action, *args)
5   ⋮
6   def visit_parallel(self, term, env, action, *args):
7     s1 = interaction(term.left, term.right, env, action, *args)
8     p1 = term.left.accept(self, env, action, *args)
9     q1 = term.right.accept(self, env, action, *args)
10    temp = set([PPar(k.term, term.right) for k in p1])\
11         | set([PPar(term.left, k.term) for k in q1])
12    s = temp | set([x.term for x in s1])
13    if s != set([]):
14      final_set = set([])
15      for item in s:
16        new_state = State(item,env)
17        final_set.update(set([new_state]))
18      return final_set
19    else:
20      return set([])
21
22 #interaction function implementation
23 def interaction(left, right, env, action, *args):
24   s = set([])
25   if not(isinstance(action, STau)):
26       return set([])
27   else:
28       action_pair = compairs_ws(left,right)
29       for act in action_pair:
30           a = action_eval_visitor.eval(act[0],act[2], *args)
```

```
31          p1 = next_state_visitor.next_set(left,env,a, *args)
32          a = action_eval_visitor.eval(act[1],act[2], *args)
33          p2 = next_state_visitor.next_set(right,env,a, *args)
34          s = set([PPar(x1.term,x2.term)for x1 in p1 for x2 in p2])
35          s.update(s)
36
37      if s != set([]):
38          final_set = set([])
39          for item in s:
40              new_state = State(item,env)
41              final_set.update(set([new_state]))
42          return final_set
43      else:
44          return set([])
45
46 #compairs_ws funtion implementation
47 def compairs_ws(left, right):
48     action_set1=first_action_visitor.f_action(left)
49     action_set2=first_action_visitor.f_action(right)
50     return set([(x1,x2,complement_ws(x1,x2)) \
51        for x1 in action_set1 \
52        for x2 in action_set2 \
53          if not isinstance(complement_ws(x1,x2),MisMatch)])
54
55 #complement_ws function implementation
56 def complement_ws(action1, action2):
57     if isinstance(action1,SInput) and isinstance(action2,SOutput):
58         temp = action1
59         action1 = action2
60         action2 = temp
61     if (isinstance(action1,SOutput)and isinstance(action2,SInput))\
62         and action1.name == action2.name:
63         s = match(action2.patt, expr_eval_visitor.eval(action1.expr),sigma)
64     else:
65         s = MisMatch()
66     return s
```

The `match` function is the same as that used in the implementation of the `FirstAction` visitor. The work flow is simple. We receive an input program's AST in the analyzer. The `FirstAction` visitor computes the set of first actions. The analyzer creates a π_{klt} state with a term and its environment (as a pair). The analyzer then calls the `StateVisitorImplementation` visitor to generate the new state. The `StateVisitorImplementation` invokes `NextStatesSet` to generate the next set of states. `NextStatesSet` generates the set of next states, creates a π_{klt} state (a term and an environment pair) for each term and environment in the set, and returns the set of states to the analyzer via the `StateVisitorImplementation` visitor.

4.2.4 State visitor

A π_{klt} state contains a π_{klt} term and an environment. As soon as we get a π_{klt} term, we create a π_{klt} state for that term. For instance, when the execution of our analyzer begins, we receive the AST for the input model. We create a π_{klt} state with that AST (which is a π_{klt} term) and an empty environment. From this state our analyzer will generate the next set of states (where each state contains a π_{klt} term and an environment extension if there is any). The `StateVisitorImplementation` receives the π_{klt} state and calls the function of the `NextStateSet` based on the term in the state, and pass the environment to the function of the `NextStateSet` as a parameter. Figure 4.10 shows the class diagram for State Visitor.

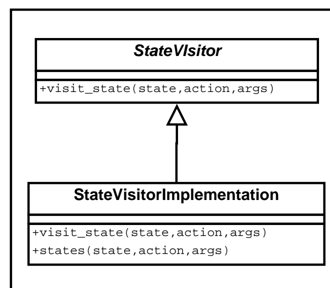


Figure 4.10: π_{klt} state visitor

4.2.5 Symbolic action visitors

The `SymbolicActionVisitor` is used to implement some analyses and some internal activities required by the analyzer. Figure 4.11 shows the class diagram for `SymbolicActionVisitor`.

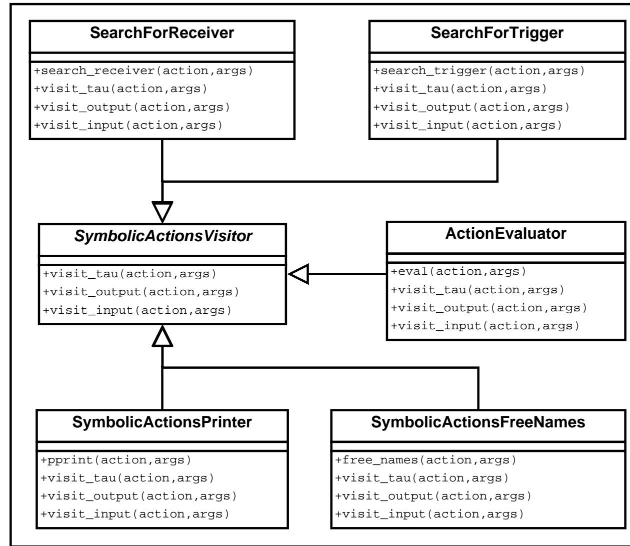


Figure 4.11: π_{klt} symbolic action visitor

The `SearchForReceiver` visitor is used to determine if a pre-specified signal or message is present in a π_{klt} state or not. For instance, let `P = `**`when`**`{x?5→nil}` be a process which has the first action `{x?5}`. Say we want to search for the channel `x` which receives a message. The `SearchForReceiver` visitor will compare the pre-specified (taken as input before the execution of the analyzer begins) channel name `x` with the channel name `x` of the current process `P` and because they are same, the state is saved in a queue. After the generation of state space, we can find the paths where the channel has received a message. The `SearchForTrigger` visitor performs a similar task as `SearchForReceiver`, except it looks for channels that are triggered.

The `ActionEvaluator` visitor evaluates the `Expression` in an output action and returns the concrete output action. An action is a concrete output action, if the expression is evaluated. If the action is an input action, the `ActionEvaluator` substitutes the pattern with a concrete value which can match. When we compute the first actions of a process, the output actions are symbolic (expressions are not evaluated for output action, patterns are not substituted for input actions). Then we generate

the set of next terms with the first actions. Before the generation of the next terms, we evaluate the actions (i.e., evaluate the expression for the output action, substitute the pattern for the input action). For instance, `a=x!(2+3)` is an action which is symbolic (i.e., the expression is not evaluated). The `ActionEvaluator` visitor evaluates the expression and returns the concrete action `a=x!5`.

The `SymbolicActionFreeNames` visitor determines the free names of a symbolic action. The visitor `SymbolicActionPrinter` prettyprints the π_{klt} first actions.

4.2.6 Action visitor

The `ActionVisitor` calculates the free names and bound names of an action. `ActionBoundNames` returns the set of bound names of an action and `ActionFreeNames` returns the set of free names of an action.

4.2.7 Values visitor

The `ValuesVisitor` is used to implement the comparison operators of π_{klt}. It takes two values and a comparison operator, and returns the Boolean value as a result.

4.3 Performance optimization of the analyzer

We can improve the performance of the analysis modes by employing some techniques. We employed some techniques to improve the performance of the implementation of the visitors. For example, we used set comprehension, instead of using list or tuple because set operations in Python are fast. Moreover, we tried to reduce the comparisons as much as possible. We will now describe some of the most effective optimization techniques.

4.3.1 Improvement of alpha-equivalence checking

As soon as a new state is generated in all the modes of analysis, we need to compare that state with the existing states in the state space. If the state is found to be alpha-equivalent to any one

of the states in the tree, that new state is discarded. As described before, two processes are alpha-equivalent if and only if there exists a substitution of the bound variables of one process that makes both of them equal. The implementation is quite straightforward but the following optimization is used to determine non-alpha-equivalence more quickly: if two processes have different lengths, they cannot be alpha-equivalent and no substitution is even attempted. The following example will illustrate this optimization. Let P and Q be two processes:

```
P = when{x?5->z!5}
Q = when{x?5->z!5 | x?6-> z!6}
```

P and Q are not equivalent because of their different branches. Q has two branches, where P has only one. We can thus get rid of alpha-equivalence checking of P and Q by comparing the numbers of the branches and determine that they are not alpha-equivalent. Algorithm 4.1 shows the algorithm for equivalence checking for π_{klt} listeners.

Algorithm 4.1 Alpha-equivalence checking of π_{klt} listeners

```
1  P:= Process 1, Q:= Process 2
2  if P and Q are equal:
3    return True # algorithm terminates
4  else if lengthOfBranch(P)==lengthOfBranch(Q):#lengths are same
5    for (branch1,branch2) where branch1 is in P ||
6    and branch2 is in Q):#explore the branches in pair of each listener
7      if branch1[0] == branch2[0] # names of channels are equal
8      and branch1[1] == branch2[1] # patterns are of same construct
9        #create substitution for pattern
10       σ = create_substitution(branch1[1],branch2[1])
11       if σ is ⊥: # mismatch
12         return False # algorithm terminates
13       else:
14         # extend σ for elapsed time variable
15         σ[branch2[2]] = EName(branch1[2])
16         new_term = apply σ on branch2[3]
17         if new_term and branch1[3] are not equivalent:
18           return False #algorithm terminates
19         end if
20       end if
21   end for
22 else:
23   return False
24 end if
```

The above algorithm implements the alpha-equivalence for listener processes. However, alpha-equivalence still is quite restrictive. Consider for example, the processes P and Q below:

```
P = when{x?6-> z!6 | x?5-> z!5}
Q = when{x?5-> z!5 | x?6-> z!6}
```

`P` and `Q` are not alpha-equivalent. However, they clearly have the same behavior, because the order in which the branches in the listeners occurs does not impact the behavior of the processes. To find this kind of behavioral equivalence, we also implemented a check that compares each branch in one listener with each branch of the other listener. However, due to its complexity, it is only used when enabled explicitly by the user.

Algorithm 4.1 represents one example of the procedures used in the implementation of alpha-equivalence checking. For different π_{klt} constructs the algorithms vary. It is not possible to define a general algorithm for all the constructs because of their different syntax.

4.3.2 Get rid of redundant renaming

Renaming plays a vital role in the implementation of π_{klt} analyzer. Renaming guarantees the avoidance of capturing free names of a process. However sometimes it is unnecessary as capture may actually not occur. Unfortunately, we cannot determine in which situations free variables are safe. But, we can improve the renaming technique to get rid of some cases of excessive renaming. This is one of the most time consuming techniques in our implementation. We provide the example snippets for both the old and new implementations of process term substitution of `new` processes. According to the definition of substitution [54], the algorithm for `new` process is shown in Algorithm 4.2.

Algorithm 4.2 Substitution of new process term

```
1 P:= process term, σ := substitution
2 σ' = new empty substitution
3 fresh_name = fresh() # create a fresh name
4 σ'[P.name] = fresh_name
5 new_proc = apply σ' on P.subterm
6 result_proc = apply σ on new_proc
7 P.name = fresh_name
8 P.subterm = result_proc
9 return P
```

Say, `P = new x in Q` is a process and $\sigma = \{y/x\}$ is a substitution to be applied on `P`. When we call the implementation of Algorithm 4.2 for applying the substitution, first we create a new substitution

$\sigma' = \{^{x'}/_{x}\}$ where $\mathtt{x}'$ is a fresh name to ensure that the declared name differs from the fresh name. We apply σ' on $\mathtt{Q}$ and get a new process term $\mathtt{P}'$ = **new** $\mathtt{x}'$ **in** $\mathtt{Q}'$, where $\mathtt{Q}' = \mathtt{Q}\{^{x'}/_{x}\}$. Now, we apply σ on $\mathtt{Q}'$ and return the resulting new process term with a fresh name in the declaration and $\sigma(\mathtt{Q}')$ as new sub-term of the process.

While this algorithm is safe enough to avoid the capturing of free names in $\mathtt{Q}$, this is not efficient. We can see that we are applying substitution twice on $\mathtt{Q}$ with σ and σ'. This is redundant and we can get rid of this redundant renaming by extending the substitution σ before applying it. For the same process $\mathtt{P}$ and substitution σ, we can extend σ by appending a new substitution σ' . So, $\sigma := \sigma \cup \sigma'$. We can now apply σ on $\mathtt{Q}$, and return a new process with the fresh name and a new subterm. Before returning the new process we can exclude σ' from σ because σ' is supposed to be a local substitution and should not have any impact on other processes. Algorithm 4.3 shows the algorithm that uses this technique.

Algorithm 4.3 Modified substitution of new process term

```
1  P:= process term, σ := substitution
2  fresh_name = fresh() # create a fresh name
3  if P.name is in σ:
4     tmp = σ
5  σ[P.name] = fresh_name #extending sigma
6  new_proc = apply σ on P.subterm
7  P.name = fresh_name
8  P.subterm = new_proc
9  del σ[P.name]
10 σ = tmp
11 return P
```

4.3.3 Implementing the seen set

Our π_{klt} analyzer saves the state space in the form of a tree and we can perform the analyses on that tree. We used a queue to traverse the state space tree using standard BFS. While checking for alpha-equivalence, the traversing may increase the time complexity. Let us describe the problem with an example. Assume that in a state space tree there are nine nodes. A new state is generated at this point and it is found to be equivalent to the ninth node in the tree. We discard the node from being included in the tree. Even though this is an example of the worst-case scenario (linear time search complexity O(n)), we can improve the performance of our analyzer by a more efficient

implementation of the seen set. We can use a hash table with process lengths as keys (average search time complexity is O(1)). Figure 4.12 shows an example.

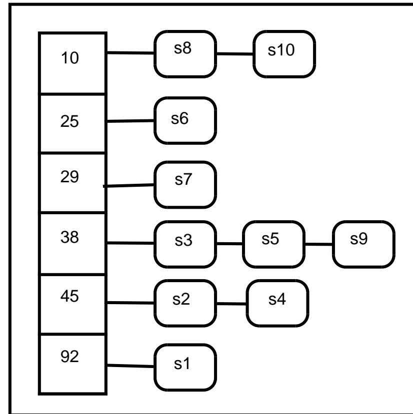


Figure 4.12: Hash table

The first column in Figure 4.12 shows the length of the processes. If we get a process P of length 25, we have to compare P with the process in `s6`. If they are found alpha-equivalent, P is discarded, otherwise we create a new π_{klt} state `s11`, and add it after `s6`. This way, we reduce the number of alpha-equivalence checks. If we get another process Q of length 26 we have to compare with the length first. If the length is already in the table, we have to check for alpha-equivalence with the processes of that length. As 26 is not the table, we now know that process Q is not alpha-equivalent to any of the states we have seen before. So, we create a new bucket in the table with 26 as the key and create a state with Q and add it to the newly created bucket.

Chapter 5

Evaluation of the π_{klt} analyzer

In this chapter we demonstrate the effectiveness of our π_{klt} analyzer by using three case studies. The first example is a simple program that deadlocks immediately. The second example expresses the classical Token Ring algorithm in kiltera. The final problem is taken from the transformation of a UML-RT state machine to a kiltera model.

5.1 Case study: finding a deadlock

5.1.1 General description: death model

This is a simple problem where two processes wait forever to get a message from other processes or the environment. If the processes do not receive any message, they cannot progress further. The situation creates a deadlock. Let us consider two processes in kiltera named process `A` and process `B`. Figure 5.1 shows the two processes and how they are connected.

Process `A` and process `B` are connected via the ports `x` and `y`. Port `x` of process `A` is hooked up with the port `x` of process `B` via channel `z1`. Similarly, the `y` ports are connected via channel `z2`. Both processes wait to receive a message and thus create a deadlock state. The kiltera model for the problem is shown below:

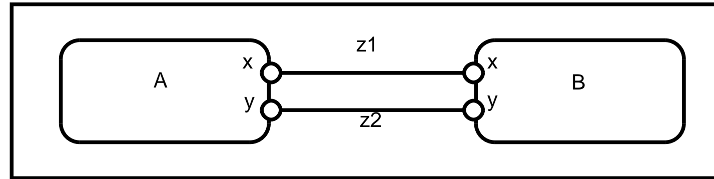


Figure 5.1: Death: an example of deadlock

```
1  module Death:
2  process A[x,y]:
3    when x →
4      trigger y.
5
6  process B[x,y]:
7    when y →
8      trigger x.
9  in
10   event z1,z2 in
11     par
12       A[z1,z2]
13       B[z1,z2]
```

5.1.2 Analysis question: death model

The example model shows that there is a state where both processes wait for a message from any source (either environment or from the other process). Can we find the deadlock using our π_{klt} analyzer?

5.1.3 Analysis output: death model

As the model is simple and small, let us first look at the state space tree in Figure 5.2 for the Death model. We present a portion of the state space tree to illustrate the analysis output.

The nodes are represented by the rounded corner rectangular boxes. The upper part of the boxes contains the AST and the lower part holds the index number of the node in the tree. The index number indicates the order in which the next states are generated by the analyzer. To shrink the size of the tree, for the first few levels, we put the node in the same horizontal line (only one node per level). Node 0 and node 2 do not contain the ASTs in Figure 5.2 because they are almost the full size of the model. The first three steps are common for all the examples. Node 0 is the model itself, node 1 is the call of the module of the model, and node 2 contains the process definitions inside a module. The nodes represented by the dotted rectangles are generated by the open system analysis only (open system analysis generates the entire state space tree). In other words, closed system analysis takes into account τ transitions only. Thus, in a closed system analysis, the dotted rectangular boxes will not be produced. As soon as node 10 is generated, our analyzer determines that the node is alpha-equivalent to node 8. Node 10 will be discarded from the tree, and will not be explored further. The arrows below the leaves of the tree indicate that we cut the portion of the entire tree to fit in a page.

Exhaustive mode

Let us answer the analysis question posed in Subsection 5.1.2. As we can see node 8 is the node where both processes wait for input. If we run our analyzer using open system analysis, this deadlock state (node 8) cannot be detected. This situation is desirable as we do the open system analysis to generate all possible paths of execution.

If we run the analyzer for a closed system, we will get rid of the states that are reached with non-τ first actions. This closed system analyses will lead us to node 8 and now there is no other path that we can explore. This is the deadlock state. Our analyzer will report the path to the deadlock state.

The analysis parameters for closed system analysis are:

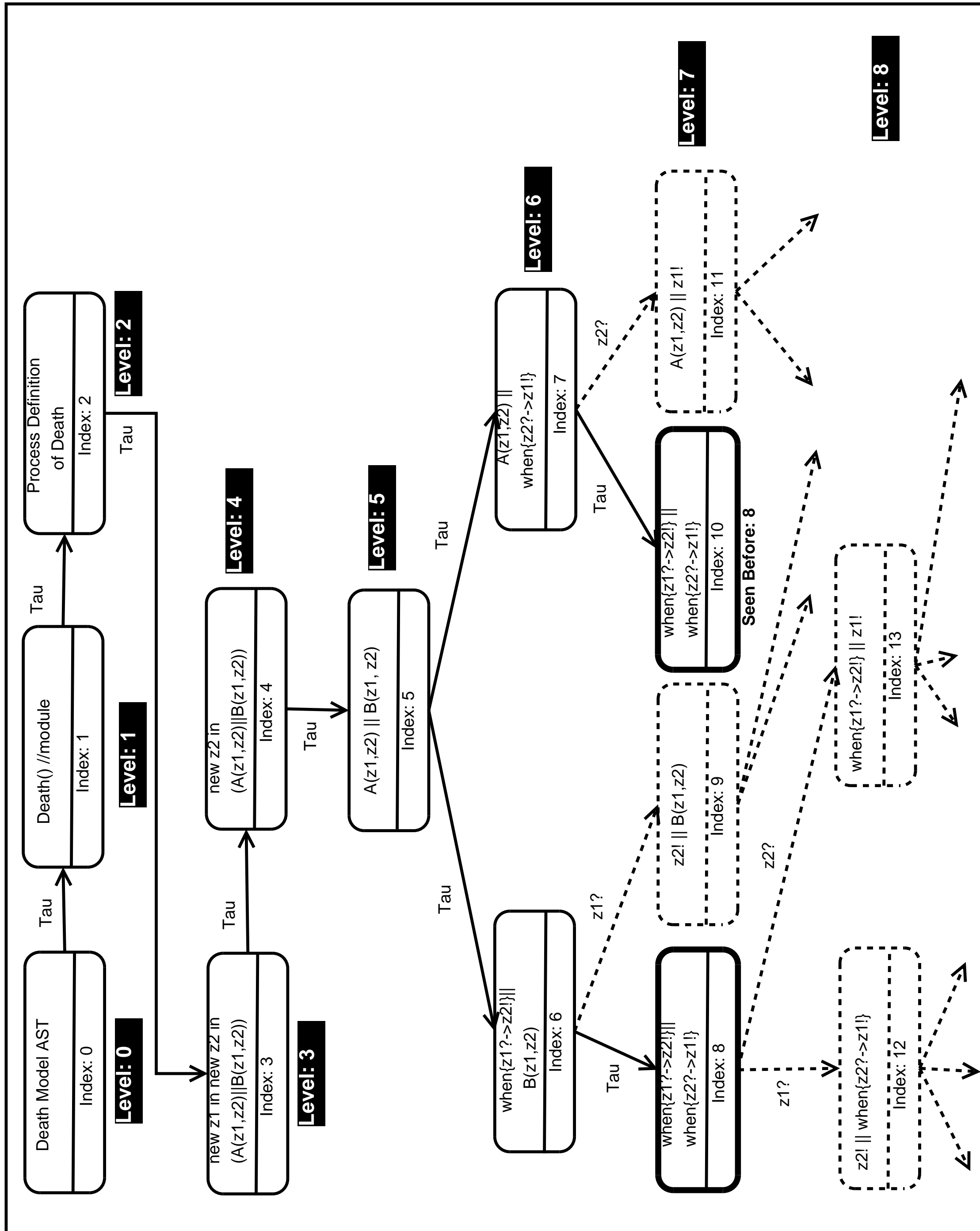


Figure 5.2: Portion of the state space tree for the Death model

```
exhaustive = True
time = 1000 # Time bound(disabled if exhaustive is true)
depth = 40 # Depth bound(disabled if exhaustive is true)
trigger='' # No triggers to search
receiver='' # no receivers to search
deadlock = True # we want to find deadlock state
stable = False
closed = True # Analyzer will run closed system analysis
```

After the execution of the analyzer is finished we will get a summary of the analysis:

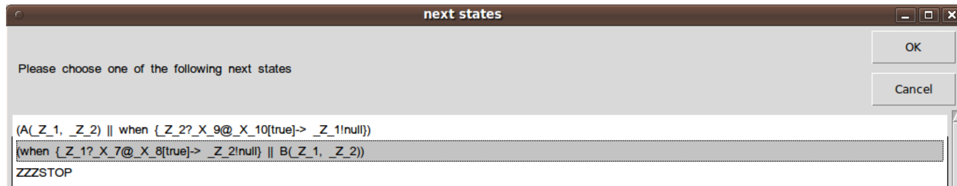
```
Check report folder: analyses finished.
---------------------------------------------
Time elapsed :  0.195577144623 seconds # CPU time
Total States :  9 # total number of states
Depth Reached:  7 # depth reached
Time Passed  :  0 # model time
---------------------------------------------
```

```
1
2 # Generating Report for Deadlock States
3 ================================================================================
4 Root: def {proc Death() = def {proc A(_Y_3, _Y_4) = when {_Y_3?_X_2@_X_3[true]-> _Y_4!null},proc B(_Y_5, _Y_6) = when {_Y_6?_X_4@_X_5[true]-> _Y_5!null}} in new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))} in Death()
5 Action: STau()  :  'def {proc Death() = def {proc A(_Y_3, _Y_4) = when {_Y_3?_X_2@_X_3[true]-> _Y_4!null},proc B(_Y_5, _Y_6) = when {_Y_6?_X_4@_X_5[true]-> _Y_5!null}} in new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))} in Death()'
6 Term: Death()
7 Action: STau()  :  'Death()'
8 Term: def {proc A(_Y_3, _Y_4) = when {_Y_3?_X_2@_X_3[true]-> _Y_4!null},proc B(_Y_5, _Y_6) = when {_Y_6?_X_4@_X_5[true]-> _Y_5!null}} in new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))
9 Action: STau()  :  'def {proc A(_Y_3, _Y_4) = when {_Y_3?_X_2@_X_3[true]-> _Y_4!null},proc B(_Y_5, _Y_6) = when {_Y_6?_X_4@_X_5[true]-> _Y_5!null}} in new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))'
10 Term: new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))
11 Action: STau()  :  'new _Y_1 in new _Y_2 in (A(_Y_1, _Y_2) || B(_Y_1, _Y_2))'
12 Term: new _X_6 in (A(_Z_1, _X_6) || B(_Z_1, _X_6))
13 Action: STau()  :  'new _X_6 in (A(_Z_1, _X_6) || B(_Z_1, _X_6))'
14 Term: (A(_Z_1, _Z_2) || B(_Z_1, _Z_2))
15 Action: STau()  :  'A(_Z_1, _Z_2)'
16 Term: (when {_Z_1?_X_7@_X_8[true]-> _Z_2!null} || B(_Z_1, _Z_2))
17 Action: STau()  :  'B(_Z_1, _Z_2)'
18 Term: (when {_Z_1?_X_7@_X_8[true]-> _Z_2!null} || when {_Z_2?_X_11@_X_12[true]-> _Z_1!null})
19
20 This state has been matched 0 times
21
22
23 The state is reach after 0 unit of simulation time from root.
```

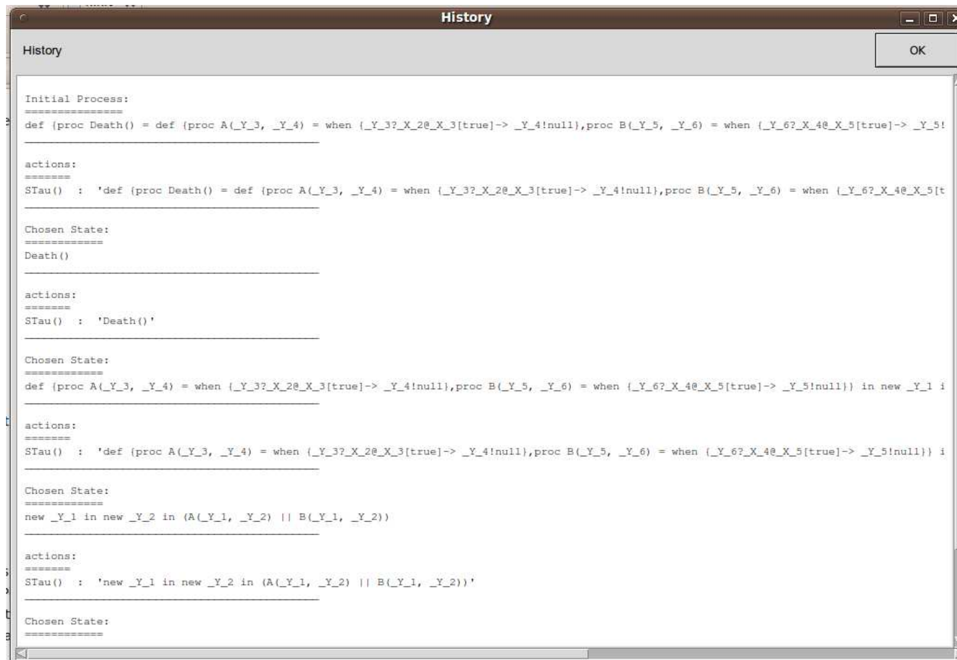
Figure 5.3: Deadlock state report screen shot

Figure 5.3 shows a screen shot of the report.

(a) Choose among the first actions screen shot



(b) Choose among the next states screen shot



(c) History of interactive analysis

Figure 5.4: Interactive mode screen shots

Interactive mode

If we use the interactive mode of analysis, we can choose any branch of the state space tree and explore the path. Figure 5.4 shows screen shots of the dialogs in interactive mode. When we execute our analyzer using interactive mode, the analyzer presents the set of first actions to the user and asks him to choose an action. Figure 5.4a shows the dialog to choose between the first actions of the process. Figure 5.4b shows the dialog to choose the next state among the possibilities. Finally, Figure 5.4c shows the path that the user has chosen so far. When the user chooses to stop or there is no other state to explore, the analyzer stops and allows the user to save the trace of the interactive analysis.

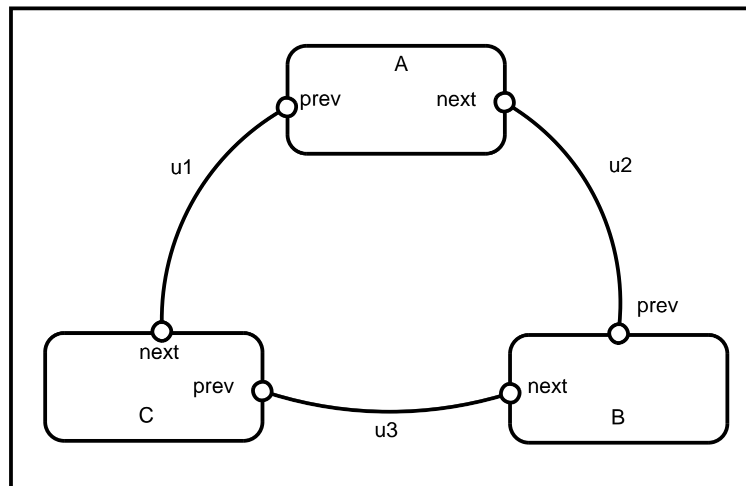


Figure 5.5: Token ring example

5.2 Case study: token ring algorithm

5.2.1 General description: token ring model

The token ring algorithm is a classical algorithm to achieve mutual exclusion in a distributed system. These processes form a logical ring and each process is assigned a position in the ring. Each of the processes knows which process is next in line after itself. The algorithm works as follows:

Process 0 has the token when initialized. The token circulates around the ring. When a process has the token, it checks to see if it is trying to enter the critical section (CS). If so, the process enters the CS, performs the operation in the critical section and then exits from the CS. After exiting, the process passes the token to its successor with whom the process is connected. The process can not release the token while it is in the CS. The process is not allowed to enter the CS more than once using the same token. If a process is handed the token by its neighbor and is not interested in entering the CS, it just passes the token along to the next process.

Figure 5.5 shows the token ring example for three processes A, B and C. Each of the processes in Figure 5.5 can be represented as a kiltera process. All the processes have two ports: `prev` and `next`. The `next` port of process `A` is hooked up with the `prev` port of process `B` through the channel `u2`.

Similarly, other ports are connected via channel `u3` and `u1`. The kiltera model for the token ring example is given below:

```
1 module TokenRing:
2 process A[prev, next](cs_delay):
3   when prev →
4     print ("A : Entering critical section")
5     wait cs_delay
6     print ("A : Exiting critical section")
7     trigger next
8     A[prev, next](cs_delay)
9 process B[prev, next](cs_delay):
10   when prev →
11     print ("B : Entering critical section")
12     wait cs_delay
13     print ("B : Exiting critical section")
14     trigger next
15     B[prev, next](cs_delay)
16 process C[prev, next](cs_delay):
17   when prev →
```

```
18      print ("C : Entering critical section")
19      wait cs_delay
20      print ("C : Exiting critical section")
21      trigger next
22      C[prev, next](cs_delay)
23 in
24    event u1, u2, u3 in
25      par
26        A[u1, u2](2.0)
27        B[u2, u3](5.0)
28        C[u3, u1](3.0)
29        schedule u1 after 2.0.
```

Each of the processes is given a time period to stay in the critical section. For process `A`, `B` and `C` the time periods are 2.00 seconds, 5.00 seconds and 3.00 seconds (lines 26, 27 and 28). In line 29 we trigger an event `u1` after waiting 2.00 seconds. This triggering can be interpreted as assigning the first token to the process `A`. After the trigger process `A` enters the CS and passes the token to process `B`. The token then starts to circulate among the processes in the ring.

5.2.2 Analysis questions: token ring model

We can ask several questions for analysis:

- Can we find a stable state, i.e., a state that does not have any τ actions?
- Can we reach a deadlock state, i.e., a state that does not have any action?
- How much time does it take for the token to move from process A to process B?
- Can we find a state such that there is more than one process involved in the critical section at the same time? If we can find such a state, the model is unsafe.

5.2.3 Analysis output: token ring model

Exhaustive and bounded mode

Let us generate the state space tree in bounded mode. As the processes in the model are parallel, the size of the state space will grow exponentially in the exhaustive mode. Moreover we use closed system analysis. The summary of the output we get is:

```
Check report folder: analysis finished.
--------------------------------------------
Time elapsed :  0.253895044327 seconds # CPU time
Total States :  30
Depth Reached:  23
Time Passed  :  12.0 # model time
--------------------------------------------
```

For the same model, the open system analysis output will be (bounded mode: depth = 20, model time = 20):

```
Check report folder: analyses finished.
--------------------------------------------
Time elapsed :  6.48799085617 seconds
Total States :  658
Depth Reached:  20 # depth reached
Time Passed  :  17.0
--------------------------------------------
```

Let us answer the questions posed in the Subsection 5.2.2.

Can we find a stable state, i.e., a state that does not have any τ actions?

If we run our analyzer using closed system analysis, we will not find any stable state. In open system analysis, our π_{klt} analyzer has found one stable state. The model was designed so that there will always be an interaction between the processes. As the token is passed from process to process, the

result of the analysis is unexpected. Let us look at the trace to the stable state in Figure 5.6. For simplicity and readability we present a portion of the state space tree.

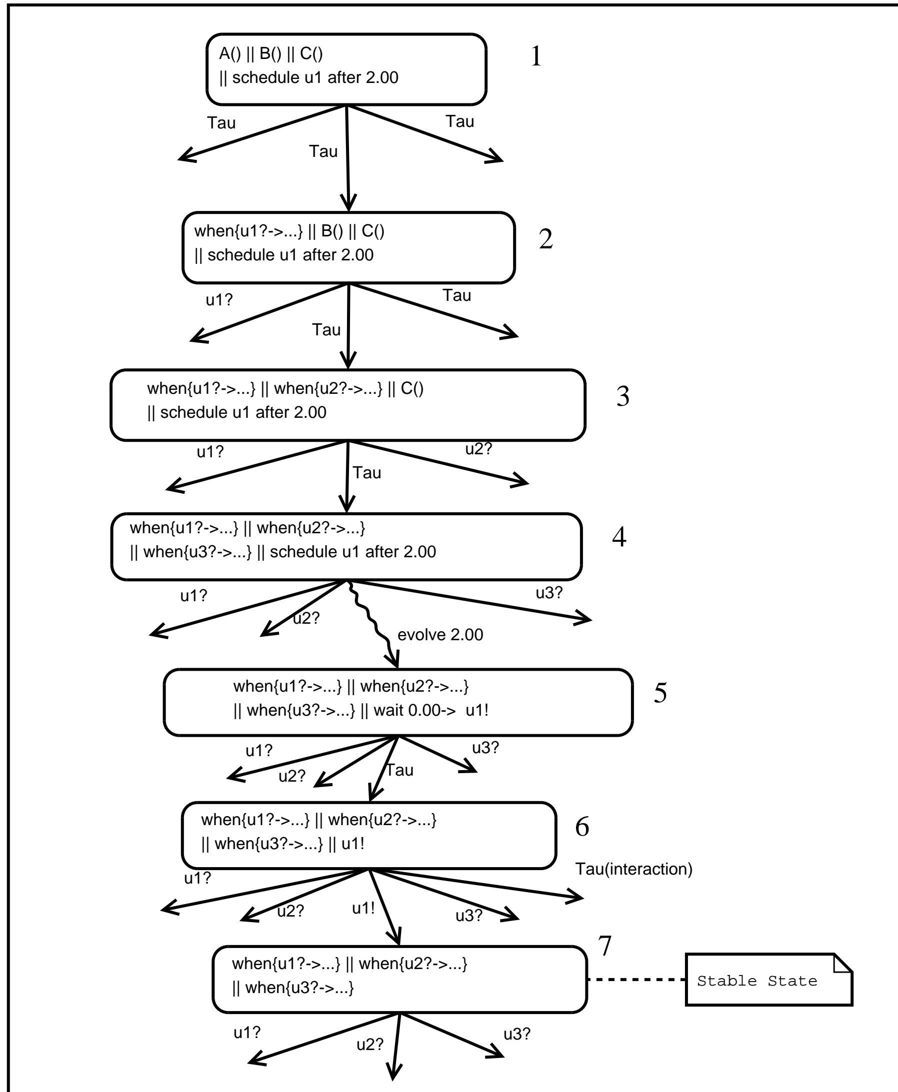


Figure 5.6: Trace to a stable state

After the first few steps of execution we will reach to the state marked as 1 in the Figure 5.6. This state has three τ actions (τ action for process description). When we reach state 4, the processes are ready to receive the token. Then we wait for 2.00 seconds (state 5) and pass the first token to process A (state 6). The passing of the token would result in an interaction between the environment

and process A (τ action of state 6). But this is an open system analysis. So, we can see that if the token is taken away by the environment, we can reach a state which is stable (state 7). State 7 does not have any τ action. It indicates that the processes will be waiting for the token from the environment.

Can we reach a deadlock state, i.e., a state that does not have any action?

After the execution of an exhaustive closed system analysis, our analyzer reports that there is no deadlock state in the model. In case of bounded (for depth = 20 and time = 20) open analysis, our analyzer also did not find any deadlock.

How much time does it take for the token to move from process A to process B?

We set the specific receiver to find message exchange over the channel `u2`. Our analyzer produces a trace that leads to a state where channel `u2` receives a message after 4.00 units of time.

This analysis is simple. In our model we first wait 2.00 seconds and then passes the token to process A. Process A waits for 2.00 seconds, and pass the token to process B over the channel `u2`. The resulting time is 4.00 seconds which is shown by our analyzer.

Can we find a state such that there is more than one process involved in the critical section at the same time?

We set the parameters `u1`, `u2` and `u3` in the set to search all the events in a state. If we can find a state in the state space tree where all the events are ready to be triggered by the processes, that means we have more than one processes at the same time in the CS. The analyzer shows that there is no such state where all the events are ready to be triggered. This result implies that our model is safe. No two processes can be in the CS at the same time for a model to be safe.

5.2.4 Simple scalability analysis of π_{klt} analyzer

To evaluate the scalability of the analysis of the token ring, we have modified the token ring example. Each process now stays in the critical sections for 5.00 seconds. We have tested the performance of our π_{klt} analyzer for varying number of processes. We run our π_{klt} analyzer on Ubuntu 9.10 on an Intel Pentium 4 machine (2.8 GHz clock speed, 1.5 GB memory). We started with three processes,

and then incremented the number of processes by one. To do the performance evaluation, we have chosen the closed-system analysis mode. Table 5.1 shows the comparison for different numbers of processes.

Number of processes	CPU time (seconds)	Maximal model time	Total states	Depth reached
3	0.23	17	30	23
4	0.42	22	44	29
5	0.93	27	66	35
6	2.29	32	104	41
7	7.28	37	174	47
8	22.04	42	308	53
9	66.48	47	570	59
10	261	52	1088	65
11	958.07	57	2118	71
12	3544.07	62	4172	77
13	9647.87	67	8274	83
14	28990.28	72	16472	89
15	74268.68	77	32862	95

Table 5.1: Performance evaluation of the π_{klt} analyzer

′CPU time′ is the actual analysis time taken by the analyzer to generate the state space tree. ′Maximal model time′ is the largest model time of all explored states. ′Total states′ is the total number of states generated by the analyzer. ′Depth′ is the depth of the tree.

For a small number of processes the time taken is really small. As the number of processes increases, the state space tree grows in size. When the number of processes is more than 10, the total number of states almost doubles every time we add a new process to the network.

Figure 5.7 shows that the time required to generate the state space tree for the input model of varying number of processes. When the number of processes is 15, the required time is approximately 20 hours. This figure indicates the exponential growth of the state space tree.

5.3 Case study: dropped signal analysis

In this example we are going to show how a UML-RT state machine can be analyzed. As part of the transformation [15], we get a transformed kiltera model of a UML-RT state machine. The

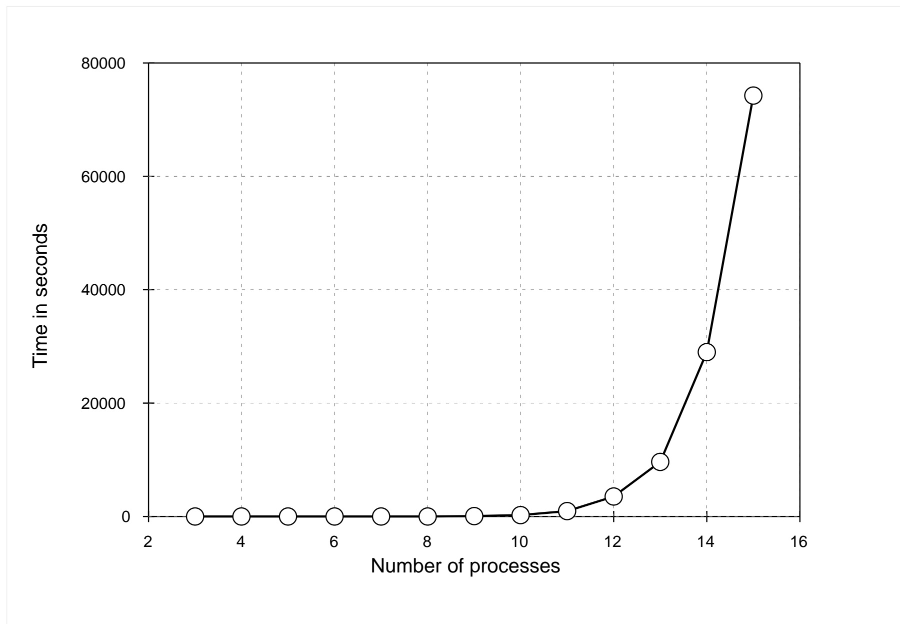


Figure 5.7: Process vs time

transformation currently supports single capsule UML-RT model. Therefore we present a UML-RT model which contains a single capsule with some behavior.

5.3.1 General description: model for dropped signal analysis

The UML-RT model contains a single capsule called `Device`. Its behavior is shown in Figure 5.8. The `PrimaryStateMachine` process corresponds to the `Device` capsule's behavior.

After the initialization of the capsule, the control goes into the `Sleep` mode. When the environment triggers the `start` signal to the capsule, it goes into the `Active` state. The `Active` state is a composite state containing two basic states. In the state `Unlocked` the `lock` signal from the environment causes a transition to the state `Locked`. Similarly, the transition from the `Locked` state goes to the `Unlocked` state with the `unlock` signal from the environment. Within the `Active` state, if the `Device` receives a `shutdown` signal from the environment, it goes into the `Sleep` state and if it receives an `error` signal, it goes into a `Diagnostic` state. The transition from the `Active` state to the `Sleep` state or the `Diagnostic` state is a group transition. The group transition implies whether the control is in the `Locked` state or in the `Unlocked` state, whenever the signal `shutdown` or `error`

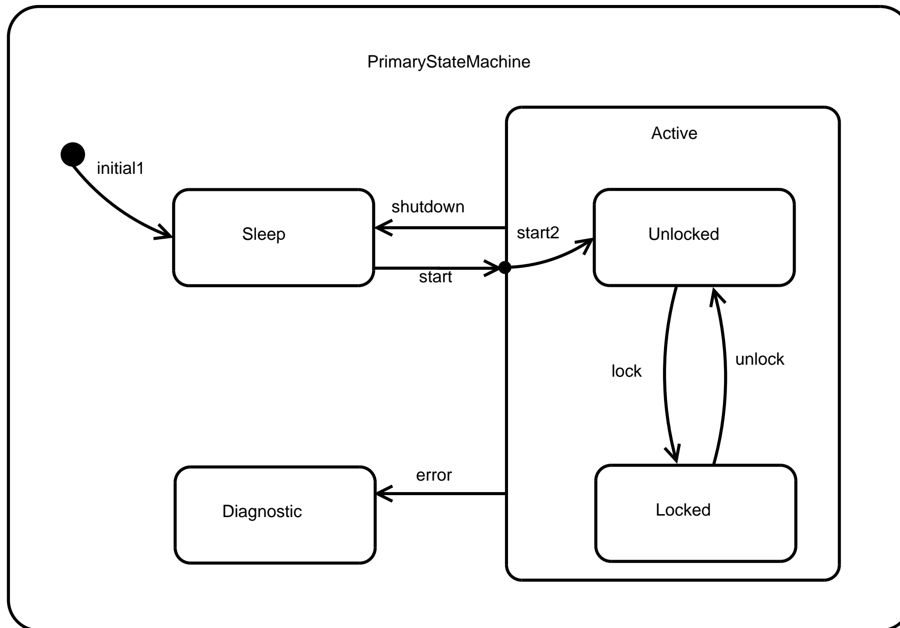


Figure 5.8: State machine of the UML-RT model

is triggered, the control goes into the `Sleep` state or the `Diagnostic` state. The transformed kiltera model of the UML-RT state machine is given in the Appendix A.

From the environment, we trigger four signals in the following order: `start`, `unlock`, `shutdown` and `error`. This triggering order is intentional. Once we are in the `Unlocked` state, if we receive an `unlock` signal, the signal will be rejected by the `Unlocked` state. We can only move to the `Locked` state with the signal `lock` from the `Unlocked` state. Therefore this `unlock` signal will be dropped. Similarly, once we are in the `Sleep` state, if we receive the `error` signal, it will also be dropped.

5.3.2 Analysis question: dropped signal analysis

We have devised a model that allows for dropped signals. Our analysis question is: can we find the state or states where a signal has been dropped?

5.3.3 Analysis output: dropped signal analysis

Before discussing the analysis output, let us describe in brief the transformation technique from a UML-RT state machine to a kiltera model. Each of the states in the UML-RT state machine is represented as a process definition in kiltera. Each state in kiltera has a handler associated with it. Defined within the primary state machine and in every state are event handler processes. In basic states, the event handler is simply the executed process. The event handler is a listener that handles state machine inputs and internal control signals, executing transition processes as necessary and terminating upon state exit. So, whenever the handler rejects a signal, the handler triggers a dropped signal, and we can find a signal that has been dropped.

To find the dropped signal, let us set the parameters as follows:

```
exhaustive=True
time=100 #disabled
depth =40 #disabled
trigger='dropped'
receiver=''
deadlock = True
stable = False
closed = True
```

We set to search for a trigger named dropped. Our analyzer runs the closed system analysis. The summary of the analysis is:

```
Check report folder: analysis finished.
-------------------------------------------
Time elapsed :  19.5355598927 seconds # CPU Time
Total States :  301
Depth Reached:  55
Time Passed  :  7.0 # Model time
```

```
------------------------------------------
```

The analyzer produces two reports. The first report shows the kiltera states where the dropped signal has been found.

```
*******************************
Dropped Signal found------→ 1
Node Number: 211
Level: 46
time_passed: 4.0
*******************************
  ⋮
*******************************
Dropped Signal found------→ 30
Node Number: 302
Level: 54
time_passed: 7.0
*******************************
```

The first lines show in which state the first dropped signals has been found. It also provides the node number in the tree and the level of the node. The timing information indicates the model time that has passed to find a dropped signal. The segment below the vertical dots indicates the final state where our analyzer found another dropped signal. We have found 30 kiltera states where a signal has been dropped. Because of the parallel compositions of the processes and their branching, the total number of states is that high.

Another report produced by our π_{klt} analyzer contains the paths from the root to the states where the signals have been dropped. There are 30 paths. The output generated by our analyzer shows the paths in the kiltera model. So we filter the output to determine which paths in the UML-RT state machine our kiltera model outputs correspond to. We simplify the output and show the path to the first dropped signal in Figure 5.9 (we show one path only).

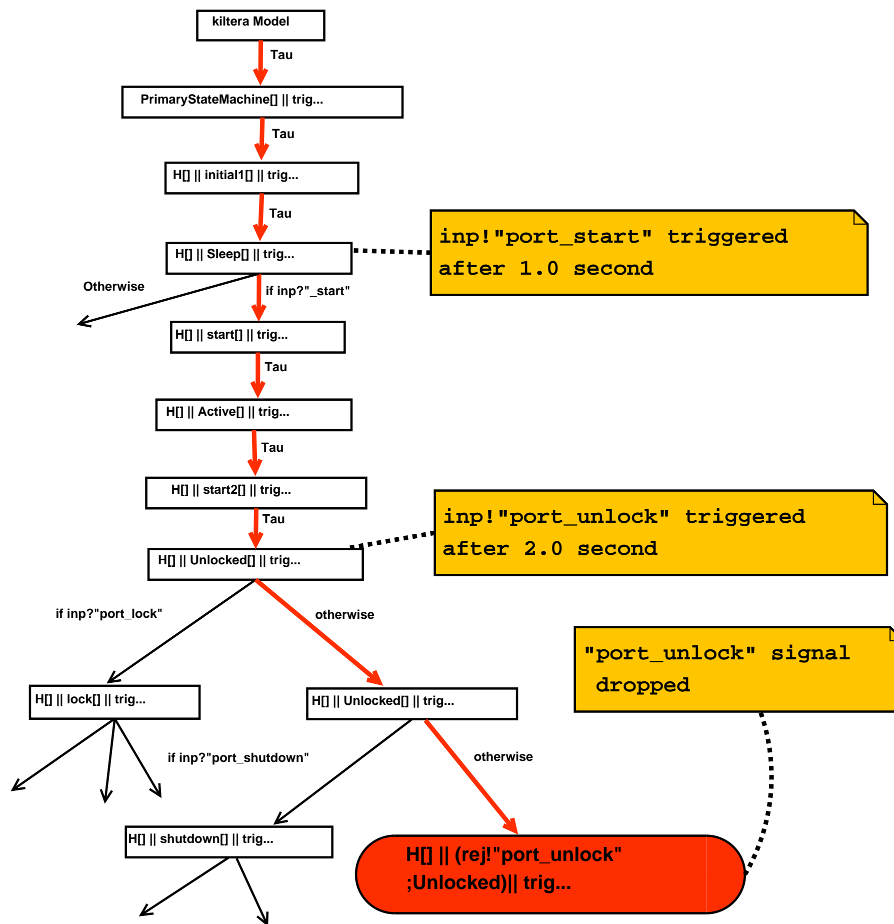


Figure 5.9: Dropped signal analysis

The corresponding path in the UML-RT state machine is:

```
Dropped Signal: unlock

Path: PSM → Sleep → Active → Unlocked

Depth Reached: 3

Time Passed : 4.0 (Model Time)
```

Chapter 6

Related work

The purpose of this thesis has been to analyze the models of real-time embedded software systems. To achieve this goal, we constructed the core of kiltera [53, 56] which we call π_{klt}[55]. The theory of π_{klt} is based on the π_{klt} calculus which is a real-time extension of the π-calculus [42, 43]. π_{klt} terms are processes which can use three different actions: input actions, output actions and internal actions. Given a π_{klt} process our tool can search the state space of the input exhaustively. The tool can then perform several analyses based on user requirements such as deadlock detection, stable state search, specific trigger search etc.

There are quite a large number of tools existing [34] which can analyze models designed in various languages. The core of kiltera is presented in [55, 53] and is inherited from Timed-CSP [58]. In this chapter we will explore some of the other analysis tools that are closely related to our tool. We will discuss two perspectives for related works: the language perspective and the analysis tools perspective.

6.1 Language perspective

π_{klt} is a real-time extension of the π-calculus. The π-calculus is a continuation of work on the Calculus of Communicating Systems (CCS) [39]. π_{klt} can deal with time, and is closely related to

Timed-CSP which is an timed extension of Communicating Sequential Processes (CSP) [25, 60].

6.1.1 CCS

The Calculus of Communicating Systems (CCS) is a process calculus introduced by Robin Milner in 1980 [39]. The language is based on communications between exactly two participants. Terms of CCS represent processes. Processes can interact with each other. The interaction can have two agents or participants: either two agents or a agent and its environment. The nature of communication in CCS is synchronized. There are several process in CCS namely, null process, prefix process, choice process, parallel composition, hiding name operators and process declarations.

Even though CCS was the one of the first attempts to define process calculi, it has several limitations [42, 43]. These limitations guided Milner to conceive the extension of CCS by allowing link mobility which had been named π-calculus.

6.1.1.1 The π-calculus

The π-calculus [41] terms are also processes that can interact with each other like CCS. But the fundamental difference is as the processes interact their interconnection may change due to link mobility. However the communication is still synchronous like in CCS and the π-calculus also allows communication between two participants. The processes are similar to those in CCS but now the hiding of names from a process is represented in form of declaring channel names, which hides that channel from the environment. Also, processes can now take parameters as input. The main limitation of the π-calculus is the lack of support for the modeling of time. The timed extensions of π-calculus are mentioned later in this chapter.

6.1.2 CSP

Communicating Sequential Processes (CSP), introduced by Hoare in 1978, was defined in the form of a programming language rather than a process calculi [25]. The original version did not have a mathematically defined semantic. A CSP program was the parallel composition of sequential

processes, where each process interacted with other processes using synchronous message passing. Later Hoare, Brookes, and Roscoe [6] came up with a new version of CSP that refined CSP into a modern process algebraic form inspired by Milner's CCS.

CSP is the calculus of interactions between the components of a concurrent system [60]. CSP introduced several new operators such as deterministic and nondeterministic choice, interface parallel, interleaving, sequential composition etc.

CCS and CSP have a lot of similar constructs which is understandable as both the languages support concurrency. Composition of CCS terms is done from an operational viewpoint using as observational bisimulation.

6.1.2.1 LOTOS

The Language Of Temporal Ordering Specification (LOTOS) was introduced in [33] in 1988 in response to a need in the telecommunication industry. LOTOS was designed to represent telecommunication protocols with varying degrees of formality. It has two main parts: a process algebraic part which is similar to CCS and CSP, and a data algebraic part based on the abstract data type language (ACT ONE) [32].

LOTOS has similarities and dissimilarities with CCS and CSP. For example, in terms of parallel composition, LOTOS and CSP are similar and LOTOS and CCS are different. In terms of choice the relationship is opposite between LOTOS, CCS and CSP [17]. Because of its refusal-based semantics LOTOS is more closely related to CSP than to CCS. In fact LOTOS refusal based semantics is based on CSP failures.

6.1.2.2 Timed CSP

Timed CSP is the first real-time extension of CSP proposed by Reed and Roscoe in [58]. The authors considered the time domain to be continuous, the processes should engage in only finitely many events, all semantic operators should be continuous and all the basic operators should be distributive except recursion and the model design should be verifiable. Later there had been a lot of research done on continuous and discrete time process algebra [49].

6.1.3 π_{klt}

π_{klt} is a real time extension of π-calculus [55] similar to timed CSP. π_{klt} is a combination of the constructs of CCS and CSP. Unlike CCS, π_{klt} communication can be asynchronous and can deal with time like timed CSP. π_{klt} processes have input actions, output actions and internal actions. This form of communication is similar to CCS and π-calculus. Neither CCS nor π-calculus can deal with time and to the best of our knowledge we did not find any previous work on timed π-calculi that has studied time-bounded equivalence [55].

6.1.4 Other timed extensions of CCS and CSP

The main version of CCS supports time and later quite a large number of works have been done to extend CCS with time. Moller introduced a Temporal Calculus of Communicating Systems (TCCS) [44] which can model real-time processes by allowing the expression and analysis of timing constraints. Wang developed an interleaving model for real–time systems by extending CCS with a notion of time [67]. Other timed extensions of CCS can be found in [48].

Like CCS, the original version of CSP did not have the time construct either. We already described the timed extension of CSP in Subsection 6.1.2.2. The work of Reed and Roscoe [58] was quickly followed by others who proposed several process calculi based on dense-time and discrete time. Few of the timed extensions on CSP are mentioned in [1, 23, 26, 27]. A brief survey of the work on timed CSP is presented in [49] and [14] describes in detail a few of the pioneering works on timed CSP.

Leonard proposed Enhanced Timed-LOTOS, denoted ET-LOTOS in [31], as an extension of LOTOS. ET-LOTOS can model real-time behaviors. It supports all the features of LOTOS. It covers both dense time and discrete time, and time values can be seen as other data values. Bowman also came up with another timed extension of LOTOS in [5] which employed true concurrency semantics.

There has been a lot of research done on the extension of π-calculus with time. Most of the extensions deal with discrete time [55]. The stochastic π-calculus [57] introduced *rate* the to input and output actions of π-calculus, which allowed the delay of an action for an amount of time which is drawn from an exponential distribution. The πRT-calculus [30] extends the π-calculus by introducing time-out operator using discrete time. Additional extensions of the π-calculus are discussed in [55].

6.2 Tools perspective

There are many existing tools for analyzing the behaviors of the models designed by the languages mentioned in Section 6.1. We will describe a few of the tools which are most closely related to our work. Note that most of these tools have been worked on for a long time and are very mature. They are therefore more user friendly and optimized than our prototype. However, none of them is capable of analyzing kiltera model.

6.2.1 ProBE

Process Behavior Explorer (ProBE) [16] is an interactive tool offered by Formal Systems (Europe) Limited [18]. The user can select a state of a CSP program, and can choose among the next possible actions to go to the next state in ProBE. The user controls the choice of the possible next actions of a CSP process and can watch the process evolve accordingly. π_{klt} has an interactive simulation mode similar to ProBE, where the user can select among the possible first actions of the given input process and then select the next state from a set of states generated from the input process and action.

6.2.2 FDR2

Failures-Divergences Refinement (FDR) [59] and subsequently FDR2 are two other tools from Formal Systems (Europe) Ltd [18]. FDR and FDR2 are the tools for refinement checking of models expressed in CSP. FDR2 takes two CSP processes as input and converts them into Labeled Transition Systems (LTS) and checks for refinement between the two processes. It supports three models: the trace model, the stable failures model and the failures/divergence model [59].

6.2.3 The concurrency workbench

The Concurrency WorkBench (CWB) [11, 12] is a tool for verifying systems written in CCS. CWB has one equivalence checking routine based on bisimulation equivalence [40] and one preorder checking

routine based on divergence preorder [66]. CWB has the model checking facility for determining whether a process satisfies a formula expressed in temporal logic. It uses a special technique to reduce the state space of processes by rewriting of the CCS expression. In CWB it is possible to express a process behavior in Timed CCS [44] and Synchronous CCS and then analyze the behavior of the processes in two different modes of simulation: exhaustive mode and interactive mode.

6.2.4 The mobility workbench

The Mobility WorkBench (MWB) is an automated tool for describing and analyzing mobile processes. The basic functionality of MWB is to decide open bisimilarity [65]. MWB supports two different languages: the π-calculus and the fusion calculus [51] which is a simplification of the π-calculus with significantly reduced expressiveness. MWB does model checking for the π-calculus and hyper-equivalence checking for the fusion calculus. It also has two different modes of operations: the exhaustive state space generation and the interactive walk through in the state space for a given input.

6.2.5 Construction and analysis of distributed processes

Construction and Analysis of Distributed Processes (CADP) [21] is a tool set for the execution, verification and analysis of communication protocols and distributed systems. CADP uses LOTOS as an input language. CADP offers two modes of operations as well: step-by-step simulation and massively parallel model-checking. It has compilers for several input formalisms: high-level protocol descriptions written in the ISO language LOTOS, low level protocol descriptions specified as finite state machines and networks of communicating automata. CADP has several equivalence checking tools [64, 2], model checkers [36, 37] and other tools with advanced functionalities such as visual checking, performance evaluation, etc [22].

6.2.6 Other tools

Apart from CWB, there exist other tools for analyzing and verifying CCS processes. The TAV tool [24] has the functions for deciding various notions of bisimilarity between processes. It is a model

checker which expects properties of the model to be expressed in Hennessy-Milner Logic extended with recursion [29]. The system AUTO [4] is another tool for CCS process analysis which allows checking for MEIJE term [3], for reducing automata using various bisimulation equivalences, for transforming automata along behavioral abstractions, and for various kinds of analysis obtained by automata. Madelaine summarized some other CCS process analysis tools in [34].

Since CSP has been proposed, researchers around the world have proposed analyzers of CSP models. Of course, FDR2 and ProBE are the most popular of the tools, there are some other helpful tools as well. The Deadlock Checker [35] proves freedom from deadlock or livelock for the parallel programs written in CSP. The tool can scale and can be used to verify networks built from thousands of processes. The Adelaide Refinement Checker (ARC) [50] is a CSP refinement checker. This tool also checks for refinement but it differs from FDR2 by representing the CSP processes as Ordered Binary Decision Diagrams (OBDDs) [7] internally. This mitigates the state explosion problem and does not require any additional algorithms for state space compression like FDR2. The Process Analysis Toolkit (PAT) [63] is another tool for CSP processes which complements FDR in many ways: 1) PAT also does the refinement checking like FDR2. But in FDR2 the specification is normalized beforehand which is computationally expensive. Instead, PAT uses an *on the fly* normalization technique. 2) Unlike FDR2, PAT uses LTL based model checking. PAT also supports an *on the fly* model checking. 3) PAT has several simulation modes: random simulation, user-guided step-by-step simulation and system graph generation.

The Concurrency WorkBench-New Century (CWB-NC) [10] is a reimplementation of the Edinburgh CWB which offers similar functionality but is faster, memory efficient and provides diagnostic information when a verification routine fails. CWB-NC supports various types of verifications in contrast to CWB and other tools such as reachability analysis, model checking with respect to temporal logic formulas, verification using the same design language for the system and specification. Moreover it supports the following languages: CCS, SCCS, TCCS, PCCS, CSP and LOTOS [13, 10].

Real-Time Maude [46] is implemented in Maude [9] as an extension of full Maude. Real-time Maude is a language to formally specify real-time and hybrid systems. The underlying formalism is based on rewriting logic. The rewriting logic formalism is flexible and general [38] resulting in ease of specification. Real-time Maude provides precise formal specification of the system, allows

the specification to be analyzed and allows the user to define the form of communication at a high level of abstraction [47]. The tool for Real-time Maude offers a wide range of analysis techniques: timed rewriting for simulation purposes, untimed and time-bounded search for reachable states from initial state and time-bounded linear temporal logic model checking.

Chapter 7

Conclusion and future work

In the following sections we are going to summarize our achievements, the limitations of our work and we will provide some future work.

7.1 Our contribution

7.1.1 π_{klt} analyzer

The main contribution of this thesis is the design and implementation of an analyzer for kiltera models which supports several kinds of analysis, such as:

- deadlock: We search for the π_{klt} states that do not have any action, and the execution cannot progress further. Our analyzer reports these deadlock states and provides the path from the root to these states.
- stable states: Our analyzer searches for the states in the state space tree which do not have any internal transitions and reports the paths from the root to these states.
- timing analysis: We provide simple timing analysis. The user can identify the model times of each of the explored states and the maximal model time. If we search, e.g., deadlocked states, the analyzer tells the user the model time associated with that state.

- group of events: We can search for a group of events. We can search for all the events in a group, or just any event in a group.
- longest and shortest execution of time: Our analyzer can report which of the states in the state space tree have the longest or shortest model time. The analyzer reports the state with its corresponding path.
- maximum frequency: Our analyzer keeps record of how frequently a state in the seen set has been found alpha equivalent to a new state explored during the search.
- analysis of open and closed systems: Our π_{klt} analyzer can perform open and closed system analysis based on the user's choice. The open system analysis generates all the possible states, while the closed system analysis generates the states that are the result of internal interactions between the processes.

7.1.2 Transformation of models

We have identified methods to simplify a π_{klt} process. These methods are Deep Simplification and Shallow Simplification. These simplifications do not change the behavior of the process. These transformations are standard in the process algebra literature, but they have not been applied to the automated analysis of kiltera.

7.1.3 Removing redundant renaming

Renaming is a necessary step during the analysis of a kiltera model. We need renaming to implement the semantics of interaction and to perform transformations such as scope extrusion. However, this renaming can prove costly if the model is large. To reduce the cost of renaming of variables we have identified some cases where renaming is redundant. We have removed the redundant renaming to speed up the execution.

7.2 Limitations

Even though we tried to solve the problems of our analyzer regarding performance, we still have some issues. At the moment the analyzer does not scale to industrial size models. We have tested our analyzer for UML-RT models which have only one capsule. The number of states in these UML-RT model was not very large. A real time model may have hundreds of states in a state machine of a UML-RT model. In that case, our analyzer may take an unacceptably long time to complete. We did not perform stress testing on our analyzer.

Another issue is to overcome the problem of excessive renaming of a process during execution. We have tried to get rid of redundant renaming, but more could be done. Alpha equivalence checking also takes a long time whenever the state space grows. We used shallow simplification of processes in the implementation. We could implement normalization which would improve the performance of alpha equivalence checking.

We have used a hash table to implement the seen set with the length of process as hash keys. The performance may be improved using different data structures.

The user interface we provided is only command line based. We provided a prototype of an analyzer that can perform analyses on kiltera models. There is no visualization of output using the GUI. Our outputs are in textual form, and sometimes it is difficult to map states mentioned in the output to states in the analyzed process due to renaming. Even though we have a data dictionary that keeps the history of renaming, we did not use the dictionary to transform the output into actual processes. We used the dictionary only to search for a specific signal or a receiver in the state space tree.

7.3 Future work

There are many ways to improve the performance of the analyzer. We will discuss some of the possible avenues to extend the capabilities of our analyzer.

Equivalence checking

At present we only check for alpha equivalence between two processes. Alpha equivalence is a good and easy way to compare processes. Alpha equivalence checking can be replaced by bisimulation. Bisimulation provides a more expressive way to compare the behavior of two processes. This would be an interesting extension of our analyzer.

De Bruijn notation

Even if we use alpha equivalence in the future, the performance can be significantly improved using the De Bruijn notation [45]. This notation will provide a mechanism to get rid of renaming while implementing substitution. Instead, we just need to check for equality to guarantee the equivalence between the processes.

Analyses techniques

Although we have provided some analyses techniques, the π_{klt} analyzer can still be extended to more kinds of analysis. The current techniques can be improved as well. One of the important extensions can be to allow the analyzer to perform offline analyses.

Model checking

Our analyzer can be used to determine the complete state space of a kiltera model. The state space can be used for checking properties and perform model checking. The analyzer can be extended to perform assertion violation, safety and liveness checks, properties verification using Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [8].

7.4 Conclusion

In this thesis we have developed an analyzer for the high level language kiltera. After completion of the analysis we can see the analysis results in a abstract level. One of the main objectives of our

thesis was to develop an analyzer for kiltera models which are actually being transformed from UML-RT state machines. The models are complete executable source codes written in kiltera. Our π_{klt} analyzer is well capable of analyzing transformed kiltera models from UML-RT state machines and filtering the analyses output so that a user can map executions output by the analyzer to executions in the UML-RT state machines.

Bibliography

[1] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[2] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: A modular tool for on-the-fly equivalence checking. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 581–585, 2005.

[3] G. Boudol. *Notes on algebraic calculi of processes*, pages 261–303. Logics and models of concurrent systems. Springer-Verlag New York, Inc., 1989.

[4] G. Boudol, R. De Simone, and D. Vergamini. Experiment with auto and autograph on a simple case of sliding window protocol. *INRIA report*, 870, 1988.

[5] H. Bowman and J. Derrick. Extending LOTOS with time: A true concurrency perspective. *Transformation-Based Reactive Systems Development*, pages 383–399, 1997.

[6] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[7] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(35):677–691, 1986.

[8] E.M. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, page 54. Springer, 1997.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic. *Lecture Notes In Computer Science; Vol. 4350*, page 756, 2007.

[10] R. Cleaveland, V. Natarajan, S. Sims, and G. Luttgen. Modeling and verifying distributed systems using priorities: A case study. *Software-Concepts and Tools*, 17(2):50–62, 1996.

[11] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Automatic Verification Methods for Finite State Systems*, pages 24–37. Springer, 1989.

[12] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):72, 1993.

[13] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Computer Aided Verification*, pages 394–397. Springer, 1996.

[14] J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[15] J. Dingel, E. Paen, E. Posse, R.R. Rahman, and K. Zurowska. Definition and implementation of a semantic mapping for UML-RT using a timed pi-calculus. In *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, pages 1–8. ACM, 2010.

[16] P.B. Explorer. *ProBE user manual.* Formal Systems (Europe) Ltd, 2003.

[17] C. Fidge. A comparative introduction to CSP, CCS and LOTOS. *Software Verification Research Centre, University of Queensland, Tech. Rep*, pages 93–24, 1994.

[18] Formal Systems (Europe) Limited. http://www.fsel.com/.

[19] M. Fowler and K. Scott. *UML distilled: A brief guide to the standard object modeling language.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[21] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. 2001.

[22] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification*, pages 158–163. Springer, 2007.

[23] R. Gerth and A. Boucher. A timed failures model for extended communicating processes. *Automata, Languages and Programming*, pages 95–114.

[24] J.C. Godskesen, K.G. Larsen, and M. Zeeberg. Tav (tools for automatic verification): Users manual. Technical report, Institute for Electronic Systems, Department of Mathematics and Computer Science, The University of Aalborg, 1989.

[25] C.A.R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[26] A. Jeffrey. Abstract timed observation and process algebra. In *CONCUR'91*, pages 332–345. Springer.

[27] A. Jeffrey. Discrete timed CSP. Technical report, Programming Methodology Group Memo 78, Department of Computer Sciences, Chalmers University, 1991.

[28] D.E. Knuth, D.E. Knuth, R.E. Tarjan, and A. Goldberg. The Art of Computer Programming. *Journal ACM*, 35:921–940.

[29] K. Larsen. Proof systems for Hennessy-Milner logic with recursion. *CAAP'88*, pages 215–230, 1988.

[30] J.Y. Lee and J. Zic. On modeling real-time mobile processes. In *Proceedings of the twenty-fifth Australasian Conference on Computer Science-Volume 4*, pages 139–147. Australian Computer Society, Inc., 2002.

[31] L. Leonard and G. Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 10(3):248–266, 1998.

[32] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–342, 1992.

[33] ISO Lotos. A formal description technique based on the temporal ordering of observational behaviour. *International Organisation for Standardization-Information Processing Systems-Open Systems Interconnection, Geneva*, 1988.

[34] E. Madelaine. Verification tools from the CONCUR project. *Bulletin of the European Association of Theoretical Computer Science*, 47:110–126, 1992.

[35] J.M.R. Martin and S.A. Jassim. A tool for proving deadlock freedom. In *Proc. of the 20th World Occam and Transputer User Group Technical Meeting*, 1997.

[36] R. Mateescu. Local model-checking of an alternation-free value-based modal mu-calculus. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 98, 1998.

[37] R. Mateescu and H. Garavel. XTL: A meta-language and tool for temporal logic model-checking. *Software Tools for Technology Transfer (STTT'98)*, page 33, 1998.

[38] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[39] R. Milner. A calculus of communicating systems, volume 92 of Lecture Notes in Computer Science, 1980.

[40] R. Milner. *Communication and concurrency.* Prentice Hall International Ltd. Hertfordshire,UK, 1995.

[41] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[42] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[43] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and computation*, 100(1):41–77, 1992.

[44] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *CONCUR'90: theories of concurrency–unification and extension, Amsterdam, the Netherlands, August 27-30, 1990: proceedings*, page 401. Springer, 1990.

[45] G. Nicolas. A survey of the project automath*. *Studies in Logic and the Foundations of Mathematics*, 133:141–161, 1994.

[46] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[47] P.C. Ölveczky. Real-time Maude 2.3 manual. 2007.

[48] Y. Ortega-Mallen, D. de Frutos-Escrig, S.D.I. y Autoraatlca, and F. de Clencias Matematicas. Timed observations: A semantic model for real-time concurrency 1. In *Programming concepts and methods: proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990.* North Holland, 1990.

[49] J. Ouaknine and S. Schneider. Timed CSP: a retrospective. *Electronic Notes in Theoretical Computer Science*, 162:273–276, 2006.

[50] A.N. Parashkevov and J. Yantchev. ARC-a tool for efficient refinement and equivalence checking for CSP. In *IEEE 2nd International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 1996.

[51] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS*, volume 98, pages 176–185, 1998.

[52] E. Posse. Parsing revisited: A transformation-based approach to parser generation. *PyCon*, 2007.

[53] E. Posse. *Modelling and simulation of dynamic structure discrete event systems*. PhD thesis, School of Computer Science - McGill University, 2008.

[54] E. Posse. Symbolic simulation of π_{klt}. 2010. Unpublished manuscripts.

[55] E. Posse and J. Dingel. Theory and implementation of a real-time extension to the π-calculus. In *Proc. Int. Conf. on Formal Techniques for Distributed Systems (FMOODS&FORTE'10)*, LNCS, 2010. To appear.

[56] E. Posse and H. Vangheluwe. kiltera: a simulation language for timed, dynamic structure systems. In *ANNUAL SIMULATION SYMPOSIUM*, volume 40, page 293. IEEE Computer Society; 1999, 2007.

[57] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578, 1995.

[58] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[59] A.W. Roscoe. Model-checking CSP. In *A classical mind*, pages 353–378. Prentice Hall International (UK) Ltd., 1994.

[60] A.W. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1998.

[61] B. Selic. Using UML for modeling complex real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, pages 250–260. Springer, 1998.

[62] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[63] J. Sun, Y. Liu, and J.S. Dong. Model checking CSP revisited: Introducing a process analysis toolkit. *Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322, 2009.

[64] L.P. Tock. The bcg postscript format. Technical report, INRIA Rhone-Alpes, 1995.

[65] B. Victor and F. Moller. The mobility workbench - a tool for the pi-calculus. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 428–440, London, UK, 1994. Springer-Verlag.

[66] D. Walker. Bisimulation equivalence and divergence in CCS. Technical Report ECS-LFCS-87-29, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1987.

[67] Y. Wang. CCS+ time= an interleaving model for real time systems. In *Proceedings of ICALP*, pages 217–228, 1991.

Appendix A

kiltera model and UML-RT state machines

We present the kiltera model for the third case study shown in Chapter 5. The example state machine is shown again in Figure A.1.

Each kiltera module corresponds to a UML-RT capsule. Within the scope of these modules, there is a process definition: the `PrimaryStateMachine` process definition. The `PrimaryStateMachine` process corresponds to the `Device` capsule's behaviour. The inputs to the statemachine are processed via the `inp` channel. State nesting in UML-RT can be respresented via process nesting in kiltera. For example, consider the `Active` state at the top level. This UML-RT state maps to the kiltera process definition `S_Active` in the `PrimaryStateMachine` process definition. The UML-RT `Active` state contains two substates `Locked` and `Unlocked`, and similarly the kiltera process definition `S_Active` contains two subsequent process definitions `S_Locked` and `S_Unlocked`. The mapping differentiates between composite and basic states, where basic states contain no substates or transitions.

Within the primary state machine and the composite states, transitions are mapped to kiltera processes, prefixed by `Q_`, that represent transition targets. Defined within the primary state machine and in every state are event handler processes. In composite states, this is the process definition `H`. In basic states, the event handler is simply the executed process. The event handler is a listener that

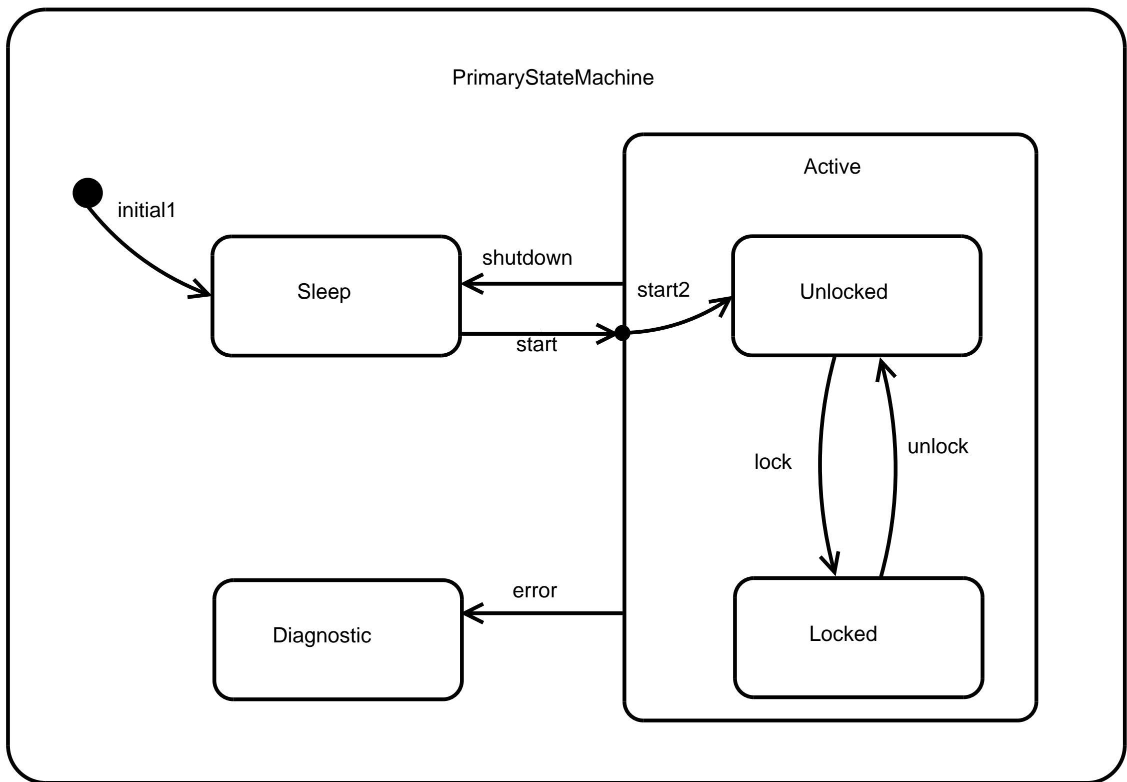


Figure A.1: UML-RT state machine

handles statemachine inputs and internal control signals, executing transition processes as necessary and terminating upon state exit.

Consider the initialization of the capsule and the first two statemachine inputs in the example. At time t=0, the PrimaryStateMachine process will be executed. PrimaryStateMachine's event handler and the top-level initial transition will also be executed. The target of the top-level initial transition is S_Sleep, and so that process will start. S_Sleep is a basic state, which means that it only has to start its event handler. At time t=1.0, the event port_start is sent to the PrimaryStateMachine process via channel inp. This event is matched by PrimaryStateMachine handler's listen process and it is then forwarded to channel inp_in, which is connected to the active substate, S_Sleep. S_Sleep's handler determines that it can act on the event port_start, so it triggers the acc channel to tell the PrimaryStateMachine's handler that it has accepted the input. The PrimaryStateMachine's handler resets itself, and S_Sleep's handler executes the target of the start transition, S_Active. Other transformation rules are similar. We show the kiltera model below. Note that this kiltera model is the result of an automatic transformation that transforms a subset of UML-RT state machines into kiltera programs.

```
1 module DeviceNoHistory[]:
2 process PrimaryStateMachine[inp]:
3   process S_Sleep[inp,acc,rej,exit,exack,sh](entry):
4     process B_exitPoint__dnMGUGI2Ed_a4NaN8NgLUA[sh_in]:
5       par
6         trigger sh.
7         Q_start__dn2NoGI2Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
8     in
9       when
10        inp with x →
11          if x="port_start" then
12            seq
13              trigger acc.
14              Q_start__dn2NoGI2Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
```

```
15          else
16            seq
17              trigger rej with "Sleep".
18              S_Sleep[inp,acc,rej,exit,exack,sh](entry)
19        | exit →
20          trigger exack.
21  process S_Active[inp,acc,rej,exit,exack,sh](entry):
22    process Q_lock__FId4MGI3Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,exit_in,
        exack_in,sh_in]:
23      S_Locked[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
          entryPoint__GpOgUGI3Ed_a4NaN8NgLUA")
24    process Q_unlock__FjckAGI3Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,
        exit_in,exack_in,sh_in]:
25      S_Unlocked[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
          entryPoint__GXTC0GI3Ed_a4NaN8NgLUA")
26    process Q_Initial__uy75QGI5Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,
        exit_in,exack_in,sh_in]:
27 S_Unlocked[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
    DEFAULTENTRY_Unlocked__2OpkQGI2Ed_a4NaN8NgLUA")
28    process B_exitPoint__hw670GI2Ed_a4NaN8NgLUA[sh_in]:
29      par
30        trigger sh.
31        Q_error__hxJlUGI2Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
32    process B_exitPoint__kyPXwGI2Ed_a4NaN8NgLUA[sh_in]:
33      par
34        trigger sh.
35        Q_shutdown__kylWAGI2Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
36    process S_Unlocked[inp,acc,rej,exit,exack,sh](entry):
37      process B_exitPoint__FIQc0GI3Ed_a4NaN8NgLUA[sh_in]:
```

```
38        par
39          trigger sh.
40          Q_lock__FId4MGI3Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
41      in
42        when
43          inp with x →
44            if x="port_lock" then
45              seq
46                trigger acc.
47                Q_lock__FId4MGI3Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh
                    ]
48            else
49              seq
50                trigger rej with "Unlocked".
51                S_Unlocked[inp,acc,rej,exit,exack,sh](entry)
52        | exit →
53            trigger exack.
54    process S_Locked[inp,acc,rej,exit,exack,sh](entry):
55      process B_exitPoint__FjPIoGI3Ed_a4NaN8NgLUA[sh_in]:
56        par
57          trigger sh.
58          Q_unlock__FjckAGI3Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,sh]
59      in
60        when
61          inp with x →
62            if x="port_unlock" then
63              seq
64                trigger acc.
```

```
65                 Q_unlock__FjckAGI3Ed_a4NaN8NgLUA[inp,acc,rej,exit,exack,
                     sh]
66             else
67               seq
68                 trigger rej with "Locked".
69                 S_Locked[inp,acc,rej,exit,exack,sh](entry)
70         | exit →
71             trigger exack.
72     process H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]:
73       when
74         inp with x →
75           seq
76             trigger inp_in with x.
77             when
78               acc_in →
79                 seq
80                   trigger acc.
81                   H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
82             | rej_in with context →
83                 if x="port_shutdown" then
84                   seq
85                     trigger exit_in.
86                     when
87                       exack_in →
88                         seq
89                           trigger acc.
90                           Q_shutdown__kylWAGI2Ed_a4NaN8NgLUA[inp,acc,rej
                               ,exit,exack,sh]
91                 else
```

```
92                  seq
93                    trigger rej with ("Active",context).
94                    H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
95        | exit →
96            seq
97              trigger exit_in.
98              when
99                exack_in →
100                 trigger exack.
101       | sh →
102           done
103     in
104       event inp_in,acc_in,rej_in,exit_in,exack_in,sh_in in
105         par
106           if entry="entryPoint__qnUUwGI2Ed_a4NaN8NgLUA" then
107             Q_Initial__uy75QGI5Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,
                    exit_in,exack_in,sh_in]
108           else
109             done
110           H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
111   process S_Diagnostic[inp,acc,rej,exit,exack,sh](entry):
112       when
113         inp with x →
114           seq
115             trigger rej with "Diagnostic".
116             S_Diagnostic[inp,acc,rej,exit,exack,sh](entry)
117       | exit →
118           trigger exack.
```

```
119   process Q_Initial__OK27MGI2Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,exit_in
        ,exack_in,sh_in]:
120 S_Sleep[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
      DEFAULTENTRY_Sleep__OK2UIGI2Ed_a4NaN8NgLUA")
121   process Q_start__dn2NoGI2Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,exit_in,
        exack_in,sh_in]:
122     S_Active[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
          entryPoint__qnUUwGI2Ed_a4NaN8NgLUA")
123   process Q_error__hxJlUGI2Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,exit_in,
        exack_in,sh_in]:
124     S_Diagnostic[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
          entryPoint__xz_94GI2Ed_a4NaN8NgLUA")
125   process Q_shutdown__kylWAGI2Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,
        exit_in,exack_in,sh_in]:
126     S_Sleep[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]("
          entryPoint__o5YzUGI2Ed_a4NaN8NgLUA")
127   process H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]:
128     when
129       inp with x →
130         seq
131           trigger inp_in with x.
132           when
133             acc_in →
134               H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
135           | rej_in with context →
136               par
137                 trigger dropped with context.
138                 H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
139   in
```

```
140     event inp_in,acc_in,rej_in,exit_in,exack_in,sh_in in
141       par
142         H[inp_in,acc_in,rej_in,exit_in,exack_in,sh_in]
143         Q_Initial__OK27MGI2Ed_a4NaN8NgLUA[inp_in,acc_in,rej_in,exit_in,
              exack_in,sh_in]
144 in
145   event inp in
146     par
147       PrimaryStateMachine[inp]
148       schedule inp with "port_start" after 1.0.
149       schedule inp with "port_unlock" after 2.0.
150       schedule inp with "port_shutdown" after 3.0.
151       schedule inp with "port_error" after 4.0.
```