VERIFICATION AND IMPLEMENTATION OF EMBEDDED SYSTEMS FROM HIGH-LEVEL MODELS
VERIFICATION AND IMPLEMENTATION OF EMBEDDED SYSTEMS FROM HIGH-LEVEL MODELS

By
BOJAN NOKOVIC, M.Eng., B.Sc.

A Thesis
Submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

McMaster University © Copyright by Bojan Nokovic, March 2016
Verification and Implementation of Embedded Systems from High-Level Models

Bojan Nokovic
M.Eng., (Engineering Entrepreneurship and Innovation)
McMaster University, Hamilton, Canada
B.Sc., (Telecommunications)
University of Sarajevo — Faculty of Electrical Engineering, Sarajevo, Bosnia and Herzegovina

Dr. Emil Sekerinski

xvii, 164
Abstract

Existing modelling tools for embedded systems analyze models that do not necessarily reflect executable code. We propose a holistic approach that includes visual modelling of the system and its environment, qualitative and quantitative verification of the model, and automated executable code generation.

For high-level modelling, we introduce pCharts, a variant of hierarchical state machines with state invariants, probabilistic transitions, timed transitions, stochastic transitions, state costs, and transition costs. With embedded systems in mind, pCharts follow an event-centric interpretation, in which events are executable procedures, implying that their execution is fast enough that no queuing of events is needed.

Our pCharts tool, pState, allows the whole system to be formally analyzed and code for executable parts generated. The goal is an automated approach from modelling and analysis to code generation. On a series of case studies, we demonstrate this technique.

An off-the-shelf probabilistic model checker is used for analysis by compiling and transforming a hierarchical pCharts into a flat collection of probabilistic guarded commands with costs. From the pCharts model, also executable code that preserves the verified properties can be generated.
Acknowledgements

First, I am gratefully thankful to my supervisor, Dr. Emil Sekerinski, for his careful guidance throughout my study. Without his support and dedication, this work could not have been possible. On Dr. Sekerinski’s recommendation, I was awarded the Queen Elizabeth II Graduate Scholarships in Science and Technology that helped me to finalize my research work. He funded my visit to many international conferences (Williamsburg, Montreal, Grenoble, Edinburg, Budva, Rome, Lund, and Oslo) where I had an opportunity to present our work and to learn about the latest achievements in the area of software engineering.

Next, I would like to express great appreciation to my supervisory committee members, Dr. Douglas Down and Dr. Mark Lawford, for excellent comments and suggestions on supervisory committee meetings and for a careful review of the thesis, and to the internal examiner, Dr. Alan Wassyng, for insightful comments on this thesis. I am tremendously grateful to my external examiner Dr. Juergen Dingel for his positive review of this thesis and indication of the parts that required further explanation.

Last but not least, special thanks to my fellow graduate students Ronald Eden Burton, Shucui Yao, and former student Dr. Tian Zhang for their constructive feedback in our group meetings.
Contents

Abstract iii

Acknowledgements iv

Declaration of Academic Achievement xiii

1 Introduction 1
  1.1 System Verification ........................................ 2
  1.2 Holistic Approach ........................................ 6
  1.3 Motivational Example .................................... 7
  1.4 Contribution of this Work ............................... 11
  1.5 Structure of the Thesis .................................. 13

2 pCharts, pState, and Related Work 14
  2.1 Hierarchical State Machines ............................. 15
  2.2 Probabilistic Model Checking ............................. 19
  2.3 Code Generation ......................................... 21
  2.4 Model-based WCET Analysis ............................... 22
  2.5 Documentation ........................................... 24
  2.6 Summary .................................................. 25
3 Overview of the Translation Scheme of pCharts

3.1 Probabilistic Transitions ......................................... 26
3.2 Timed Transitions ................................................. 28
3.3 Basic Charts Operation ............................................ 29
3.4 Exponential Timed Transition ..................................... 31
3.5 Uniform Timed Transition ......................................... 33
3.6 Hierarchy ............................................................. 36
3.7 Concurrency .......................................................... 37
3.8 Communication ....................................................... 38

4 Formal Definition and Normalized pCharts ..................... 40

4.1 Chart Components ................................................. 40
4.2 Translation to Flat Guarded Commands .......................... 50

5 Model-based WCET Analysis with Invariants ................. 64

5.1 Introduction .......................................................... 64
5.2 WCET Calculation Principles ..................................... 68
5.3 Transition with Specified Deadline .............................. 70
5.4 WCET Taking State Invariants into Account .................... 74
5.5 WCET Taking into Account Timed Transitions ................ 75
5.6 WCET of Simple Timed Transitions .............................. 77
5.7 Modelling Probabilistic Timing Analysis ......................... 80
5.8 Probabilistic WCET ................................................. 84
5.9 Summary .............................................................. 85

6 Code Generation .................................................... 88

6.1 Transformations .................................................... 88
List of Tables

3.1 Transition $unif(3s,6s)$ probability ........................................ 36
5.2 Estimated vs. actual run time .................................................. 79
5.3 Probabilities calculated by $pState$ ........................................... 82
5.4 Costs calculated by $pState$ ..................................................... 82
5.5 $ETP_{x\otimes y}$ calculated by $pState$ ......................................... 83
5.6 Expected time to reach $S_4$ ..................................................... 85
6.7 Micro-controller PIC16F636 pins connection ................................ 96
8.8 Cost vs. channel quality ......................................................... 113
8.9 Min/max expected cost vs. channel quality .................................. 116
9.10 Micro-controller PIC16F636 test pins connection ......................... 137
## List of Figures

1.1 Conventional software design process ........................................... 2
1.2 Software design process with model checking ................................. 3
1.3 The pCharts model checking design process .................................... 5
1.4 Top-level pState architecture ..................................................... 8
1.5 Sender-receiver probabilistic transmission ..................................... 9
1.6 Interrogator-tag communication timing diagram .............................. 11
3.1 pCharts transition .............................................................. 27
3.2 State label structure ........................................................... 27
3.3 State representation ............................................................ 29
3.4 Event transition ................................................................. 30
3.5 Several transitions on the same event ......................................... 30
3.6 Probabilistic transition .......................................................... 30
3.7 Timed transition ................................................................. 31
3.8 Exponential distribution delay .................................................. 31
3.9 Exponential delay ............................................................... 32
3.10 Uniform distribution delay ..................................................... 34
3.11 Uniform delay ................................................................. 34
3.12 Hierarchical states .............................................................. 36
3.13 Entering superstate ............................................................ 36
3.14 Priority transition ................................................. 37
3.15 State entered by fork ............................................. 37
3.16 Parallel composition ............................................. 38
3.17 Broadcast event .................................................. 39
4.1 Inter-level transition ............................................. 43
4.2 Examples for transition completion ............................. 44
4.3 Examples of max function and fold iterator .................... 46
4.4 Transitions on event $E$ ........................................... 54
4.5 Timed transitions .................................................. 59
4.6 Broadcast on probabilistic transition ............................ 61
4.7 Parallel probabilistic transition .................................. 62
5.1 Multiple transitions on the $tick$ event .......................... 65
5.2 Transition on event $E$ with guard, body, and deadline ....... 71
5.3 Transition with parallel composition in body ................... 73
5.4 WCET state invariant annotation .................................. 75
5.5 Transition on $E$ disabling and enabling timed transitions ... 75
5.6 Timed transition $t$ scheduled 100ms after $S$ is entered .... 77
5.7 Convolution $X \otimes Y$ in pCharts ................................. 81
5.8 Example of SPTA in pCharts ....................................... 82
5.9 Average cycles calculation ........................................ 84
5.10 Probabilistic transition .......................................... 84
6.1 Operation on event $E$ ............................................. 89
6.2 Structure of C code for PIC 16F6xx ............................... 90
6.3 Simple operation on event $E$ .................................... 93
6.4 pCharts model of Key-fob ........................................ 96
6.5 Structure of target code .......................................... 99
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Statecharts with invariants for TV set</td>
<td>104</td>
</tr>
<tr>
<td>7.2 Class diagram of the visitor pattern in \textit{pState}</td>
<td>105</td>
</tr>
<tr>
<td>8.1 Fischer's protocol in \textit{pCharts}</td>
<td>108</td>
</tr>
<tr>
<td>8.2 Alternating bit protocol in \textit{pCharts}</td>
<td>111</td>
</tr>
<tr>
<td>8.3 \textit{pCharts} model of RFID tag excitation in \textit{pCharts}</td>
<td>115</td>
</tr>
<tr>
<td>8.4 \textit{pCharts} model of the Hubble space telescope</td>
<td>118</td>
</tr>
<tr>
<td>8.5 Wireless sensor power model in \textit{pCharts}</td>
<td>120</td>
</tr>
<tr>
<td>9.1 Collision model, three tags, five time slots</td>
<td>127</td>
</tr>
<tr>
<td>9.2 Collision probability for tags $N = [2, 3]$ and time slots $M = [3..9]$</td>
<td>128</td>
</tr>
<tr>
<td>9.3 Power consumption of collection period</td>
<td>129</td>
</tr>
<tr>
<td>9.4 Tag model</td>
<td>131</td>
</tr>
<tr>
<td>9.5 Testing generated executable code framework</td>
<td>138</td>
</tr>
<tr>
<td>A.1 State-transition diagram of the MDP model</td>
<td>146</td>
</tr>
<tr>
<td>A.2 State-transition diagram of the PTA model</td>
<td>149</td>
</tr>
</tbody>
</table>
Declaration of Academic Achievement


Nokovic, B. and E. Sekerinski (2015). Analysis and implementation of embedded system models: Example of tags in item management application. Workshop on Model-Implementation Fidelity (MiFi), Grenoble, France. (to appear as a


Chapter 1

Introduction

The design and analysis of complex embedded systems is of practical interest in many application domains including telecommunications, consumer electronics, automotive, and avionics industry. In a complex embedded system, not only functional correctness, liveness and timing guarantees are relevant, but also quantitative properties like resource consumption (e.g. power), reliability (e.g. lost messages, life expectancy), and performance (e.g. throughput). These properties cannot be analyzed by considering only the software part; rather, the environment has to be considered as well. Furthermore, quantitative properties cannot be analyzed in isolation. For example, the power consumption of a wireless transmitter depends on the algorithms implemented in the sender and receiver. The analysis of the correctness of the implementation and its power consumption is intertwined.

The original motivation for this work came from the design of radio-frequency identification (RFID) tags for postal systems [Nokovic and Sekerinski 2010, 2011]. Thus, while we model complex embedded systems, the code is executed on microcontrollers with limited resources, like the ones used in active RFID tags.
1.1 System Verification

The goal of system verification is to establish if the design or product under consideration has certain properties. The basis for verification is a specification that prescribes what the system has to do. If the system satisfies all specified properties it is considered correct, otherwise it is defective. Correctness is always relative to the specification; it is not an absolute property of a system. When we say that a system is correct, it is considered to be correct with respect to a specification.

Problems Related to Conventional Software Design Process. Some of the standard software verification techniques are peer review (inspection) and testing. Peer review is usually done by software engineers who were not involved in the development. Although this technique is common and used in almost 80% of all software projects, on average only about 60% of defects are caught [Boehm and Basili, 2001].

It is difficult to detect some errors, such as errors in concurrent processes or algorithm defects. The correctness of software systems in the conventional software development process, shown in Figure 1.1, is relative to the specification and consequently executed test cases. This process can discover errors but cannot guarantee correctness: an error may still exist in the product. It was observed decades ago that program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence [Dijkstra, 1972].

The other problem is that errors are discovered too late, when the product is already built. The repair cost of software bugs is directly related to the phase of

![Figure 1.1: Conventional software design process](image-url)
the software development cycle in which errors are detected. Bugs detected during the *design* and *programming* phases are relatively inexpensive to fix, while the bugs detected during *unit* and *system* testing are much more expensive. It is of importance to locate software bugs early in the development cycle, the sooner the better [Peled et al., 2001].

**Model Checking Software Design Process.** To overcome these two problems, the technique of *model checking* was proposed [Clarke and Emerson, 1982]. It is an automated method for verifying finite state systems, a process which for a given program $M$ and specification $h$, determines whether or not the behaviour of $M$ satisfies $h$ [Emerson, 2008].

Model checking performs an exhaustive graph search to find out whether or not the behaviour described by a temporal property holds of the system’s state graph [Emerson, 2008]. To perform model checking on software, we need to translate the code into a mathematical model and then verify properties over the model. The result of model checking is *true* when the model satisfies a property formula or *false* when that is not a case. From a model that describes the system behaviour in a mathematically precise manner, a simplified model suitable for model checking is generated. Model checking allows one to explore all possible system states in a systematic manner. In Figure 1.2, the software design process with model checking is shown. On the system model, the *property* in the form of a temporal logic formula has to be specified. Verification determines if the specified properties are satisfied or not.

![Figure 1.2: Software design process with model checking](image-url)
Embedded Systems Modelling. An embedded system is any system that interacts extensively with its environment. The software development process for embedded systems should describe the external environment of the system and the interaction with its environment. In embedded systems, the impact of the working environment and the reaction to external stimuli determine the correctness of a system. To verify those properties, a model of the environment has to be created together with a model of the system. If errors are detected by model checking, either the system model or the environment model need to be modified.

The Embedded Computer Systems Analysis and Modelling method (ECSAM) \cite{lavi2005} shows that hierarchical state machines are suitable for describing an environment, a system, and their interactions. The weakness of this modelling method and associated tools like STATEMATE \cite{harel1996, werner1998, bode2006} is that the syntax that does not allow the specification of probabilistic processes.

With a probabilistic model, it is possible to analyze a wide range of quantitative properties. For example, we can find out "What is the probability of a failure causing the system to shut down within 2 hours?", "What is the worst-case probability of the protocol terminating in error, over all possible initial configurations?", or "What is the worst-case expected time taken for the algorithm to terminate?" \cite{kwiatkowska2007}.

We propose pCharts, an extended hierarchical states modelling formalism, by which we can specify probabilistic processes in both a system and its environment. Qualitative and quantitative properties are expressed directly in the model. Verification of qualitative properties returns $true$ if the property is satisfied, otherwise $false$. Quantitative property verification returns a numerical value. In Figures 1.2 and 1.3, the qualitative property $in\, Q \Rightarrow \neg in\, T$ states that whenever the system is in state $Q$,
it should not be in state $T$. The quantitative query $\$power.max$ returns the maximum value of $power$ spent by the time the state is reached.

By probabilistic model checking we can quantify the amount of certainty for a formula to be satisfied. The result is a real number in range $[0,1]$. An impossible outcome has probability 0; a certain outcome has probability 1.

Our approach is to create models for the system and its environment, and then to automatically generate both the structure on which model checking can be performed and executable code to run on some embedded system. While the executable code can be generated only from a sub-model with deterministic transitions, the mathematical structure can be generated on the whole system to examine probabilistic transitions.

We introduce an algorithm to generate both input code for a model checker and executable code. The algorithm is implemented in our graphical programming tool $pState$.

Following the basic idea that modelling is a process of gaining a deeper understanding of a system through imitation [Lee and Seshia, 2015], using $pState$ we can specify what the system does, then from the specified structure we can generate executable code which determines how the system works, and finally by generating input code for the model checker we reason why the system does what it does, or fails to do what it is specified to do by the model.
1.2 Holistic Approach

We propose a holistic methodology with a visual formalism for the design of probabilistic reactive systems and the generation of a model as a Markov decision process (MDP) or probabilistic timed automata (PTA). Those mathematical structures are used as input for the PRISM probabilistic model checker [Hinton et al., 2006]. On a probabilistic structure, we can formally analyze systems which exhibit random or probabilistic behaviour. Properties in PRISM are specified in probabilistic computational tree logic PCTL [Aziz et al., 1995; Hinton et al., 2006], which is a probabilistic extension of the temporal logic CTL [Clarke and Emerson, 1982]. On pChart models, temporal properties are specified in an intuitive way such that they can be written by engineers who are experts in their domain, without necessarily needing to be familiar with temporal logic. The property specification language incorporates temporal logic.

Existing automated tools for analyzing discrete, timed, probabilistic and stochastic models have a textual user interface, which makes them less suitable for engineers developing large systems. The implementation of a graphical user interface in the form of extended hierarchical state machines allows a view, where the whole system is represented from the perspective of related states. The automatic generation of the input code and properties for the probabilistic model checker makes model checking more widely usable in software engineering process [Jansen, 2003].

Through the pState editor, the system model, safety properties, and quantitative queries are entered. Systems are verified over the model of MDP or PTA. After validation, executable code for the software part of the system can be generated, safety or liveness properties can be verified, and quantitative properties can be analyzed. In addition to this, we show how worst-case execution time (WCET) with verified invariants can be calculated directly from the model, and how an execution time
profile (ETP) can be built for probabilistic transitions. The correctness of transitions with respect to state invariants can be checked with a satisfiability modulo theories (SMT) solver. The generated code contains comments placed on the pCharts model. The holistic method of \textit{pState} is shown in Figure 1.4.

State of the art commercial modelling tools like RHAPSODY [Harel and Kugler, 2004], SCADE Suite [Esterel, 2015], Stateflow with Simulink [Mathworks, 2011], Papyrus [Eclipse, 2011b], Yakindu [Yakindu, 2013], and IAR visualSTATE [IARSystems, 1999] are capable of modelling both the system and its environment. They can verify the design, generate code, and perform executable analysis. The main differences between our tool and those commercial tools is in (1) the ability of \textit{pState} to represent the model by extended hierarchical state machines (extended by invariants, probabilistic transitions, and costs), that allows quantitative properties and probabilistic execution time profile to be analyzed, (2) \textit{pState} generates event-centric code that reduces the state space of probabilistic models and make executable code simple and efficient. This is in contrast to the state-centric interpretation of other tools, where events are data in queues. The limitation is that this works only for a small embedded systems with a limited number of transitions, like active RFID tags or wireless sensors, where no queueing of events is needed. \textit{Overall our thrust is to have a tool that is founded in solid theory and intuitive enough to be used by engineers for analyzing design tradeoffs and for generation of executable code for small, but practical systems.}

1.3 Motivational Example

Figure 1.5 illustrates the elements of pCharts. This setup is typical for networks of sensors, in particular RFID tags [Nokovic and Sekerinski, 2010]. The state \textit{System} is an AND (concurrent) state with children \textit{Sender} and \textit{Receiver}, separated by the
dashed line. Both Sender and Receiver are XOR states with Basic states as children. The sender is initially in state Sleep and the receiver in state Listening. The sender exits the sleep mode on wake-up event \textit{wup}. For postal RFID tags \cite{Paun2006}, that event can be created either by a low frequency electromagnetic field, by a motion sensor, or by an internal timer. When this event is generated by a motion sensor or internal timer, the sender always goes into transmission mode. On the other hand, an electromagnetic field can be created by system antennas (\textit{good field}), or by other sources like power lines, monitors, cell phones, or electrical machines (\textit{parasite field}). A good field has a unique identification number. If the sender recognizes the field identification number, it goes into transmission mode; otherwise it goes back into sleep mode. This is expressed by the probabilistic transition that Sender with probability 0.6 goes to state Sending and with probability 0.4 goes back to state Sleeping. A sent message may reach the receiver or may get lost. This is expressed by another probabilistic transition that with probability 0.9 broadcasts \textit{msg} to the receiver, which causes the receiver to go from Listening to Off. The receiver then shuts off to save power, while the sender (with unidirectional transmission) keeps retransmitting the
Figure 1.5: Sender-receiver probabilistic transmission

message. We like to analyze the following properties of the system:

• Is the system correct in the sense that the receiver is attentive when needed? We express this by attaching the invariant \( \text{in Sleeping} \Rightarrow \neg \text{in Off} \), at the state \( \text{System}; \ pState \) reports true.

• What is the minimal probability that the receiver shuts off? We express this by attaching the query \(?P.min\) to state \( \text{Off}; \ pState \) reports 1.0.

• What is the maximal number of expected message transmissions of the sender until the receiver shuts off? For this, we attach a cost of \( \$t = 1 \) to the sending transition and can ask what is the maximal expected value of \( t \) upon entering state \( \text{Off} \) by attaching the query \(?t.max\) to \( \text{Off}; \ pState \) reports 1.11.

• What is the maximal expected energy consumption until the message reaches the receiver? For this, we attach the cost of \( \$energy = 0.1 \) to state \( \text{Sleep} \) and
\$energy = 2\$ to \textit{Sending}. Now, we can ask what the maximal expected value of \textit{energy} is in the state \textit{Off} by attaching the query \$\textit{energy}.\max\$ to \textit{Off}; \textit{pState} reports 2.39.

- Is the probability that the receiver shuts off at least 0.5? We express this by attaching the query \$P > 0.5\$ to \textit{Off}; \textit{pState} reports \textit{true}.

When an RFID tag is used for \textit{item management}, master-slave communication between an interrogator (reader) and the tag has to be established. We assume that the communication is according to the ISO/IEC 18000-7.2\cite{DASH7-Alliance} protocol. The tag collection process is an iterative process that includes methods for coordinating responses from the tag population and handling collisions which occur when multiple tags transmit at the same time. The entire tag collection process is referred to as a \textit{Complete Collection Sequence}. Figure\ref{fig:1.6} shows a complete collection sequence consisting of a \textit{wakeup period} (WP) followed by a series of \textit{collection periods} (CP). Each collection period consists of a \textit{synchronization period} (SP), a \textit{listen period} (LP), and an \textit{acknowledge period} (AP). The LP is further divided into multiple time slots (TS). We assume that we have \textit{n} tags and \textit{m} time slots. When a collision happens, as shown for tags \#1 and \#\textit{n}, they have to retransmit in the next CP. To analyze the power consumption of a tag device, the collision probability has to be taken into account. To fully analyze a system in which tags are working according to this protocol, we need to determine (1) how the collision probability depends on the number \textit{m} of time slots and the number \textit{n} of tags in RF reading range, (2) the average tag power consumption taking into account collision probability. The power consumption is related to the number of retransmissions and a retransmission happens only in the case of a collision. Without quantification of the collision probability from the model of the environment, we can not accurately determine the average power.
consumption of the device.

![Interrogator-tag communication timing diagram](image)

Figure 1.6: Interrogator-tag communication timing diagram

### 1.4 Contribution of this Work

*pState* is intended to be capable of assisting system designers during embedded system code generation. The tool can be used (1) for system property analysis like collision probability of the protocol shown in Figure 1.6, (2) for device property analysis (e.g. power consumption), and (3) to generate code for the device. We propose a holistic process in which validation is done by qualitative and quantitative model checking and executable code is generated after validation is completed. The main contributions of this thesis are:

- An event-centric interpretation of pCharts through probabilistic guarded commands [Nokovic and Sekerinski, 2014]. The translation is simple and intuitive enough to serve as the definition of pCharts. The definition supports state hierarchies with inter-level transitions and concurrent states with broadcasting in arbitrary combinations.
• For the purpose of probabilistic model checking, a flat structure of guarded commands is automatically created from the hierarchical pCharts representation.

• From an intermediate representation of the nested control structures, generation of C code and assembly code with some restrictions for a RISC 8-bit microcontrollers is supported.

• We propose an extension of hierarchical charts to allow the specification of a cost (or reward) of being in a state and of taking a transition. In pState, the cost specifications are validated and then translated as annotations of the generated probabilistic guarded commands. A theory for costs in that form is given by priced probabilistic automata.

• We allow the specification of a temporal formula directly on the state, taking into account the hierarchical structure of the chart [Nokovic and Sekerinski, 2015b].

• States and transitions documentation is generated from overlaid comment boxes and passed to the generated code to allow traceability, e.g. for the certification of the generated code [Nokovic and Sekerinski, 2015b].

• We propose model-based WCET analysis with invariants and probabilistic WCET analysis [Nokovic and Sekerinski, 2015c].

• The effectiveness of each contribution is demonstrated on a number of case studies discussed in the thesis [Nokovic and Sekerinski, 2013, 2014, 2016, 2015b,a].
1.5 Structure of the Thesis

The thesis is organized as follows:

Chapter 2 discusses in detail related work and gives the basic concepts of hierarchical state machines and model checkers;

Chapter 3 gives an overview of the translation scheme of probabilistic and timed transitions;

Chapter 4 gives the formal definition of pCharts;

Chapter 5 presents model-based WCET analysis with invariants and model-based probabilistic WCET;

Chapter 6 describes transformations of chart transitions into an intermediate code representation and executable code generation for different targets;

Chapter 7 describes the modular design and implementation of \textit{pState};

Chapter 8 presents a number of case studies to illustrate different aspects of automatic quantitative analysis and executable code generation;

Chapter 9 shows how an open RFID standard can be analyzed by \textit{pState};

Chapter 10 concludes the thesis with a discussion of future work;

Appendix A gives the formal definition of MDP models, PTA models, and PCTL.
Chapter 2

pCharts, pState, and Related Work

The probabilistic model checker PRISM [Kwiatkowska et al., 2011] is a tool for modelling and analysis of systems that exhibit random or probabilistic behaviour. PRISM describes the behaviour of systems by a set of commands; it cannot generate executable code. A translation of visual models from extended UML state diagrams [Jansen, 2003] to PRISM is proposed by [Zhao et al., 2010]. Nested concurrency representation is not supported and generation of executable code is not possible. A number of UML modelling tools such as Papyrus [Eclipse, 2011b], MADES [Baresi et al., 2015], StarUML [MKLab, 2016], STATEMATE [Bode et al., 2006], Open ModelSphere [Grandite, 2016], Yakindu [Yakindu, 2013] do not allow verification conditions to be stated directly in a model and probabilistic behaviour cannot be specified. To overcome those limitations we propose a new formalism, pCharts, and a new tool, pState.

pCharts are based on the classical visual formalism of hierarchical state machines. Statecharts [Harel and Naamad, 1996] are extended by probabilistic transitions [Jansen, 2003; Nokovic and Sekerinski, 2013], timed transitions [Jansen, 2003; Nokovic and Sekerinski, 2014], state invariants [Back, 2009; Sekerinski, 2009], state
and transition cost [Nokovic and Sekerinski, 2014] and quantitative queries [Nokovic and Sekerinski, 2015b].

The main pieces of our holistic approach are extended hierarchical state machines, probabilistic model checking, executable code generation, executable time analysis, and design documentation.

2.1 Hierarchical State Machines

State machines are a formalism used to model real-time systems that can be described as a collection of discrete states. The machine goes from one state to another when stimuli or events are received. The first limitation of conventional Mealy-Moore [Moore, 1956; Mealy, 1955] state machines is related to a large number of states, specially for a complex system. All states are at one hierarchical level, and there is no communication between them. Another limitation is a lack of support for concurrent transitions. In the conventional state machine model, states form an OR state; the machine is either in one state or another, it can not be in two or more states concurrently. The number of states grows exponentially with the number of parallel components; that makes this representation difficult to use for complex systems. On the other side, hierarchical state machines, statecharts [Harel, 1987], overcome limitations of flat state machines by providing a construct known as an AND state. An AND state allows the statechart to have children or substates of a higher level state to be active at the same time. All concurrent states of the chart accept broadcasted events. Relationships between concurrent states can be formed through synchronization techniques and decomposition of states. Compared to classical state machines, statecharts are more expressive. We can think of hierarchical state machines just as syntactic sugar, compact representation of a flat state machine.
Complex real-time systems can be specified in an intuitive comprehensible graphical manner with fewer states. Since 1987, when the paper [Harel, 1987] that introduced the language of statecharts was published, more than 100 variants of statecharts have been introduced. Variants of statecharts semantics are mostly with respect to superstep, transition priorities, and history [von der Beeck, 1994]. Statecharts respond to an event by engaging in an enabled transition, which can generate new events. Effects of the step can be calculated in the same step, or take effect only in the following one. Another difference in statecharts is in how nondeterminism is handled when more than one transition is enabled. Classical statecharts give priority to the outer transition while UML statecharts give priority to the inner transition [OMG, 2009].

The main differences between UML semantic and classical semantics, are described in [Eshuis and Wieringa, 2000]. The behaviour of a system described by statecharts is a set of possible runs [Harel and Naamad, 1996] which are formally specified by state configurations or statuses and executing steps.

**pCharts, an Extended Hierarchical State Machine.** To be able to specify and verify probabilistic systems we designed pCharts, extended hierarchical state machines, which borrow hierarchical states, concurrent states, and broadcasting from statecharts and adds state invariants, probabilistic transitions, and costs (or rewards) attached to states and to transitions.

State invariants are safety conditions that can be attached to any state in a state hierarchy and specify what conditions must hold in that state. Every incoming transition to a state must ensure that the state invariant holds, and every outgoing transition can assume that the invariant holds. State invariants can express the safety of an embedded system or consistency of a software system. The accumulated invariant of a state consists of a conjunction of invariants “inherited” from ancestor states and
a combination of invariants of descendant states. We have implemented accumulated
invariants in $pState$ following the definition and algorithm of [Sekerinski, 2009]. For
using a model checker for invariant verification, we interpret invariants as temporal
always conditions rather than as inductive invariants [Sekerinski, 2008].

Probabilistic transitions can be used to express randomized algorithms or quantify
the uncertainty of the environment. Probabilistic descriptions are useful for analyzing
the quality of service, response time, unreliable environments, and fault-tolerant
systems.

Costs or rewards are quantitative information associated with a transition or state.
Models with costs represent *priced probabilistic timed automata* and can be used to
reason about properties like (1) minimum/maximum expected time before some trans-
ition will take place, or (2) expected steps to reach a particular state.

Several interpretations of events in statecharts have been proposed. With embed-
ded systems in mind, pCharts follow an *event-centric* interpretation, in which events
are executable procedures, implying that their execution is fast enough that no queu-
ing of events are needed [Sekerinski and Zurob, 2001]. For example, if an event leads
to broadcasting of another event, the second one is executed in parallel with the first
one. Syntactic constraints impose that there are no race conditions during paral-
lel execution and hence the effect of an external event is a single atomic statement.
This makes the structure of the generated code simple and the code efficient. This
is in contrast to the *state-centric* interpretation in UML and STATEMATE [Harel and
Naamad, 1996], in which events are data in queues. These interpretations are called
requirements-oriented and implementation-oriented semantics in [Eshuis et al., 2002],
with our event-centric interpretation being the requirements-oriented semantics. As
shown in the development of *iState* [Sekerinski and Zurob, 2001], the event-centric
approach is suitable for reactive systems where events are processed quickly enough
so that no queuing of events is necessary and where blocking of events is undesirable. This semantic is close to [Mikk et al., 1998], but we do not support spontaneous transitions—transitions without an event. In the response to an external event, a system may broadcast additional events. The execution step completes as soon as a chain of reactions comes to a halt [Lütten and Mendler, 2002]. A number of quantitative extensions of statecharts similar to pCharts have been proposed, all based on UML state machines [Jansen et al., 2002; Jansen, 2003; Zhao et al., 2010; Leitner-Fischer and Leue, 2011] (the corresponding tools all use PRISM). These follow the state-centric interpretation in which the state of a chart is given by the configuration (the “states of the current state”), a set of events, and the valuation of the variables (e.g. p. 67 in [Jansen, 2003]). In pCharts, the state consists only of the configuration and the valuation of the variables, thus reducing the state space and facilitating model checking.

pCharts represent a synchronous reactive system. The system is not preemptive. Transitions of the top level state machine have priority over transitions in the refinement state machine. Reset always initializes the refinement of the destination state to its initial state, so history transitions are not supported. This allows us to create a more compact flat state machine from the hierarchical representation.

Another difference between UML State Machine Diagrams [Fowler, 2003] and pCharts is related to internal state activities specification. UML allows two types of internal state activities ”do-activities”, and ”regular activities”, specified by entry and exit keywords. Regular activities occurs ”instantaneously” while do-activities can take finite time and can be interrupted. pCharts not allow the explicit specification of internal state activities, but the entry part of the activity can be specified in the body of the incoming transitions, and the exit part can be specified in the body of the outgoing transitions. Execution of the body of the transitions is instantaneous, as in
UML, and cannot be interrupted by another event. The UML Profile for Modelling quality of service (QoS) provides the user with facilities to define a wide variety of QoS requirements and properties \cite{Kumar2010}, divided into categories: performance, dependability, security, integrity, coherence, throughput, latency, efficiency, demand, reliability, and availability. In pCharts, a transition deadline can be specified directly on the model. A specified deadline is similar to the QoS latency property, since both refer to a time interval during which a response to an event must be completed. The behavioural diagrams used in the OMG systems modelling language SysML consists of activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams. State machine diagrams in SysML are based on the standard UML state machine concept, so do not support state invariants. The UML profile for real-time systems (UML-RT) extends the basic UML concepts to facilitate the design of complex real-time systems \cite{Kumar2010}. UML-RT is an industrial standard and can be used to design event-driven real-time systems. The standard is primarily focused on architecture specification of real-time systems, but the behaviour can be modelled by state machine diagrams. UML-RT state machines do not allow concurrent states.

### 2.2 Probabilistic Model Checking

A set of interacting or interdependent components that exhibit probabilistic aspects is called a *probabilistic system* \cite{Baier2008}. Probabilistic aspects are essential in randomized algorithms, modelling unreliable and unpredictable system behaviour, model-based performance evaluation, quality of service, workloads on the system imposed from environment. The automated verification of probabilistic systems is
a technique for establishing if certain properties hold for a model of a finite state concurrent system. Properties are expressed in temporal logic extended with probabilistic and reward operators. The model is typically specified in a high-level modelling language as a variant of Markov chains annotated with costs or rewards. Properties of the system model, like whether the executions violate safety or the program eventually terminates, are called qualitative properties. Since various systems have random assignments or delays, we need quantitative analysis in order to find out properties such as the probability of sending a minimum number of messages within a given time limit. Probability can also be used to quantify unreliable or unpredictable behaviour, like in sensor networks where properties such as component failure and packet loss can be described probabilistically. To describe the real-time requirements of embedded systems, which includes soft real-time constraints, we need to support a stochastic process representation with timed transitions between states.

A system in which transitions between states are given by probability distributions is considered to be a Markov chain. The probability distribution only depends on the current state, and not on the path that led to that state. This is known as the memoryless property. Although appropriate for modelling random phenomena, Markov chains are not appropriate for modelling of concurrent processes. For that, nondeterministic choices need to be allowed, and such a system is called a Markov decision process (MDP). Although PRISM can analyze several types of probabilistic models, discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), probabilistic automata (PAs), and probabilistic timed automata (PTAs), pCharts allows nondeterministic transitions that can be represented only as MDP or PTA models. A formal description of MDP and PTA is given in Appendices A.2 and A.3.

Models of a probabilistic nature can be created by Stochastic Petri Nets [Marsan].
Petri Nets are inherently nondeterministic, but Stochastic Petri Nets models are with a continuous time scale and do not allow nondeterminism\cite{King and Pooley, 2000}.

\textit{pState} uses PRISM as a backend probabilistic model checking tool. PRISM has been used to analyze a broad range of studies in the area of \textit{a communication network and multimedia protocols, power management systems, performance and reliability properties}, etc. The tool is open source with good manual, tutorial, and support.

### 2.3 Code Generation

From the pChart representation, \textit{pState} generates intermediate code and from the intermediate code, it generates input code for the probabilistic model checker or executable code for the whole model, or for selected portions without probabilistic transitions.

**Input Code for Probabilistic Model Checker.** If a pChart representation does not have timed transitions, an MDP model is generated, otherwise a PTA model. Input code together with the specified formulae is passed to the model checker, and the result of the verification is shown on the screen.

In \cite{Zhao et al., 2010} a translation of extended UML diagrams \cite{Jansen, 2003} to PRISM is proposed, but the parallel composed system is passed as a system of multiple models to PRISM. This process does not allow the representation of nested concurrency, only of top level concurrency. On the other hand, \textit{pState} translates arbitrarily nested parallel compositions and creates one model. In addition to the model checker input code, it is possible to generate executable code, so \textit{pState} is not only a PRISM front-end.
**Executable Code.** For a sub-model without probabilistic transitions, \( pState \) can generate either C code or restricted assembly code for some target micro-controller. We implement event-centric code generation, while other tools like RHAPSODY [Harel and Kugler, 2004], SCADE Suite [Esterel, 2015], Stateflow with Simulink [Mathworks, 2011] follow a state-centric implementation. Our event-centric implementation of code generation for micro-controllers is similar to IAR Visual State tool for implementing embedded applications based on state machines. Upon detecting an event, Visual State looks at a list of actions by using a function pointer table. We assume that events are processed fast enough so that we do not need an *event handler* as employed in Visual State to handle queuing.

Translation of *if-else* or *case* statements of intermediate code is straightforward and it is done by just converting them into assembly syntax. The translation of *parallel* statements needs extra processing since that statement has to be first converted into sequential compositions.

### 2.4 Model-based WCET Analysis

Following the observation that WCET should ideally be part of an integrated development environment of embedded real-time programs [Engblom et al., 2001], we integrated WCET calculation into \( pState \). Our method of WCET calculation is *static*, or *verification-based*. The upper bound of task execution is estimated on the code itself, taking into account the hardware architecture. Static analysis guarantees that the execution time will not exceed the upper bound, but sometimes this estimation may be pessimistic, which can be verified by *measurement-based* methods.

The idea to use model checking for WCET analysis was proposed in [Metzner, 2004]. In our implementation, from the assembly code, for each event (basic block),
the number of processor clock ticks is calculated as a transition cost. Then the runtime can be calculated as a cost on a specified path using the PRISM model checker. It is known that timed-automata and model checkers like UPPAAL [Larsen et al., 2015] are suited for WCET calculation [Pavlidis, 2006; Béchennec and Cassez, 2011]. With the METAMOC [Dalsgaard et al., 2010] method of computing, WCET analysis can be reduced to computing the longest path of a control flow graph (CFG) represented as a timed automaton. We are not aware that any probabilistic model checker has been used for the automatic execution time calculation of systems specified by hierarchical state machines. The advantage is the possibility to verify events execution times in the early phase of design, during the specification process by a probabilistic model checker.

**WCET of Events with Invariants.** The connection of timing analysis with invariants as code annotations was used for finding infeasible paths before [Gulwani et al., 2009; Holsti et al., 2008; Engblom et al., 2001]. In our work, invariants are specified directly in the model itself, not inserted in the code.

Invariants are verified on PRISM models. If the invariants are true, in the next step the conditions of event transitions are verified. For that type of verification we use Yices 2.2. [Dutertre, 2014], an open source satisfiability modulo theories (SMT) solver. Yices is efficient and can run on multiple platforms. Parts of the code that do not satisfy invariants are excluded from WCET calculation.

**Probabilistic WCET.** Real-time systems can be classified as hard, firm or soft [Laplante, 1992]. In hard real-time systems a deadline should always be less than the WCET; missing a single deadline leads to catastrophic failure. In soft real-time systems, missing deadlines affects performance, but a system is still considered to be
operational. For firm systems, sporadically missing a deadline is not a problem, but if that happens too often, it may lead to failure. Therefore, in soft and firm real-time systems, it is useful to quantify execution time distribution. In addition to the calculation of the time needed to reach some state, the probability of reaching that state with a time bound is calculated.

2.5 Documentation

Tools like Rational Rhapsody [IBM, 2015] or SCADE LifeCycle Reporter [Esterel, 2015] are capable of creating detailed and complete reports of models in different formats like RTF, HTML etc. They also allow creation of new templates for report generation, and creation of reports according to standards like DO-178, which is a document dealing with the safety of avionic software. The report contains general project description, software architecture, and model information. Our tool is not capable of creating these documents, but model documentation should be part of the model development process and it is part of a holistic tool. Our state documentation is similar to UML state machines contents, but in a UML the comment box has the upper right corner bent, also known as a note symbol [OMG, 2009].

A design in pState can be documented in text boxes on the model itself and passed to generated code. There are three types of comments, general comments, state comments, and transition comments. The main reason for including the documentation is to justify the design. Passing the comments to the generated code allows forward and backward traceability, which would be necessary for the certification of generated code. Each comment can be connected to either a state or a transition by a dashed comment line. If it is not connected, it is associated to the state surrounding the comment box. General comments are associated with the root state.
State Documentation. As the generated code is event-centric, i.e. states become variables and events become procedures, the comments about a state are inserted in the generated code where the state is declared.

Transition Documentation. A transition comment is inserted into the generated code of the transition event. Timed transitions do not have an associated event name, but a name is generated for the corresponding procedure, and the comment is associated with that procedure in the same way as for untimed events.

2.6 Summary

We designed a holistic modelling language pCharts. The associated tool, pState, allows to comprehensively specify both a system and its environment. The goal is to increase engineering efficiency by automatic quantitative and qualitative design verification and code generation directly from the model.
Chapter 3

Overview of the Translation Scheme of pCharts

We give an overview of the translation scheme of pCharts to probabilistic guarded command. First, we show the translation scheme for probabilistic and timed events with cost in plain state diagrams, and then the translation scheme for hierarchy, concurrency, and communication.

3.1 Probabilistic Transitions

A probabilistic transition consists of a non-empty set $ss$ of source states, an event name $E$, an optional guard $g$, a Boolean expression, and a non-empty set of probabilistic alternatives. Each probabilistic alternative consists of a probability $p_i \in [0..1]$, an optional body $a_i$, where each $a_i$ is a statement without loops but possibly with broadcasts, optional cost specifications $C = e$, where $c$ is a cost name and $e \geq 0$ is a real-valued expression, and a non-empty set of target states $ts_i$. If the source or target consists of more than one state, these originate from or go to concurrent states.
Furthermore, the sum of the probabilities of all alternatives must be 1. We use the transition representation shown in Figure 3.1. States are symbolized by (rounded) boxes. The node \( p \) is left out if there is only one probabilistic alternative.

\[
\sum_{i=1}^{n} p_i = 1
\]

Figure 3.1: pCharts transition

Figure 3.2 illustrates the structure of a state label. A state consists of an optional state name \( S \), a possibly empty list of declarations, an optional state invariant \( p \), a Boolean expression, and optional cost specifications \( c = e \), where \( c \) is a cost name and \( e \geq 0 \) is a real-valued expression. A declaration declares a local event \( E \), a constant \( C = e \), where \( e \) is a constant expression, an integer subrange variable \( i : l..u \), where \( l, u \) are constant expressions with \( u \geq l \), a Boolean variable \( b : bool \), an array \( A \) of \( n \) integers in the range from \( k \) to \( v \), where \( n, k, v \) are constant expressions with \( v \geq k \) and \( n \geq 1 \), or an array \( B \) of \( m \) Boolean variables.

\[
S; E; C = e; i : l..u; b : bool; A[n : k..v; B^m : bool; \ldots
\]

\( p \) \( c = e \)

Figure 3.2: State label structure

Costs attached to transitions are “one-time” costs, like the decrease of life expectancy of a component when switching on and off, or the count of the number of message transmissions. For states with timed transitions, costs attached to the state are the costs of being in the state one time unit.


3.2 Timed Transitions

For the timed transition after we use two equivalent notations: after(t tu) or t.. tu, where the transition occurs when the clock reaches at least t time units (tu), t ∈ N; the time unit can be specified in milliseconds (ms), seconds (s), hours (h), or days (d).

The transition on which a time-out event is generated if the last occurrence of event e is t time units ago, is introduced in the original statecharts paper [Harel, 1987] and named timeout(e,t).

A timed event which occurs when the edge is enabled between two specified time delays t₁ and t₂, where 0 < t₁ ≤ t₂, and t₁, t₂ ∈ N is called a between time event. We use the syntax t₁ tu .. t₂ tu, to specify this transition. For instance, if a transition happens between 2ms and 5ms, we use the notation 2ms..5ms, but we also allow using ”between”, i.e. between(2ms, 5ms). To specify a transition which happens exactly at specific time we use syntax t tu, and this has the same meaning as between(t tu, t tu).

In the specification of environments we allow two kinds of stochastic transitions: exponential and uniform, as in [Jansen, 2003, 2013]. In the transition on exp(t tu), the delay is defined by an exponential distribution with an average duration of t time units. The timed transition unif(t₁, t₂,tu) indicates a uniformly distributed delay with the given minimum duration of t₁ and maximum duration t₂ time units. In general, any distribution can be represented as the combination of continuous exponential and continuous uniform distribution, but our implementation is a discrete approximation. Binomial, normal, Bernoulli, Poisson and other distributions from the exponential family are not supported, but the implementation would be similar to the exponential distribution.

Stochastic events can be used only in the specification from which code for the
model checker is generated. Currently we do not generate executable code for stochastic events.

Specification of absolute time transitions, like the transition to happen at May 1st 2016, Noon is not supported. Our semantics does not include spontaneous transitions, so event before(t) is neither supported.

3.3 Basic Charts Operation

pCharts consists of a finite number of states and transitions between those states. Upon an event, a system may evolve from one state into another. We represent the states of a state diagram as a variable of an enumerated set type [Sekerinski, 1998].

\[
\begin{align*}
\text{r} & : \{S_1, \ldots, S_n\} \\
\text{in}(S) & \triangleq r = S, \\
\text{goto}(S) & \triangleq r := S, \\
\text{cost}(S_1) & \triangleq c_1, \text{cost}(S_2) \triangleq c_2
\end{align*}
\]

Figure 3.3: State representation

On some event, a transition takes only place if in the current state there is a transition on that event. Otherwise, the event is ignored. Suppose only one transition in the system for event named E exists. Boolean expression guard [g], cost specifications \$c, and action /a are optional. Actions are assumed to be instantaneous. Operation \( \text{op}(E) \) of event E returns a set of priced guarded commands, a pair with a guarded command and a non-negative real value, written as \( \text{op}(E) = g \rightarrow S \ \$c \), where S is the target state.

By the symbol \( \parallel \) we represent independent (parallel) composition of statements.
Statement \texttt{skip} is the empty statement, and symbol // is prioritizing choice.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (S1) {$S_1$};
  \node (S2) [right of=S1] {$S_2$};
  \draw[->] (S1) -- node[above] {$l : E[g]c/a$} (S2);
  \node (op) [right of=S2] {$\text{op}(E) \equiv (\text{in}(S_1) \land g \rightarrow \text{goto}(S_2)) \parallel a \ c$}
        [right of=S2] {$\text{// \text{skip}}$};
\end{tikzpicture}
\caption{Event transition}
\end{figure}

In case of several transitions labelled with $E$, the one starting from the current state is taken, if any transition is taken at all. By the symbol [] we specify nondeterministic choice. States $S'_1,..S'_n$ do not have to be distinct:

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (S1) {$S_1$};
  \node (S1') [right of=S1] {$S'_1$};
  \draw[->] (S1) -- node[above] {$l_1 : E[g_1]c_1/a_1$} (S1');
  \node (S2) [right of=S1'] {$S'_2$};
  \node (S3) [right of=S2] {$S'_3$};
  \draw[->] (S1') -- node[above] {$\ldots$} (S2);
  \node (S4) [right of=S3] {$S'_n$};
  \node (S5) [right of=S4] {$\ldots$};
  \node (S6) [right of=S5] {$\text{// \text{skip}}$};
\end{tikzpicture}
\caption{Several transitions on the same event}
\end{figure}

In case of a probabilistic transition, each alternative consists of a probability $p_i \in [0..1]$, an optional body $a_i$, where each $a_i$ is a statement that may include broadcasts. By the symbol $\oplus$ we specify probabilistic choice.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (S0) {$S_0$};
  \node (S1) [right of=S0] {$S_1$};
  \node (S2) [right of=S1] {$S'_2$};
  \node (S3) [right of=S2] {$S'_3$};
  \node (S4) [right of=S3] {$S'_4$};
  \node (S5) [right of=S4] {$\ldots$};
  \node (S6) [right of=S5] {$\text{// \text{skip}}$};
\end{tikzpicture}
\caption{Probabilistic transition}
\end{figure}

Each timed transition has a unique label, written $l$ below. We introduce the shorthand $g \xrightarrow{t} S$ for a guarded command executed at time $t$. In a PTA this involves

\[ \sum_{i=1}^{n} p_i = 1 \]
a clock variable whose value is tested in the guard and whose value is reset in the body of the guarded command:

\[
S_1 \xrightarrow{l : t[g]\$c/a} S_2
\]

\[
t \in n_1..n_2 \mid exp(n) \mid unif(n_1,n_2)
\]

\[
op(l) \equiv in(S_1) \land g \xrightarrow{t} goto(S_2) \parallel a \ \$c
\]

Figure 3.7: Timed transition

### 3.4 Exponential Timed Transition

By the \(exp\) distribution we can specify the probability of transitions as a function of time. Most of the transitions will happen in the first few ticks after transition is enabled, but some transitions can happen much later. The transition

\[
\text{Figure 3.8: Exponential distribution delay}
\]

is interpreted as

\[
exp(X) \equiv 1 - e^{-t/X}, \quad \text{for} \quad e^{-t/X} > \epsilon
\]

\[
exp(X) \equiv 1, \quad \text{for} \quad e^{-t/X} \leq \epsilon
\]

where \(t\) is time when \(S_0\) is entered, \(t \in [0..K]\), and \(e^{-K/X} \leq \epsilon\). When state \(S_0\) is entered, \(t\) is set to 0, and is increased at each transition. Constant \(X > 0\) is the rate, or exponential scale. This transition is defined as a sequence of probabilistic transitions from source state \(S_0\) to target state \(S_1\).
Translation of stochastic transitions (exponential and uniform) are represented as a sequence of PRISM [Hinton et al., 2006; Oxford, 2014] probabilistic guarded commands. A PRISM module is described by commands, comprising a guard and one or more updates. Updates are multiple assignment statements of the form:

\[
[\text{guard}] \rightarrow \text{prob}_1 : \text{update}_1 + ... + \text{prob}_n : \text{update}_n
\]

The guard is a predicate over all the variables in the model, and each update describes a transition probability prob if the guard is true. We use PRISM's symbolic verification engine, Multi-Terminal Binary Decision Diagrams (MTBDDs) [Wang and Kwiatkowska, 2005], with both Stochastic games and Digital clock model checking methods.

The transition in Figure 3.8 is represented by a PRISM command in which with probability \( e^{-1/X} \) goes back into \( S_0 \) and with probability \( 1 - e^{-1/X} \) goes to \( S_1 \). The discrete representation of \( \text{exp}(X) \) has infinitely many steps, so we introduce model termination epsilon (\( \epsilon \)). If the probability to go back to state \( S_0 \) is less than \( \epsilon \), the loop is terminated. In the current implementation, tolerance or epsilon is \( 10^{-5} \), but we consider to allow this number to be specified by the user. Lowering \( \epsilon \) exponentially increases the state space and can make the verification process much longer or even impossible because PRISM can handle efficiently only models with up to \( 10^8 \) states.
Example:

The implementation is a discrete approximation of the exponential cumulative distribution function.

Let $X=3$, $t=1$. Then $\exp(X) = 1 - e^{(-1/3)} \approx 1 - 0.716 = 0.284$. This means that after one time unit the probability that the transition will be taken is $\approx 28.4\%$. The probability $p$ that at time $t = 1$ the transition will not be taken is $p \approx 1 - 0.284 = 0.716$, or $71.5\%$. The PRISM input code which represents the exponential transition from $S_0$ to $S_1$ is given in Listing 3.1.

The maximum number of time ticks in the model is specified by constant $K$. Since termination epsilon is $\epsilon = 10^{-5}$, the constant is $K = 35$, that is calculated in the process of PRISM code generation. The probability to reach state $S_1$ in $K$ ticks is $0.99999143$, so the probability to go back to $S_0$ is less than $\epsilon$. Integer variable $t$ is initialized to 0 and can have values from 0 to $K$.

3.5 Uniform Timed Transition

By this transition we can specify that the likelihood of the transition is equal for any time in the range when the transition is enabled. A uniform timed transition
is interpreted as

\[
\text{unif}(t_1, t_2) \equiv 0, \text{ for } (t < t_1)
\]
\[
\text{unif}(t_1, t_2) \equiv (t - t_1)/(t_2 - t_1), \text{ for } (t_1 \leq t \leq t_2)
\]
\[
\text{unif}(t_1, t_2) \equiv 1, \text{ for } (t_2 < t)
\]

where \( t \) is time in state \( S_0 \). This transition can be understood as a sequence of probabilistic transitions from source state \( S_0 \) to target state \( S_1 \), where \( n \) is initially \( t_2 - t_1 \). We introduce a pseudo-state as a \textit{fat dot} to indicate that a new value for \( n \) is assigned after a probabilistic alternative.

**Example:**

Let \( t_1 = 3 \) and \( t_2 = 6 \), and \( n = t_2 - t_1 = 3 \). Then \( \text{unif}(t_1, t_2) = (t - t_1)/n = (t - 3)/3 \) for \( t_1 \leq t \leq t_2 \). The guarded command code which represents the uniform transition from \( S_0 \) to \( S_1 \) is shown in Listing 3.2.

This command can be executed on a PRISM Markov Decision Protocol (MDP) model as shown in Listing 3.2. Unfortunately, PTA models do not support dynamic
Listing 3.2: \textit{unif}(3, 6) in MDP model

\[
\text{□ (state=S0) & (n>0) & (3<t) & (t<6) ->} \\
((n-1)/n):(state'=S0) & (n'=n-1) & (t'=t+1) + 1/n:(state'=S1);
\]

changes of probability, as specified by \((n-1)/n\) where \(n\) is an integer variable. To implement \textit{unif}(t_1, t_2), we need to \textit{unroll the loop} and specify the probability of a transition for each clock value between \(t_1\) and \(t_2\). In our example, for \textit{unif}(3, 6), the code which can be handled by PTA in Listing 3.3.

Listing 3.3: \textit{unif}(3, 6) in PTA model

\begin{verbatim}
const double p1=0.667; const double p2=0.5;
...

invariant
    (state=S0 => x <=t2)
endinvariant

[] (state=S0 & (t1<=x) & (t1<t) & (t<=t1+1) -> p1:(state'=S0) & (t'=t+1)+(1-p1):(state'=S1));
[] (state=S0 & (t1<=x) & (t1+1<t) & (t<=t1+2) -> p2:(state'=S0) & (t'=t+1)+(1-p2):(state'=S1));
[] (state=S0 & (x>=t2) & (t>=t2) -> (state'=S1));
\end{verbatim}

We initialize \(t_1 = 3, t_2 = 6, p1 = (N-1)/N = 1/3, p2 = (N-2)/(N-1) = 1/2\) and \(N = t_2 - t_1 = 3\), and variable \(t\) is initialized to \(t_1 + 1\). In this implementation, we have more guarded commands than in the MDP implementation, but the state space is the same since we do not use variable \(n\) to calculate the probability in the loop. By the formula

\[
P_{\min} = \mathbb{Q}[F (state = S_1 & t \leq T)]
\]

we can find out the probability to be in state \(S_1\) in some time units e.g. seconds.
### 3.6 Hierarchy

Composite states can have *substates* or *children*. If the system is in a state with substates, it is also in exactly one of those substates. Conversely, if a system is in a substate of a superstate, it is also in that superstate. In pCharts, following statecharts, a superstate with substates is drawn by nesting.

\[
\begin{align*}
S_1 & : c_1 \\
T_1 & : d_1 \\
& \cdots \\
T_n & : d_n \\
& \cdots \\
S_m & : c_m
\end{align*}
\]

where

\[
\begin{aligned}
& r : \{S_1, \ldots, S_m\}, \\
& s : \{T_1, \ldots, T_n\}
\end{aligned}
\]

\[
\begin{aligned}
& cost(T_1) \triangleq d_1, \ldots, cost(T_n) \triangleq d_n, \\
& cost(S_1) \triangleq c_1, \ldots, cost(S_m) \triangleq c_m
\end{aligned}
\]

When entering a superstate, the substate to be entered has to be specified as well. This is expressed by letting the transition arrow point to a specific substate. If we have two transitions on the same event \( E \), as shown in Figure 3.14, the transition going out from the superstate will have higher priority. If the guard \( g \) is *true*, the transition from \( S_1 \) to \( S_2 \) will be taken, otherwise, the transition from \( T_1 \) to \( T_2 \). Without

#### Table 3.1: Transition \( \text{unif}(3s,6s) \) probability

<table>
<thead>
<tr>
<th>( T )</th>
<th>( \text{unif}(3s,6s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leq 3 )</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0.33</td>
</tr>
<tr>
<td>5</td>
<td>0.66</td>
</tr>
<tr>
<td>( \geq 6 )</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 3.12: Hierarchical states*

*Figure 3.13: Entering superstate*
condition \( g \) on the transition from the superstate, the transition inside the superstate would never be taken.

\[
\begin{align*}
op(E) & \equiv (\text{in}(S_1) \land g \rightarrow \text{goto}(S_2)) \land d, \\
& \| \text{in}(S_1) \land \text{in}(T_1) \land \neg g \rightarrow \\
& \text{goto}(T_2) \land c) \\
& \quad \text{// skip}
\end{align*}
\]

Figure 3.14: Priority transition

### 3.7 Concurrency

Concurrency is expressed by orthogonality: a system can be in two independent states simultaneously. This is drawn by splitting a state with a dashed line into independent substates, each of which consists of a number of states in turn. A state with concurrent substates is entered by a fork into states in each of the concurrent substates. This corresponds to setting the variables for all the concurrent states:

\[
\begin{align*}
r & \colon \{S_1, S_2\}, \quad s \colon \{Q, T\} \\
q & \colon \{Q_1, \ldots, Q_m\}, \quad t \colon \{T_1, \ldots, T_n\} \\
\nop(E) & \equiv (\text{in}(S_1) \rightarrow \text{goto}(S_2) \parallel \text{goto}(Q_1) \\
& \parallel \text{goto}(T_1) \land c) \\
& \quad \text{// skip}
\end{align*}
\]

Figure 3.15: State entered by fork

Two concurrent states may have transitions on the same event, shown in Figure 3.16. In case this event occurs, these transitions are taken simultaneously. This corresponds to the parallel composition of the transitions. It has implications on the
global variables which can occur in the conditions and the actions; a variable can only be assigned by one action only.

\[
\begin{align*}
\text{op}(E) & \equiv (\text{in}(S_1) \land \text{in}(Q_1) \land \text{in}(T_1)) \rightarrow \\
& \quad \text{goto}(Q_2) \parallel \text{goto}(T_2) \ (c + d), \\
& \quad \text{in}(S_1) \land \text{in}(Q_1) \land -\text{in}(T_1) \rightarrow \\
& \quad \text{goto}(Q_2) \ c, \\
& \quad \text{in}(S_1) \land \text{in}(T_1) \land -\text{in}(Q_1) \rightarrow \\
& \quad \text{goto}(T_2) \ d) \\
& \text{// skip}
\end{align*}
\]

Figure 3.16: Parallel composition

3.8 Communication

Communication between concurrent states is possible in three ways: first, concurrent states can communicate by global variables. These can be set in actions and read in actions and conditions, following the rules for variables given earlier. Secondly, the condition or the action of a transition may depend on the current substate of a concurrent state. Thirdly, concurrent states can communicate by broadcasting of events. On a broadcast of an event, all concurrent states react simultaneously. Events are either generated internally through a broadcast or externally by the environment. Broadcasting an event corresponds to calling the operation for that event. On Figure 3.17, the initial configuration is \((Q_1, T_1)\). On external event \(E\), state \(Q\) changes from \(Q_1\) to \(Q_2\), and broadcasts event \(F\), which changes state \(T\) from \(T_1\) to \(T_2\). After event \(E\), the chart is in \((Q_2, T_2)\).
Figure 3.17: Broadcast event

This shows how transitions can be represented graphically using pCharts and presents a scheme for the translation of pCharts transitions into an intermediate code. The translation is similar to abstract machine notation (AMN) of the B method translation \cite{abrial1996, sekerinski1998} extended by probabilistic transition and cost. From this code we can generate either input code for probabilistic model checker or executable code for a specified target.
Chapter 4

Formal Definition and Normalized pCharts

In this chapter, we formally define pCharts components in a textual form and then describe the translation into a flat structure as a step in translation to code. This chapter is based on [Nokovic and Sekerinski, 2014].

4.1 Chart Components

A pChart is a structure with states, transitions, expressions, types, and statements that are defined in turn.

States. We assume Variable and Event are variable and event names, Basic, AND, XOR are finite, mutually disjoint sets of state names, Composite = AND \cup XOR is the set of composite states, and State = Basic \cup Composite is the set of all states:
Root ∈ XOR

parent : State − {Root} → State  

var : State → (Variable → Type)  
ev : State → PEvent  

inv : State → Expr  

cost : State → Expr  

Symbol ↦ represents partial function; only variables local to the state are declared.

All states form a tree that is rooted in Root, formally

\[ Root ∈ parent^*[\{s\}] \]

for any \( s ∈ State \), where \( r^* \) is the transitive and reflexive closure of relation \( r \) and \( r[S] \) is the image of the set \( S \) under \( r \). We let the relation \( children \) be the inverse of \( parent \), i.e. \( children = parent^{-1} \). Basic states don’t have children,

\[ children[Basic] = \{\}. \]

The children of an AND state are said to be concurrent, the children of an XOR state are said to be exclusive. The children of an AND state must be XOR states,

\[ children[AND] ⊆ XOR. \]

The variables of Root are the global variables, the events of Root are the global events, and the invariant of Root is the global invariant.

**Transitions.** For the transitions of a chart we assume that Transition is a finite set of transitions and Alternative is a finite set of probabilistic alternatives:
The state \textit{Root} must not be the source or target of any transition, \( \text{Root} \notin \text{source}(t) \) and \( \text{Root} \notin \text{target}(d) \) for any \( t \in \text{Transition} \) and \( d \in \text{alt}(t) \). The default transition of an \textit{XOR} state \( s \), if defined, must go to a descendant of \( s \), i.e. \( \text{target}(\text{default}(s)) \subseteq \text{children}^+[\{s\}] \), where \( r^+ \) is the transitive closure of relation \( r \). The set \textit{Time} is defined as:

\[
\text{Time} \doteq \text{between}(\mathbb{N}_{\geq 0}^+, \mathbb{N}_{\geq 0}^\infty) \cup \exp(\mathbb{N}_{\geq 0}^+) \cup \text{unif}(\mathbb{N}_{\geq 0}^+, \mathbb{N}_{\geq 0}^+)
\]

Timed transitions can be \textit{regular} (\textit{between}) or \textit{stochastic} (\textit{exp} or \textit{unif}).

The \textit{closest common ancestor} \( \text{cca}(ss) \) of a set \( ss \) of states is the state that is a proper ancestor of each state in \( ss \) and all other common ancestors are also its ancestor. We write \( x r y \) for the pair \((x, y)\) belonging to relation \( r \).

\[
c = \text{cca}(ss) \equiv c \in \text{parent}^+[ss] \land (\forall a \in \text{State} . a \in \text{parent}^+[ss] \Rightarrow a \text{parent}^* c)
\]

The closest common ancestor exists and is unique for any non-empty set of states.
that does not include the Root state. States $r$, $s$ are orthogonal, written $r \perp s$, if their closest common ancestor is an AND state and neither is an ancestor of the other. A set $ss$ of states is called orthogonal, written $\perp ss$, if every pair of distinct states of $ss$ is orthogonal. All source states of a transition must be orthogonal, $\perp source(t)$ for all $t \in Transition$ and targets of an alternative must be orthogonal, $\perp target(d)$ for all $d \in Alternative$. The scope of a transition is the state closest to Root through which the transition passes.

$$scope(t) \equiv \text{cca}(source(t) \cup (\bigcup d \in \text{alt}(t) . target(d)))$$

For the transition $t$ from Figure 4.1, $scope(t) = S$. The path from state $s$ to a set $ss$ of descendants of $s$ is the set of those states that are descendants of $s$ and ancestors of states in $ss$, excluding $s$ but including the states of $ss$.

$$\text{path}(s, ss) \equiv \text{children}^+[\{s\}] \cap \text{parent}^*[ss]$$

For the transition from Figure 4.1 $\text{path}([U], \{X, Z\}) = \{X, Z, W, Y, V, S\}$. The states exited by a transition are all the states on the path from the scope of the transition to its sources. The states explicitly entered by a transition $t$ are all the states on the path...
from the scope of the transition to a specific probabilistic alternative; if an alternative targets a descendant of an \textit{AND} state, then other states may be implicitly entered as well. In general, \textit{entered}(s, d) for state $s$ and alternative $d$ targeting a descendant of $s$ are all states on the path from $s$ to the target of $d$.

\[
\text{exited}(t) \equiv \text{path(scope}(t), \text{source}(t))
\]
\[
\text{entered}(s, d) \equiv \text{path}(s, \text{target}(d))
\]

For the transition from Figure 4.1 \textit{exited}(t) = \{U, T\} and \textit{entered}(U, t) = \{V, W, X, Y, Z\}.

Given a state set $ss$, the \textit{implicit children} are those children of \textit{AND} states of $ss$ that are not in $ss$. As children of \textit{AND} states are \textit{XOR} states, all implicit children are \textit{XOR} states. If a chart enters $ss$, it also enters all its implicit children.

\[
\text{imp}(ss) \equiv \text{children}[ss \cap \text{AND}] - ss
\]

The \textit{completion} of an alternative $d$ starting at state $s$ and targeting descendants of $s$ is the set of all alternatives that are taken when $d$ is taken: it adds to $d$ the default alternatives of explicit \textit{XOR} targets of $d$ and all default alternatives of implicitly
entered states.

\[
\text{comp}(s, d) \triangleq \{(s, d)\} \cup (\bigcup r \in (\text{target}(d) \cap \text{XOR}) \cup \text{imp}(\text{entered}(s, d)) \\
. \text{comp}(r, \text{default}(r)))
\]

In Figure 4.2 (a), we have that \(\text{target}(t) = \{U\}\), \(\text{default}(U) = u\), and therefore \(\text{comp}(S, t) = \{(S, t), (U, u)\}\). In (b), for \(t\) we have that \(\text{entered}(S, t) = \{Z, Y, V\}\) and \(\text{imp}(\text{entered}(S, t)) = \{W\}\). As \(\text{default}(W) = u\), we get \(\text{comp}(U, t) = \{(U, t), (W, u)\}\).

Certain XOR states are required to have a default initial state: a default alternative must be defined for the root state, \(\text{Root} \in \text{dom default}\), and any XOR state that is the target of some alternative or that is being implicitly entered as it has an AND ancestor that is being entered. Formally this means that \(\text{default}\) must be defined such that \(\text{comp}(\text{scope}(t), d)\) is well-defined for all \(t \in \text{Transition}\) and \(d \in \text{alt}(t)\).

Before code generation, the validation performs three checks on charts: (1) Composite states must not be childless, AND states must have at least two children, each child of an AND state must be an XOR state; (2) all XOR states have initial transitions; (3) transitions between concurrent states are not allowed.

**Expressions.** A chart expression is composed from program variables, state tests in \(S\), where \(S\) is any state except \(\text{Root}\), and functions \(\text{fn}\) applied to zero or more arguments:

\[
\text{Expr} ::= \text{Variable} \mid \text{in} S_1, \ldots, S_m \mid \text{fn}(\text{Expr}_1, \ldots, \text{Expr}_n)
\]

A function without arguments must be an integer, Boolean, or real constant. A function can also be one of the unary operators \(\neg e, -e\), one of the binary arithmetic, Boolean, and relational operators \(e + e, e - e, e * f, e \text{ div } f, e \mod f, e \land f, e \lor f, e \Rightarrow f\).
\( e \leftarrow f, e = f, e \neq f, e < f, e \leq f, e > f, e \geq f \), or the logarithm, minimum, or maximum function, \( \log(e_1, e_2), \min(e_1, \ldots, e_m), \max(e_1, \ldots, e_m) \). The iterator \texttt{map} applies a function to each element of an array, producing an array, and the iterator \texttt{fold} applies a binary operator to an array, producing a single value: \texttt{map \ e \_1 \ e \_2}, \texttt{fold \ e \_1 \ e \_2}.

\[
\text{S}; x:0..10; y:0..10; z:0..10 \\
T \xrightarrow{E/z := \min(x, y)} U
\]

\[
\text{S}; x:0..50; b\^5:0..10 \\
T \xrightarrow{E/x := \text{fold} + b} U
\]

(a) \hspace{2cm} (b)

Figure 4.3: Examples of \texttt{max} function and \texttt{fold} iterator

In Figure 4.3 on event \( E \) the chart goes from \( T \) to \( U \). In (a), the function \texttt{max} returns the maximum of integers \( x, y \). In (b), \texttt{fold} returns the sum of array elements. This is equivalent to \( x := b[0] + b[1] + \cdots + b[4] \).

\textbf{Type.} A \textit{chart type} is either an integer subrange, Boolean, real, or array.

\[
\text{Type} ::= \text{integer..integer} \mid \text{bool} \mid \text{real} \mid ^\text{integer} : (\text{integer..integer} \mid \text{bool})
\]

The partial function \( \text{type} : \text{Expr} \rightarrow \text{Type} \) determines the type of an expression. The type of a variable is determined by its declaration; the scope rules of languages with nested structures apply here to nested states. If variable \( v \) occurs in the body of a transition with scope \( S \), then \( \text{decl}(v, S) \) is the closest ancestor, or \( S \) itself, where \( v \) is declared:

\[
\text{decl}(v, S) \triangleq \text{if } v \in \text{dom}\ \text{var}(S) \text{ then } S \text{ else } \text{parent}(S)
\]
Thus, if \( v \in Variable \) occurs in state \( S \), its type is:

\[
type(v) \equiv \text{var}(\text{decl}(v, S), v)
\]

While expressions can be of any type, variables can only be of subrange or Boolean type. An expression \( e \) is well-typed if \( type(e) \) is defined. Transition guards, state invariants, and conditions of conditional statements have to be of Boolean type. Probabilities of alternatives, costs of states, and costs of transitions have to be of type real.

**Statements.** Statements are inductively constructed as follows. Assuming that \( b \) is a Boolean expression, \( xv \) is a list of unique variables, \( ev \) is a list of expressions of the same length as \( xv \), \( Q, R \) are statements, \( pv \) is a list of real expressions, and \( QV \) is a list of statements of the same length as \( pv \), the set \( Statement \) consists of:

- **skip** the empty statement, always enabled
- **stop** the always blocking statement
- \( xv := ev \) the multiple assignment statement, always enabled
- \( g \rightarrow R \) guarded command, enabled if \( g \) holds and \( R \) is enabled
- \( Q \parallel R \) nondeterministic choice between \( Q \) and \( R \), enabled if either one is
- \( Q \parallel R \) prioritizing choice, \( Q \) if \( Q \) enabled, else \( R \)
- \( Q \parallel R \) independent (parallel) composition of \( Q \) and \( R \), provided the assigned variables are disjoint, enabled if both \( Q \) and \( R \) are
- \( pv : QV \) probabilistic choice among \( QV \) with corresponding probability of \( pv \), provided \( \sum pv = 1 \)

Thus, a statement is either *blocking* or *enabled*.

A *chart statement* is either **skip**, a multiple assignment, a parallel composition, or
a conditional. In addition, a chart statement can broadcast event \( E \in \text{Event} \), simply written as \( E \). All assignment statements have to be type-correct, i.e. the types of the left and right hand side have to agree, and all broadcast statements have to be conflict-free, in a sense to be defined shortly. The grammar of chart statement is:

\[
\text{ChartStatement} ::= \begin{align*}
&\text{if } \text{Expr} \text{ then ChartStatement} | \text{else ChartStatement} \\
&| \text{ChartStatement} \parallel \cdots \parallel \text{ChartStatement} \\
&| \text{Variable}, \ldots, \text{Variable} := \text{Expr}, \ldots, \text{Expr} \\
&| \text{Event}
\end{align*}
\]

Target code is created by translating intermediate code to executable as described in Chapter 6. The abstract syntax of the executable code is:

\[
\text{TargetStatement} ::= \begin{align*}
&\text{if } \text{Expr} \text{ then TargetStatement} | \text{else TargetStatement} \\
&| \text{TargetStatement}; \ldots; \text{TargetStatement} \\
&| \text{Variable} := \text{Expr} \\
&| \text{case } \text{Variable} \text{ of } \text{State} : \text{TargetStatement} \\
&| \quad \ldots \text{State} : \text{TargetStatement} \\
&| \text{call } \text{Event} \\
&| \text{var } \text{Variable} = \text{Expr}; \text{Statement}
\end{align*}
\]

**Costs.** Let \( \Sigma \) be set of states, \( \text{Act} \) a (finite) set of actions, and \( \mathcal{D} \) a discrete probability distribution. We define an untimed probabilistic statement as \( \Sigma \leftrightarrow (\mathcal{D} \Sigma \times \mathbb{R}) \), and a timed statement as \( \Sigma \times (\text{Act} \cup \mathbb{R}_{\geq 0}) \rightarrow (\mathcal{D} \Sigma \times \mathbb{R}) \). We assume \( P \) to be a precondition, \( S \) to be a program statement, and \( Q \) to be a postcondition. The Hoare
triple

\[ \{ P \} S \{ Q \} \quad (4.1) \]

means if precondition \( P \) is true before \( S \) is executed, and if the execution of \( S \) terminates, then postcondition \( Q \) is true afterwards. For \( S = x := E \$c \), where \( E \) is some expression, \( c \) costs, and \( d \) final costs we have

\[ \{ P \} x := E \$c \{ Q, d \} \equiv (P \Rightarrow Q[x \leftarrow E]) \land d = c \quad (4.2) \]

For \( S = \text{stop} \)

\[ \{ P \} \text{stop}\{Q, d\} \equiv \text{true} \quad (4.3) \]

For \( S = S_1 \parallel S_2 \)

\[ \{ P \} S_1 \parallel S_2 \{ Q, d \} \Leftarrow \]

\[ \{ P \} S_1\{Q, c_1\} \land \{ P \} S_2\{Q, c_2\} \land d = c_1 \max c_2 \quad (4.4) \]

For \( S = S_1\| S_2 \)

\[ \{ P \} S_1\| S_2 \{ Q_1 \land Q_2, d \} \Leftarrow \]

\[ \{ P \} S_1\{Q_1, c_1\} \land \{ P \} S_2\{Q_2, c_2\} \land d = c_1 + c_2 \quad (4.5) \]
Formulas. Properties are specified according to the following concrete syntax:

\[
\begin{align*}
Property & \ ::= \ ?(Probability \ | \ Reward)(.\min \ | .\max \ | > \ real \ | < \ real)[Bound][Target] \\
Bound & \ ::= \ F < \ Time \\
Target & \ ::= \ '((Expr)'')' \\
Probability & \ ::= \ P \\
Reward & \ ::= \ $Identifier \\
Time & \ ::= \ digit\{digit\}(d \ | \ h \ | s \ | ms) \\
Identifier & \ ::= \ letter\{letter \ | \ digit\}
\end{align*}
\]

Properties are attached to a state or written in the special property box. Currently, only simple properties can be attached to states. For more complex properties, which include more than one condition, property boxes have to be used. For properties attached to a state \( pState \), creates a PCTL formula by automatically taking into account the state hierarchy.

Conventions. In charts, if a transition guard \([g]\) is missing, it is assumed to be \textit{true}. If a transition \(/B\) body is missing, it is assumed to be \textit{skip}. If there is only one probabilistic alternative, its probability of 1 is left out.

4.2 Translation to Flat Guarded Commands

The independent (or parallel) composition \( Q \parallel R \) is a generalization of multiple assignment. It is well-defined if the variables assigned by \( Q \) and \( R \), are disjoint. For the variables accessed by \( Q \) and \( R \), their initial values are taken. So, there is no interleaving. The probabilistic choice is more commonly written as \( p_1 : Q_1 \oplus \cdots \oplus p_n : Q_n \) or, using comprehension notation, \( \oplus i \in I . p_i : Q_i \). As both nondeterministic choice
and independent composition are associative, we write simply $Q_1 \parallel \cdots \parallel Q_n$ and $Q_1 \parallel \cdots \parallel Q_n$ without parenthesis, or, using comprehension notation, $\{ i \in I : p_i : Q_i \}$ and $\{ i \in I : p_i : Q_i \}$. The conditional statement is defined in terms of above statements:

$$\text{if } b \text{ then } Q \triangleq (b \rightarrow Q) \parallel (\neg b \rightarrow \text{skip})$$

$$\text{if } b \text{ then } Q \text{ else } R \triangleq (b \rightarrow Q) \parallel (\neg b \rightarrow R)$$

The following rules are used to eliminate parallel composition. Let $b$ be a boolean expression, $Q, R, S$ statements.

$$\text{if } b \text{ then } Q = \text{if } b \text{ then } Q \text{ else skip} \quad (4.6)$$

$$(\text{if } b \text{ then } Q \text{ else } R) \parallel S = \text{if } b \text{ then } (Q \parallel S) \text{ else } (R \parallel S) \quad (4.7)$$

$$(Q \parallel R) \parallel S = Q \parallel R \parallel S \quad (4.8)$$

$$Q \parallel \text{skip} = Q \quad (4.9)$$

Applying these rules transforms a statement to nested if-then-else statements with the innermost statements being multiple assignments. That form is used for WCET calculation as shown in detail later on.

Probabilistic guarded commands can be defined by predicate transformers [Morgan et al., 1996]; for our purposes, a simpler definition by relations between the initial state and distributions over the final state is sufficient, i.e. as functions of type $\Gamma \rightarrow \mathcal{PD}\Gamma$, where $\mathcal{D}\Gamma = \Gamma \rightarrow [0,1]$ such that $\sum d = 1$ for all $d \in \mathcal{D}\Gamma$, distribution $d$ is not 0 for finitely many states of $\Gamma$, and $\mathcal{P}$ is the powerset operator. Intuitively, a statement first makes an arbitrary nondeterministic choice among distributions and then a probabilistic choice according to that distribution. The independent composition leads to a cross product of the state space, i.e. if $Q : \Gamma \rightarrow \mathcal{PD}\Gamma$ and $R : \Delta \rightarrow \mathcal{PD}\Delta$...
then $Q \parallel R : (\Gamma \times \Delta) \rightarrow \mathcal{PD}(\Gamma \times \Delta)$. The following properties of statements will be used, where $b, c$ are Boolean expressions, $P, Q, R$ are statements, $xv, yv$ are lists of variables, and $ev, fv$ are lists expressions of same length as $xv, yv$.

\[
xv := ev \parallel yv := fv = xv, yv := ev, fv
\] (4.10)

\[
b \rightarrow c \rightarrow Q = (b \land c) \rightarrow Q
\] (4.11)

\[
b \rightarrow (P \parallel Q) = (b \rightarrow P) \parallel (b \rightarrow Q)
\] (4.12)

\[
(P \parallel Q) \parallel R = (P \parallel R) \parallel (Q \parallel R)
\] (4.13)

\[
(b \rightarrow Q) \parallel R = b \rightarrow (Q \parallel R)
\] (4.14)

Above laws are known for standard (non-probabilistic) statements. We will also need following laws involving probabilistic choice:

\[
(p : P \oplus q : Q) \parallel R = p : (P \parallel R) \oplus q : (Q \parallel R)
\] (4.15)

\[
p : (q_1 : Q_1 \oplus q_2 : Q_2) \oplus r : R = p \times q_1 : Q_1 \oplus p \times q_2 : Q_2 \oplus r : R
\] (4.16)

Probabilistic choice can be distributed inside guarded nondeterministic choice provided that one of the guards is true:

\[
p : (b_1 \rightarrow P_1 \parallel b_2 \rightarrow P_2) \oplus q : Q =
\]

\[
b_1 \rightarrow (p : P_1 \oplus q : Q) \parallel b_2 \rightarrow (p : P_2 \oplus q : Q) \text{ if } b_1 \lor b_2
\] (4.17)

To see why the condition is necessary, assume that $b_1 = b_2 = \text{false}$: the left side blocks with probability $p$ and choses $Q$ with probability $q$, but the right side always blocks.

We present two translations of charts to target statements, $op$ which generates guarded commands for regular events and $top$ which generates guarded commands
for timed events.

**State Model.** For representing the configuration (or “state”) of a chart, we use a model that makes it easy to express independent updates of concurrent states and state tests of any state in the hierarchy, and can directly be mapped to a programming language [Sekerinski 1998]. For every XOR state $s$, including Root, a variable $lc(s)$, ranging over $uc(c)$ for every child $c$ of $s$, is declared. We interpret $lc(s)$ and $uc(s)$ to be the state $s$ starting with a lowercase or an uppercase letter. (We assume that these variables and their values are distinct from variables declared in the chart.) This model allows to define the state test and state assignment for any state $s$ that is a child of an XOR state by inspecting and assigning the variable for that state:

\[
\begin{align*}
test(s) & \equiv lc(parent(s)) = uc(s) \\
assign(s) & \equiv lc(parent(s)) := uc(s)
\end{align*}
\]

In Figure 4.4, $test(s)$ and $assign(s)$ are defined for all states $s$ except $R$, $V$ and $X$:

\[
\begin{align*}
test(P) & \equiv root = P & assign(P) = root := P \\
test(Q) & \equiv root = Q & assign(Q) = root := Q \\
test(S) & \equiv r = S & assign(S) = r := S \\
test(T) & \equiv r = T & assign(T) = r := T \\
test(U) & \equiv r = U & assign(U) = r := U \\
test(W) & \equiv v = W & assign(W) = v := W \\
test(Y) & \equiv x = X & assign(Y) = x := Y \\
test(Z) & \equiv x = Z & assign(Z) = x := Z
\end{align*}
\]

Manipulation of configurations is expressed in terms of $test$ and $assign$. The predicate $in(ss)$ tests whether the current configuration is in the set $ss$; similarly $goto(ss)$
Figure 4.4: Transitions on event E

sets the current configuration to ss.

\[
\begin{align*}
\text{in}(ss) & \equiv \forall s \in ss \cap \text{children}[\text{XOR}] \cdot \text{test}(s) \\
\text{goto}(ss) & \equiv \parallel s \in ss \cap \text{children}[\text{XOR}] \cdot \text{assign}(s)
\end{align*}
\]

As special cases we have:

\[\text{in}(\{\}) = \text{true} \quad \quad \text{goto}(\{\}) = \text{skip}\]

The statement \text{goto}(ss) is well-defined if the states of ss are not exclusive. For example, in Figure 4.4, \text{goto}(\{S, W, Y\}) is well-defined, but \text{goto}(\{S, Y, Z\}) is not.

**Event Translation.** The trigger of a transition \(t\) is true if the transition guard is true and if the chart is in all source states of the transition. The effect of a transition \(t\) is a probabilistic choice among its alternatives: each alternative is completed and for each completion, the body of the completion is executed and the system moves to
all entered states of the completion.

\[
\text{trigger}(t) \equiv \text{in(exited}(t)) \land \text{guard}(t)
\]
\[
\text{effect}(t) \equiv \oplus c \in \text{alt}(t) . \text{prob}(c) :
\]
\[
(\parallel (s,d) \in \text{comp}(s,t) . \text{body}(d) \parallel \text{goto}(\text{entered}(s,d))\)
\]

For Figure 4.4, noting that for \(a\), \(\text{comp}(P,a) = \{(P,a),(R,b),(V,d),(Y,f)\}\), and \(\text{comp}(W,e) = \{(W,e)\}\), and \(\text{body}(b) = \text{skip} = \text{body}(d) = \text{body}(f)\) we get for:

\[
\text{trigger}(a) \equiv \text{in}(\{P\}) \land \text{true} \equiv \text{test}(P)
\]
\[
\text{effect}(a) = \text{body}(a) \parallel \text{goto}(\{Q\}) \parallel \text{goto}(\{S\}) \parallel \text{goto}(\{W\}) \parallel \text{goto}(\{Z\})
\]
\[
= i := 0 \parallel \text{assign}(Q) \parallel \text{assign}(S) \parallel \text{assign}(W) \parallel \text{assign}(Z)
\]

For the transition \(g\) we get:

\[
\text{trigger}(g) \equiv \text{in}(\{Z\}) \land \text{in}_g W \equiv \text{test}(Z) \land \text{test}(W)
\]
\[
\text{effect}(g) = \text{body}(g) \parallel \text{goto}(\{Z\})
\]
\[
= i := i + 1 \parallel \text{assign}(Z)
\]

By \(\text{in}_g W\) we indicate a Boolean state test relative to transition \(g\). The operation of an event \(E\) is a statement that captures the joint meaning of all transitions in a chart on \(E\). For brevity, let \(\text{Trans}(E,s)\) stand for the set of transitions on event \(E\) with scope \(s\):

\[
\text{Trans}(E,s) \equiv \{t \in \text{Transition} \mid \text{event}(t) = E \land \text{scope}(t) = s\}
\]

The function \(\text{op}(E)\) recursively visits all transitions on \(E\), starting with those on the outermost scope, \(\text{Root}\). In case there is a choice between transitions with the same scope, one is selected arbitrarily. In case there is a choice between transitions on different scopes, transition on outer scopes are given priority. All transitions on the
same event in concurrent states are taken in parallel. Of all transitions in an exclusive 
state, at most one can be taken. Following statecharts, the response to an event on 
which no transition can take place is to do nothing, i.e. skip, rather than to block.

\[
\begin{align*}
\text{op}(E) & \equiv \text{scopeop}(E, \text{Root}) \\
\text{scopeop}(E, s) & \equiv ([\] t \in \text{Trans}(E, s) . \text{trigger}(t) \rightarrow \\
& \quad \text{effect}(t)[F\text{\ op}(F)]) // \text{childop}(E, s) \\
\text{childop}(E, s) & \equiv \text{case } s \text{ of} \\
& \quad \text{XOR} : [\] c \in \text{children}[\{s\}] - \text{Basic} . \text{test}(c) \rightarrow \\
& \quad \quad \text{scopeop}(E, c) // \text{skip} \\
& \quad \text{AND} : || c \in \text{children}[\{s\}] - \text{Basic} . \text{scopeop}(E, c) \\
& \quad \text{end}
\end{align*}
\]

A transition may also broadcast an event, say $F$, either explicitly or implicitly in one 
of the alternatives of its completion; any transition taken on $F$ is taken jointly with 
those on $E$ and if no transition on $F$ can be taken, $F$ behaves as skip. Thus, the 
meaning of broadcasting $F$ is that of $\text{op}(F)$. We write $S[F\setminus T]$ for replacing event $F$
by $T$ in statement $S$. Operation $\text{op}$ generalizes to the case when more than one event
is broadcast. The function $\text{childop}(E, s)$ is defined only if $s$ is an XOR or AND state,
which the mutually recursive definition respects at each call. In Figure 4.4 there is one
event $E$ with four transitions on it. The simplifications are that choice over an empty
range is stop, that parallel composition over empty rage is skip. For any statement
$P$, skip $\parallel P = P$; stop $[\] P = P$; stop // $P = P$; $g \rightarrow P$ $[\] h \rightarrow P = g \land h \rightarrow P$;
true → P = P. With simplification we get:

\[ op(E) = test(P) \rightarrow i := 0 \parallel assign(Q) \parallel assign(S) \parallel assign(W) \parallel assign(Z) \]

// (test(P) → skip)

[] test(Q) →

(test(S) → 0.7 : assign(T) \oplus 0.3 : assign(U))

// skip

\| (test(W) \land test(Z) \rightarrow i := i + 1 \parallel assign(Z))

// skip

\| (test(W) \land i > 2 \rightarrow assign(X))

// skip)

For an operation to be conflict-free, there must not be conflicting multiple assignments to the same variable. Such a conflict may appear if the body of a transition assigns to, say \( x \), and broadcasts an event that also assigns to \( x \). As chart configurations are modified by assignments to variables, this implies that no event can be transitively broadcast twice. By extension, event broadcasting cannot be cyclic \cite{Sekerinski and Zurob 2002}. More examples of \( op \) on event without probabilistic transitions can be found in \cite{Sekerinski 2008}.

For pCharts with timed transitions, the set of timed transitions with scope \( s \) is defined as:

\[ Trans(s) \equiv \{ t \in Transition \mid event(t) = Time \land scope(t) = s \} \]

A timed transition is a unique event, so we need to create an operation for each timed
transition. Operations of timed transitions are generated by top,

\[
\begin{align*}
top() & \equiv tscopeop(Root) \\
tscopeop(s) & \equiv (\{ t \in \text{Trans}(s) . ttrigger(t) \rightarrow teffect(t)[F\setminus op(F)] \} // tchildop(s) \\
tchildop(s) & \equiv \text{case } s \text{ of} \\
\quad \text{XOR : } & \{ c \in \text{children}\{s\} \} - \text{Basic . test}(c) \rightarrow tscopeop(c) // \text{skip} \\
\quad \text{AND : } & \{ c \in \text{children}\{s\} \} - \text{Basic . tscopeop}(c) \\
\quad \text{end}
\end{align*}
\]

where:

\[
\begin{align*}
ttrigger(t) & \equiv \text{in}(\text{exited}(t)) \land \text{guard}(t) \land \text{timeout}(t) \\
teffect(t) & \equiv \text{effect}(t) \parallel \text{reset} \\
timeout(t) & \equiv \text{true} (\text{clock} \geq t) \\
reset & \equiv \text{clock} := 0
\end{align*}
\]

By clock we indicate the timer tick on the scope of transition. In Figure 4.5 we get for transitions t and u:

\[
\begin{align*}
ttrigger(t) & \equiv \text{in}(\{U\}) \land \text{in}_y Y \land \text{timeout}(t) \\
& \equiv \text{test}(U) \land \text{test}(Y) \land \text{clock} \geq 10 \\
teffect(t) & = \text{body}(t) \parallel \text{goto}(\{V\}) \parallel \text{reset} \\
& = \text{assign}(V) \parallel \text{clock} := 0 \\
ttrigger(u) & \equiv \text{in}(\{X\}) \land \text{timeout}(t) \\
& \equiv \text{test}(X) \land (3 \leq \text{clock} \land \text{clock} \leq 4) \\
teffect(u) & = \text{body}(u) \parallel \text{goto}(\{Y\}) \parallel \text{reset} \\
& = \text{assign}(Y) \parallel \text{clock} := 0
\end{align*}
\]

While, by op we generate code for the single event, by top we create operations for
all timed transitions of a pChart. The code is created by first finding the scope of each timed transition, and then generating the list of guarded commands for each transition on a particular scope. Timed transitions may broadcast an event, say $F$, but timed transitions can not be broadcasted itself. If event $F$ is broadcasted, it has to be an untimed event since a transition taken on $F$ is taken jointly with those on the timed event, and if $F$ is a timed transition, the joint transition would be delayed. That violates the rule that transitions in charts are instantaneous.

PRISM allows only a flat collection of guarded commands of the form $b_1 \rightarrow S_1 []$
\[\cdots [] b_m \rightarrow S_m,\] where each $S_i$ is of the form $p_1 : A_1 \oplus \cdots \oplus p_m : A_n$ and each $A_i$ is a multiple assignment statement. We call this the normal form of an operation. For generating a normal form, first $\text{scopeop}(E, s)$ is equivalently expressed by making the guard explicit instead of writing “// skip”. With abbreviations

\[TE \equiv [] t \in \text{Trans}(E, s) . \text{trigger}(t) \rightarrow \text{effect}(t)[F \setminus \text{op}(F)]\]
\[TT \equiv \forall t \in \text{Trans}(E, s) . \neg \text{trigger}(t)\]

we have:

\[\text{scopeop}(E, s) = TE [] TT \rightarrow \text{childop}(E, s)\]
From the definitions we observe for effect\((t)\) in TE that goto\((\text{entered}(s, d))\) is a parallel composition of multiple assignments. If body\((d)\) also consists only of multiple assignments (or skip), then we can use equation (4.10) to transform effect\((t)\) into a single multiple assignment as needed for the normal form. If body\((d)\) contains conditionals, which by definition are of the form \((b \rightarrow Q) \parallel (c \rightarrow R)\), then first by (4.13) and (4.14) the guard and the choice can be “moved out”, and on the “top level” merged by (4.12) and (4.11) with other nondeterministic choices of TE. We note that each conditional statement leads to two “top level” choices. If effect\((t)\) contains a broadcast of an event, say \(F\), then that has to be replaced by \(\text{op}(F)\). We assume that \(\text{op}(F)\) is in normal form and show how to transform TE to normal form. More specifically, suppose that \(\text{op}(E)\) and \(\text{op}(F)\) are of the form:

\[
\begin{align*}
\text{op}(F) &= b_1 \rightarrow P_1 \parallel b_2 \rightarrow P_2 \\
\text{op}(E) &= c_1 \rightarrow (p : \text{op}(F) \oplus q : Q) \parallel c_2 \rightarrow R
\end{align*}
\]

We note that any operation is always enabled as \(P \parallel Q\) is enabled if either \(P\) or \(Q\) is, so \(P \parallel \text{skip}\) is always enabled. As \(\text{op}(F)\) is always enabled, \(b_1 \vee b_2\) must hold and we can use (4.17) to transform \(\text{op}(E)\) to

\[
\begin{align*}
\text{op}(E) &= c_1 \rightarrow (b_1 \rightarrow (p : P_1 \oplus q : Q) \parallel b_2 \rightarrow (p : P_2 \oplus q : Q)) \parallel c_2 \rightarrow R
\end{align*}
\]

which can then be brought into normal form by (4.12) and (4.11). This generalizes to more than two probabilistic alternatives and nondeterministic choices accordingly. This procedure is illustrated in Figure 4.6. There are two transitions on event \(E\), one probabilistic that broadcast event \(F\) with probability \(p\) and one deterministic.

Now we show inductively that the result of scopeop\((E, s)\) can be transformed to normal form, assuming that recursive calls are returning a normal form. Considering
TT \rightarrow \text{childop}(E, s) \text{ as above, there are two cases. If } s \text{ is an XOR state, we use (4.12) and (4.11) to simplify \text{childop}(E, s). That is, assuming \text{childop}(E, s) \text{ is of the form } b_1 \rightarrow P_1 \parallel b_2 \rightarrow P_2, \text{ where } P_1, P_2 \text{ are in normal form (and } b_1, b_2 \text{ are state tests) we obtain:}

TT \rightarrow \text{childop}(E, s) = TT \land b_1 \rightarrow P_1 \parallel TT \land b_2 \rightarrow P_2

As P_1, P_2 \text{ are in normal form, (4.12) and (4.11) can be used again to flatten the whole structure. If } s \text{ is an AND state, we use (4.13) and (4.12) to simplify \text{childop}(E, s). That is, assuming \text{childop}(E, s) \text{ is of the form } (b_1 \rightarrow P_1 \parallel b_2 \rightarrow P_2) \parallel (c_1 \rightarrow Q_1 \parallel c_2 \rightarrow Q_2), \text{ resulting the normal form returned by } \text{scopeop}(E, r), \text{ we obtain:}

TT \rightarrow \text{childop}(E, s) = TT \land b_1 \land c_1 \rightarrow (P_1 \parallel Q_1) \parallel TT \land b_1 \land c_2 \rightarrow (P_1 \parallel Q_2) \parallel
TT \land b_2 \land c_1 \rightarrow (P_2 \parallel Q_1) \parallel TT \land b_2 \land c_2 \rightarrow (P_2 \parallel Q_2)

Considering now } P_1 \text{ to be } r_1 : R_1 \oplus r_2 : R_2 \text{ and } Q_1 \text{ to be } s_1 : S_1 \oplus s_2 : S_2, \text{ where } R_1, R_2, S_1, S_2 \text{ are multiple assignment statements, we use (4.15) for “moving out” the probabilistic choice in } P_1 \parallel Q_1 \text{ and then use (4.16) to flatten the nested probabilistic}
alternatives:

\[ P_1 \parallel Q_1 = r_1 \times s_1 : (R_1 \parallel S_1) \oplus r_1 \times s_2 : (R_1 \parallel S_2) \oplus r_2 \times s_1 : (R_2 \parallel S_1) \oplus r_2 \times s_2 : (R_2 \parallel S_2) \]

Repeating this process brings then \( TT \rightarrow \text{childop}(E, s) \) in normal form and therefore \( \text{scopeop}(E, s) \) in normal form, which completes the induction. The procedure for transforming all operations in normal form consists of repeatedly picking an event that does not contain a broadcast and transforming its operation to normal form by first eliminating conditionals. All occurrences of broadcasts to that event are replaced by its operation. This is repeated as long as events have not been considered. In Figure 4.7 on event \( E \), a parallel probabilistic transition happens and the chart goes either into states \( \{R_1, S_1\} \) with probability \( 0.2 \times 0.9 = 0.18 \), or to states \( \{R_1, S_2\} \) with probability \( 0.2 \times 0.1 = 0.02 \), or to states \( \{R_2, S_1\} \) with probability \( 0.8 \times 0.9 = 0.72 \), or to states \( \{R_2, S_2\} \) with probability \( 0.8 \times 0.1 = 0.08 \).
The operation on event $E$ with simplification is

$$\text{op}(E) = ((\text{test}(U) \rightarrow 0.2 : \text{assign}(R_1) \oplus 0.8 : \text{assign}(R_2)) \parallel \text{test}(P) \rightarrow \text{skip})$$

$$\parallel ((\text{test}(X) \rightarrow 0.9 : \text{assign}(S_1) \oplus 0.1 : \text{assign}(S_1)) \parallel \text{test}(X) \rightarrow \text{skip})$$

$$= ((\text{test}(U) \land \text{test}(X)) \rightarrow 0.18 : (\text{assign}(R_1) \land \text{assign}(S_1))$$

$$\oplus 0.02 : (\text{assign}(R_1) \land \text{assign}(S_2))$$

$$\oplus 0.72 : (\text{assign}(R_2) \land \text{assign}(S_1))$$

$$\oplus 0.08 : (\text{assign}(R_2) \land \text{assign}(S_2))$$

$$\parallel \text{test}(U) \rightarrow 0.2 : \text{assign}(R_1) \oplus 0.8 : \text{assign}(R_2)$$

$$\parallel \text{test}(X) \rightarrow 0.9 : \text{assign}(S_1) \oplus 0.1 : \text{assign}(S_1))$$

$$\parallel \text{skip}$$

The translation of an event operation into normal form is an intermediate step to the generation of target code. In this step, the probabilities of parallel operations on the same event have to be multiplied.
Chapter 5

Model-based WCET Analysis with Invariants

Following the trend that part of the modern software engineering process is the integration of WCET [Kirner and Puschner, 2005], we implement automatic execution time calculation for each chart transition. This chapter is based on [Nokovic and Sekerinski, 2015c].

5.1 Introduction

The full integration of embedded systems into larger products leads to increased reliance on their correct service [Lokuciejewski and Marwedel, 2011]. In modern embedded systems, not only functional requirements and liveness, but also strict timing constraints must be satisfied. Properties of a program like a loop bound, exponential path space, path feasibility, etc., and properties of the environment like memory access time determine the program execution time. An established approach is to insert constraints as deadlines in the basic blocks of source code and, after
compiling, the assembly code is used in a low-level analysis to determine the execution time \cite{Lee and Seshia 2015}. If the calculated times are less than or equal to the specified deadlines, the timing constraints are satisfied.

pCharts allows the specification of timing constraints directly on the charts during the system specification process. Timing-analysis tasks are usually divided into two groups, (1) analyzing control flow properties and (2) calculating execution time of instructions or basic blocks of instructions \cite{Lee and Seshia 2015}. From a pCharts model, \textit{pState} automatically calculates the execution time of instructions, but for analyzing properties of the control flow, we use a model checker as the backend tool.

In our holistic modelling approach, from the same model, code generation and quantitative analysis is possible. The number of executable cycles is calculated for each basic block. The calculation of the upper bound of the program execution time is not possible in general (the halting problem), but we do not allow general loops or recursive function calls, so in our model programs always terminate, and automatic calculation of the upper bound is possible.

State invariants are conditions that are attached to states in a state hierarchy.
and specify what has to hold in that state. This gives a method for checking a chart against an invariant annotation. Invariants have been used before in finding infeasible paths [Holsti et al., 2008; Engblom et al., 2001; Gulwani et al., 2009], but they are assumed to be true. In our approach, the invariants are first verified by the probabilistic model checker and then used to find infeasible paths. Invariants that are not true are ignored. This is the first time that invariants are used on hierarchical state machine models with concurrency and broadcasting for the calculation the execution time of transitions. For example, in Figure 5.1, the invariant

\[ \text{in PortA.Off} \lor \text{in PortB.Off} \]

states that PortA is in Off, or PortB is in Off, or both are in Off. They cannot be in Through simultaneously, therefore the code will never execute the tick transitions in both PortA and PortB at the same run. This is a weak invariant, a stronger invariant would be

\[ \text{in PortA.Off} \oplus \text{in PortB.Off} \]

where \( \oplus \) is exclusive or. For the execution time calculation of the tick event, there is no difference if we use the weaker or stronger one. The event tick is generated by timer clock periodically, every 10ms. The time of both executions should not contribute to the WCET. Informally, in this case:

\[
WCET(t) = \max(WCET((10ms/tick) || (tick/readA)), \ WCET((10ms/tick) || (tick/readB)))
\]
Without the invariant, the WCET of transition $t$ would be calculated as

$$WCET(t) = WCET((10ms/tick) \|(tick/readA) \|(tick/readB))$$

Transition conditions under invariants are checked by the SMT solver Yices as the backend tool. Invariants used in this way are related to flow facts annotations [Kirner and Puschner, 2005]. Flow facts, an additional knowledge about the possible program control flow that cannot be derived automatically from the program, are added to the code. When parallel composition is sequentialized, all combinations of executable paths are created. But, if some conditions are never satisfied, the associated paths as excluded. In Section 5.3 we show how conditions can be automatically verified and unreachable paths removed from the executable code. Besides simplifying the generated code, this makes the WCET calculation more accurate. We illustrate in Section 5.4 how user-specified state invariants improve WCET calculation. Accurate WCET analysis can be done only at assembly or object code level [Kirner and Puschner, 2005]; therefore our model of nested hierarchical states has to be translated into assembly code. We show that state invariants, as high-level information, can be used to improve WCET estimation. Next, we demonstrate how the expected executed time can be specified directly on the transition, and then how we can automatically check if the specified expected execution time is feasible. When the calculated WCET is greater than expected, the design has to be alternated by introducing or removing states or update the design target (e.g. using a different processor, increasing the clock frequency of the target processor).
5.2 WCET Calculation Principles

We consider static or verification-based WCET calculation. The upper bound of the task execution time is estimated on the code itself taking the hardware architecture into account. Static analysis guarantees that the execution time will not exceed the upper bound, but this estimation may be too pessimistic, which can be verified by measurement-based methods. Other techniques for WCET calculation are simulations and path analysis method for the calculation of execution scenarios [Kirner and Puschner, 2005]. In general, methods of static program analysis include value analysis, control flow analysis (CFA), processor behaviour analysis, and symbolic simulation [Kirner and Puschner, 2005]. Some well-known WCET tools for static program analysis are aiT [AbsInt, 2015], Bound-T [Tidorum, 2016], OTAWA [IRIT, 2016], and SWEET [Wilhelm et al., 2008].

A method for analyzing real-time behaviour of a reactive synchronous system with special focus on statecharts is described in [Erpenbach and Altenbernd, 1999]. In addition to WCET and schedulability analysis of statecharts models, worst-case response time (WCRT) of synchronous models is introduced. The method for calculation of WCET is implemented in Statemate [Harel and Naamad, 1996].

WCET by model checking is introduced in [Metzner, 2004], showing that model checking is adequate for low level WCET analysis dealing with caches and pipelines. Using a similar method, pState first predict the timing of program instruction for dedicated hardware, and then the time of the specified path is calculated by the model checker using the predicted timing of program instructions. For a basic block, the number of processor ticks is introduced as the cost of the transition. The runtime is calculated as the cost on a specified path. A process of model-based design with static program analysis tools which guarantees safety requirements is described in Cruz.
Tools for determining the precise bounds of resource usage like StackAnalyzer and aiT can be integrated with automatic code generators. Timed-automata and model checkers like UPPAAL are suited for WCET calculation [Béchennec and Cassez, 2011]. METAMOC reduces computing of WCET to finding the longest path of control flow graph (CFG) represented as timed-automata [Dalsgaard et al., 2010]. User-specified invariants are not part of any of these approaches. The MARTE UML profile [OMG, 2015] for modeling and analysis of real-time and embedded systems is intended to replace the existing UML Profile for Schedulability, Performance and Time (SPT) [Kumar and Jasperneite, 2010]. Behaviour scenarios are annotated with expressions of variables (i.e. $wcet1). The annotated model indicates to the analysis tools what property has to be computed. While WCET calculation is part of the MARTE extension, we are not aware that it is possible to directly specify invariants on the UML MARTE extension.

State invariants as functional properties can be used to improve the estimation of WCET on synchronous models [Raymond et al., 2013]. Some of the functional properties are hard and sometimes even impossible to find by static analysis of programs specified by high-level models. Invariants can be used to exclude infeasible program paths, which cannot be found by static analysis. In our approach invariants are part of the pCharts syntax, and semantic information represented by state invariants is specified at the design level. Existing WCET analyzers like aiT could calculate the WCET of executable code generated by pState, but the invariants need to be specified manually in the AbsInt general annotation file. That will slow down the entire process, but our goal is to increase engineering efficiency.

The application area for which pState was originally developed is embedded systems with 8- and 16-bit micro-controllers, e.g. as used in active RFID tags. Those applications have a small number of states, so the problem of scalability of model
checkers is not of primary concern. Data processing operations known as prefetching, delayed branching, and branch prediction, which can contribute to WCET on advanced microprocessors, are not present in those simple micro-controllers. We assume that the execution time of the instruction is independent of the instruction order \cite{Pavlidis2006}. There are three aspects of WCET analysis: representation level, flow facts, and execution-time modelling \cite{KirnerPuschner2005}. Representation is done by pCharts, flow facts annotation are specified by state invariants, and the representation is translated to a probabilistic timed automata model on which flow analysis is performed using model checking. By model checking we can prove that some state can be reached in a defined number of cycles, which can be done by finding the number of cycles, or deadline $T$ such that the minimal probability of reaching state $\phi$, $P_{\text{min}}(F(\phi_T))$, is the same for $T$ and $T+n$ cycles, where $n \in \mathbb{N}$. So, WCET calculation is the optimization problem of finding $T$ such that

$$P_{\text{min}}(F(\phi_T)) = P_{\text{min}}(F(\phi_{T+n})).$$

The number of cycles is specified as a cost. For execution-time modelling we use the number of microprocessor cycles.

By the implementation of the described procedure, we can verify the WCET of transitions early in the software development process, during requirement specification.

### 5.3 Transition with Specified Deadline

Figure 5.2 shows a transition from state $S$ to state $T$ on event $E$. Guard $x = 0$ represents the condition which must be satisfied for the transition to be taken, body
\[ x := x + 1 \] is the action that is executed when the transition is taken, and \( 20 \mu s \) is the specified deadline for that transition. Guard, action, and deadline are optional in charts. The translation \( op(E) \) is given by:

\[
op(E) = \text{if } in(S) \land (x = 0) \text{ then } goto(T) \parallel x := x + 1 \text{ else } \text{skip}
\]

To check if the WCET of the transition from \( S \) to \( T \) takes at most \( n \) cycles, we need to calculate the execution time on \( E \) by translating \( op(E) \) into assembly code. If the target is the RISC PIC16F6xx micro-controller, the code is in PIC assembly language, which we call \textit{picasm}. The instruction set has about 35 instructions divided into three groups, byte-oriented, bit-oriented and control operations \cite{MicrochipTechnologyInc2007}. Most of the instructions take one processor cycle, except jump and subroutine call which take two cycles. The operation \( op(E) \) can be executed if the chart is in state \( S \), written as \( in(S) \) and guard \( x = 0 \) holds. The effect of the transition is execution of the then branch \( x := x + 1 \). The chart is going to state \( T \), written as \( goto(T) \). With the simple translation scheme employed, the WCET of \( op(E) \) can be calculated compositionally:

\[
WCET(\text{picasm}(op(E))) = WCET(\text{picasm}(in(S))) + WCET(\text{picasm}(x = 0)) + WCET(\text{picasm}(goto(T) \parallel (x := x + 1)))
\]

The composition \( goto(T) \parallel (x := x + 1) \) is well-defined only if the variables assigned to \( goto(T) \) and \( (x := x + 1) \) are disjoint, but \( goto(T) \) and \textit{action} may read the variables assigned by the other. Parallel composition is a generalization of multiple
assignments and sequentialization of multiple assignments may need extra variables for temporary storage. The time to copy values into these variables has to be taken into account. Therefore, the sequentialization of multiple assignments should be done with a minimum number of auxiliary variables. Statement $\text{goto}(T) \equiv \text{root} := T$ and body of the transition do not assign joint variables, and do not read variables assigned to each other. Thus, this parallel composition is well defined and there is no need for an extra variable in creation of sequential composition, so

$$\text{goto}(T) \parallel (x := x + 1) = \text{goto}(T) ; x := x + 1$$

and the WCET of this parallel composition is

$$\text{WCET}(\text{picasm}(\text{goto}(T) \parallel (x := x + 1))) = \text{WCET}(\text{picasm}(\text{goto}(T))) + \text{WCET}(\text{picasm}(x := x + 1))$$

The generated assembly code is shown in Listing 5.1. Variable $r$ represents the root state in which $S$ and $T$ are nested. Next, we calculate the number of cycles for each component. For $in(S)$ the number of cycles is 3. The WCET of the transition

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Listing 5.1:</strong> Generated assembly code for $\text{op}(E)$</td>
<td></td>
</tr>
<tr>
<td><strong>picasm(op(E)) =</strong></td>
<td></td>
</tr>
<tr>
<td><strong>GUARD_0</strong></td>
<td></td>
</tr>
<tr>
<td>movf r, W ; $W := r$</td>
<td></td>
</tr>
<tr>
<td>xorlw S ; $W := W \oplus S$</td>
<td></td>
</tr>
<tr>
<td>btfss STATUS, 0x2 ; If $Z = 1$ skip</td>
<td></td>
</tr>
<tr>
<td>goto CONINUE_0</td>
<td></td>
</tr>
<tr>
<td><strong>ACTION_0</strong></td>
<td></td>
</tr>
<tr>
<td>movf x, W ; $W := x$</td>
<td></td>
</tr>
<tr>
<td>xorlw 0 ; $W := W \oplus 0$</td>
<td></td>
</tr>
<tr>
<td>btfss STATUS, 0x2 ; If $Z = 1$ skip</td>
<td></td>
</tr>
<tr>
<td>goto CONINUE_0</td>
<td></td>
</tr>
<tr>
<td>movlw 1 ; $W := 1$</td>
<td></td>
</tr>
<tr>
<td>addwf x, 1 ; $x := W + x$</td>
<td></td>
</tr>
<tr>
<td>movlw T ; $W := T$</td>
<td></td>
</tr>
<tr>
<td>movwf r ; $r := W$</td>
<td></td>
</tr>
<tr>
<td><strong>CONTINUE_0</strong></td>
<td></td>
</tr>
</tbody>
</table>
on event $E$ calculated by $pState$ is

$$WCET(picasm(op(E))) = 3 + 3 + 2 + 2 = 10\ cycles$$

One processor cycle for micro-controller RISC PIC16F6xx running at 4MHz is 1$\mu$s. In Figure 5.2, we specify $\Delta = 20\mu$s, which is greater than the WCET of 10$\mu$s, so we can say that the transition on event $E$ satisfies the specified requirement for this particular target. To simplify notation, we leave out the translation to assembly if understood from the context: $WCET(S)$ stands for $WCET(target_{asm}(S))$ for statement $S$.

**WCET of Parallel Composition.** Figure 5.3 shows a transition from state $T_1$ to $T_2$ on event $E$. Suppose: $stat_i \equiv if\ c_i\ then\ S_i$, $c_1 \equiv x > 0$, $c_2 \equiv x < 0$, where $S_i$ are statements. Then:

$$op(E) = if\ in(T_1)\ then\ goto(T_2)\ ||\ stat_1\ ||\ stat_2\ else\ skip$$

The parallel composition $stat_1\ ||\ stat_2$ amounts to:

$$(if\ c_1\ then\ S_1)\ ||(if\ c_2\ then\ S_2)$$

= by [4.6]

$$(if\ c_1\ then\ S_1\ else\ skip)\ ||(if\ c_2\ then\ S_2\ else\ skip)$$

= by [4.7]

if $c_1$ then ($S_1\ ||\ if\ c_2\ then\ S_2\ else\ skip$) else ($skip\ ||\ if\ c_2\ then\ S_2\ else\ skip$)

= by [4.9] nd symmetry of ||

73
\[
\text{if } c_1 \text{ then } (S_1 \parallel \text{if } c_2 \text{ then } S_2 \text{ else skip}) \text{ else (if } c_2 \text{ then } S_2 \text{ else skip)}
\]

\[= \text{ by (4.7) and symmetry of } \parallel\]

\[
\text{if } c_1 \text{ then (if } c_2 \text{ then } S_1 \parallel S_2 \text{ else } S_1) \text{ else (if } c_2 \text{ then } S_2 \text{ else skip)}
\]

\[= \text{ definition of } c_i
\]

\[
\text{if } x > 0 \text{ then (if } x < 0 \text{ then } S_1 \parallel S_2 \text{ else } S_1) \text{ else (if } x < 0 \text{ then } S_2 \text{ else skip)}
\]

For verification we use the Yices 2.2 SMT solver \cite{Dutertre2014}. If the SMT solver verifies that \(x > 0 \land x < 0\) is unsatisfiable and all \(S_i\) do not assign to common variables, above is equivalent to

\[
\text{if } x > 0 \text{ then } S_1 \text{ else (if } x < 0 \text{ then } S_2 \text{ else skip)}
\]

and the WCET of \(op(E)\) is calculated by \(pState\) as:

\[
\text{WCET}(op(E)) = \text{WCET}(in(T_1)) + \text{WCET}(goto(T_2)) + \text{WCET}(x > 0) + \\
\max(\text{WCET}(S_1), \text{WCET}(x < 0) + \text{WCET}(S_2))
\]

That is, infeasible paths are excluded. The calculation is conservative: if the SMT solver fails to establish that the conjunction of guards is unsatisfiable, the WCET of the code with all paths is taken.

\section{WCET Taking State Invariants into Account}

State invariants are used to improve WCET calculation by providing additional information which makes the analysis feasible and tight. WCET takes a particular scheduling policy into account \cite{Erpenbach1999}. In state \(S_2\) of Figure 5.4, the invariant is \(i \geq 1\). As \(i \geq 1\) holds, the \textit{then} path of the if statement will never be taken and always \textit{else} will be executed. So, the WCET time of transition
on event $F$ calculated by $pState$ is equivalent to execution of $else$ part of the action statement. If the condition $i = 0$ holds, can be decided by a SMT tool. The calculation of the execution time depends of the answer from the solver. For the code passed to Yices, the answer is $unsat$. This means $j := 0 || i := i + 1$ will never be executed, and the execution time for this branch is not calculated. Without the invariant $i \geq 1$, $pState$ calculates the WCET of the operation $F$ as $max(A_{then}, A_{else})$ where $A_{then}$ is the execution time of $then$ part of if action statement. Since $A_{then} > A_{else}$, the calculated WCET would be too pessimistic. This simple example might be done with data flow analysis, but verification with invariants is more general.

### 5.5 WCET Taking into Account Timed Transitions

We consider two timed transitions $k$ and $l$ as in Figure 5.5. Transition $k$ takes place exactly $t_1$ time units after the state $S$ is entered, if event $E$ does not occur in the meantime. Similarly, transition $l$ happens $t_2$ timed units after state $T$ is entered.

The operation on event $E$ for the transition in Figure 5.5 includes cancelling timed transition $k$ and scheduling timed transition $l$ is:
\[ op(E) = \text{if } in(S) \land g \text{ then } (\text{goto}(T) \parallel a) ; \text{ cancel } k ; \text{ schedule } l \text{ else } \text{skip} \]

For the WCET of \( E \), we need to add the time of scheduling transition \( l \) and the time for cancelling transition \( k \). Again, the transition is considered to be correct if the WCET is less or equal to specified deadline \( d \), i.e. \( WCET(op(E)) \leq d \). A scheduler is created separately and is called from the generated code for scheduling and cancelling transitions. Scheduling of timed transitions involves adding new events into a data structure of scheduled events, sorted by time and priority. The procedure \textit{cancel} contains a loop to search for the scheduled timed transition which has to be removed. When the first due-to event is removed, other events are shifted down. In the current implementation of the scheduler, the time to schedule an event if the data structure is empty is 52 processor cycles. The time needed to go through the loop and sort events is 141 cycles per event. The WCET of scheduling is less than or equal to the number of scheduled transitions multiplied by 141 plus 52 cycles. One run through the loop of a cancelling timed transition takes 35 cycles, while the time to shift the scheduled event takes 75 cycles. In a similar way, we have that the WCET of cancelling a transition is less than or equal to the number of scheduled transitions multiplied by 35 plus 75 cycles. The algorithms for schedule and cancel are presented in Chapter 6.

\[
WCET(\text{cancelling of timed transition}) = 35 \times \# \text{of scheduled transitions} + 75
\]

For the transition on event \( E \) from Figure 5.5, \( pState \) calculates

\[
WCET(op(E)) \leq 10 + 52 + 35 \times 1 + 75 = 172 \text{ cycles}
\]

The number processor cycles are related to the current implementation. Any modification of the algorithm may have an impact on those numbers.
5.6 WCET of Simple Timed Transitions

Consider timed transition $t$ in Figure 5.6. The operation $\text{top}$ on timed transition $t$ is given by

$$
\text{top}(t) = \text{if } \text{in}(S) \text{ then goto}(T) \text{ else skip}
$$

The transition takes place 100ms after state $S$ is entered. To calculate the WCET we need to add the time required by the scheduler to start this transition.

$$
\text{WCET}(\text{top}(t)) = \text{WCET}(\text{in}(S)) + \text{WCET}(\text{goto}(T)) + \text{WCET}(\text{scheduler cycle})
$$

The scheduler uses TIMER0 of the target processor to generate an interrupt every 1ms. In the interrupt service routine, the due-time for each scheduled event is decreased. This is done in a loop and the execution time depends on the number of scheduled events. Additionally, in each scheduler cycle, possible events need to be polled.

$$
\text{WCET}(\text{scheduler cycle}) = \text{WCET}(\text{decrease due time}) + \text{WCET}(\text{event polling})
$$

Based on the current implementation, the time to decrease the event due-time is 98 cycles if there is only one task scheduled. Polling of events has two parts, polling of timed transitions and polling of input events on the micro-controller’s ports:

$$
\text{WCET(\text{event polling})} = \text{WCET(\text{polling timed transitions})} + \text{WCET(\text{polling input events})}
$$
The WCET for polling timed transitions is 22 cycles, while the WCET of polling input events is only five cycles plus the time to execute the input event.

\[ WCET(\text{event execution}) \leq 22 + \sum \{5 + \text{input event } I \text{ execution} \mid \text{input event } I\} \]

For timed transition \( t \), taking into account calculations from Section 5.3, and assuming that (1) there is only one event to execute, and (2) there are no input events, \( pState \) calculates the WCET as:

\[
WCET(picasm(op(t))) = 100\text{ms} + 3\; \text{cycles [in(S)]} + 2\; \text{cycles [goto(T)]} + 98\; \text{cycles [tick]} + 22\; \text{cycles [event execution]}
\]

\[ = 100\; \text{ms} + 125\; \text{cycles} \]

If the target micro-controller runs on 4MHz, the execution time of one cycle is \( 1/(4\text{MHz}/4) \) which is \( 1\mu s \). In this case, \( WCET(op(t)) = 100.125\text{ms} \). If the specified deadline is \( \Delta \geq 125\mu s \), the transition execution time is satisfiable. But, if we need \( \Delta \geq 100\mu s \), we will find out during the specification phase that the requirement can not be implemented. In that case, a possible solution is to select a higher frequency of micro-controller clock. Instead of 4MHz, we can use an 8MHz crystal clock, which will fix the problem. By this approach, in the early phase of design, during specification, we can identify hardware limitations and make appropriate design decisions.

The operation of the timed transition in Figure 5.6 includes code that checks the source state, \( S \), before it takes the transition to another state. As the transition is only scheduled when the chart is in \( S \), that check is not needed but is still included to protect against faults like incorrectly scheduled events.

Long-running real-time systems are known to be prone to cumulative clock drifts. This can occur if the scheduling of new timed transitions is delayed by the time it takes to schedule the new transition. Cumulative drift is avoided here by having TIMER0
generate periodically an interrupt every 1ms and requiring that all transitions are completed within 1ms. That is, every transition, regular and timed, has an implicit deadline that corresponds to the timer resolution.

Comparing estimated with actual run time. For events from Figures 5.2, 5.5, and 5.6 we measured actual run time on PICDEM Lab Development Kit, 4MHz, PIC16F636 micro-controller. The results are shown in Table 5.2. The run-time is measured by Tektronix TDS 1002 two channel digital oscilloscope 60MHz, 1GS/s.

<table>
<thead>
<tr>
<th>Example</th>
<th>Operation</th>
<th>Guard</th>
<th>Calculated</th>
<th>Measured run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>$E$</td>
<td>true</td>
<td>10 cycles</td>
<td>10µs</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>false</td>
<td>10 cycles</td>
<td>6µs</td>
</tr>
<tr>
<td>5.5</td>
<td>schedule</td>
<td>true</td>
<td>52 cycles</td>
<td>52µs</td>
</tr>
<tr>
<td></td>
<td>cancel</td>
<td>true</td>
<td>110 cycles</td>
<td>104µs</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>true</td>
<td>172 cycles</td>
<td>166µs</td>
</tr>
<tr>
<td>5.6</td>
<td>$E$</td>
<td>true</td>
<td>100ms + 125 cycles</td>
<td>100ms + 122µs</td>
</tr>
</tbody>
</table>

Table 5.2: Estimated vs. actual run time

The micro-controller is running on a 4MHz clock, and the instruction cycle is 1µs. The measured running time is the same or lower than the calculated WCET in all cases. For the operation from Figure 5.2 if the guard $x = 0$ is false, the run time is 6µs. If the guard is true, the run time is 10µs. In both cases, the calculated WCET is 10 cycles, or 10µs because we do not know a value of variable $x$ when the execution time is calculated. For the operation from Figure 5.5, we measured the time to schedule transition $l$, the time to cancel transition $k$, and transitions take time. The measured run time is slightly lower than the estimated WCET time. We get similar result for the example from Figure 5.6 because $pState$ calculates the longest possible time a program can execute. To be valid, WCET must not underestimate, and to be useful must provide low overestimation [Engblom et al., 2001]. The $pState$ calculated WCET is close to the actual run time, but $pState$ never underestimates.
5.7 Modelling Probabilistic Timing Analysis

We can ask ourself questions like "What is the probability that the WCET is not met?". For the answer to that question, we need to perform probabilistic model checking of the system. To analyze WCET one needs to find out all execution time of individual tasks.

Probabilistic timing analysis can be either static or measurement based. Model based probabilistic WCET timing analysis is determined statically. Static probabilistic timing analysis (SPTA) is performed by calculating the convolution $\otimes$ of discrete probability distributions that describes execution time, or costs of the transition for each instruction. The probabilistic timing behaviour of an operation can be represented with an Execution Time Profile (ETP) expressed by the pair: $<\text{timing vector}, \text{probability vector}>$ [Abella et al., 2014]. It is assumed that distribution of instructions execution times across the whole test run, are independent.

To illustrate the technique we use the same example used in [Abella et al., 2014] for formal SPTA. Let $\mathcal{X}$ and $\mathcal{Y}$ be random variables that describe the execution time of two instructions $x$ and $y$. The SPTA is the convolution $\mathcal{X} \otimes \mathcal{Y}$. For instance let instruction $x$ execute within 1 cycle with probability 0.9, and $x$ executes in 10 cycles with probability 0.1. That is represented as $2 \times 2$ matrix

$$\mathcal{X} = \begin{pmatrix} 1 & 10 \\ 0.9 & 0.1 \end{pmatrix}$$

where the first row represents cycles or cost of the execution, and the second row represents the probability of that execution. Let instruction $y$ have the equal probability of 0.5 to execute in 2 or 10 cycles. The convolution $\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y}$ is
\[
\begin{pmatrix}
1 & 10 \\
0.9 & 0.1
\end{pmatrix}
\otimes
\begin{pmatrix}
2 & 10 \\
0.5 & 0.5
\end{pmatrix}
= \begin{pmatrix}
3 & 11 & 12 & 20 \\
0.45 & 0.45 & 0.05 & 0.05
\end{pmatrix}
\]

\[
= (\{3, 11, 12, 20\}, \{0.45, 0.45, 0.05, 0.05\}) = ETP_{x \otimes y}
\]

The example created by pCharts is shown in Figure 5.7. From the model, it is possible to calculate the execution time probability of instructions \(x\) and \(y\).

\[\text{Figure 5.7: Convolution } X \otimes Y \text{ in pCharts}\]

By the PCTL formula

\[P_{\text{min}} = \text{?}[F \text{ root } = F_i], \text{ for } 0 \leq i \leq 3 \quad (5.18)\]

we can calculate the probability of reaching any of the states \(F_0, \ldots, F_3\). The calculated probabilities are given in Table 5.3. The table gives only the probabilistic component of the ETP. To complete the ETP, we need to calculate the number of cycles to reach \(F_i\), for \(0 \leq i \leq 3\).

The cost associated with a transition depends on which \(x\) executes an alternative. In pCharts, we can specify transition costs or state costs, but not costs of an alternative branch. In our example, the costs of alternatives are given in Figure 5.8.
By the formula

$$ R\{"c"\} min = \min \left[ F \ root = F_i, \ for \ 0 \leq i \leq 3 \right] $$  \hspace{1cm} (5.19) 

the costs to reach any of final states $F_0, ... , F_3$ is calculated. The calculated costs are given in the Table 5.4.

<table>
<thead>
<tr>
<th>State</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_0$</td>
<td>3</td>
</tr>
<tr>
<td>$F_1$</td>
<td>11</td>
</tr>
<tr>
<td>$F_2$</td>
<td>12</td>
</tr>
<tr>
<td>$F_3$</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5.4: Costs calculated by $pState$

Combining probability the calculation in Table 5.5 with the costs calculation in Table 5.4 we get the executable time profile for instructions $x$ and $y$. The result is same as the one obtained by the convolution $X \otimes Y$. 

82
While in the classical approach requirements which specify probability and computational costs of instructions and the convolution calculation are separated, in a model-based calculation those processes are merged [Abella et al., 2014]. Once the specification is created as a pCharts model and query formulae are attached, pState can automatically calculate the probabilistic WCET for a particular state. From Table 5.5 we can see that 95% of the time, the execution of instructions will take 12 cycles or less.

<table>
<thead>
<tr>
<th>State</th>
<th>probability</th>
<th>costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_0$</td>
<td>0.45</td>
<td>3</td>
</tr>
<tr>
<td>$F_1$</td>
<td>0.45</td>
<td>11</td>
</tr>
<tr>
<td>$F_2$</td>
<td>0.05</td>
<td>12</td>
</tr>
<tr>
<td>$F_3$</td>
<td>0.05</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5.5: $ETP_{x \otimes y}$ calculated by pState

The calculation of ETP requires building a probability tree whose calculation time grows exponentially with the number of states. The average number of cycles for some instructions may be calculated directly from the model, by adding the final state, and a transition to that state. To calculate the average execution time of instructions $x$ and $y$, we create a model like the one shown in Figure 5.9.

Using the formula $R\{"c"\}_{min} =? [F (root = Final)]$, we can automatically calculate the average number of cycles needed to execute instructions $x$ and $y$. The result calculated by pState is 7.9 cycles.
5.8 Probabilistic WCET

The execution time of a transition in a pCharts model is specified as a cost of a transition. By analyzing reward properties, or costs to reach some state, we can calculate the probabilistic WCET. On each transition execution, time as costs is added to a global variable. In Figure 5.10, transition $t_1$ has three alternatives. On those alternatives, actions $A_1$, $A_2$, and $A_3$ are executed.

Action $A_1$ is taken with probability 0.01 or in 1% of time, action $A_2$ with probability 0.3, and action $A_3$ with probability 0.69. If execution of action $A_1$ takes time $t_{a1}$, execution of action $A_2$ takes $t_{a2}$, and execution of action $A_3$ takes $t_{a3}$ then the time to reach state state $S_4$ is given in the Table 5.6.
The worst-case execution time to reach state $S_4$ is

$$WCET = t_1 + \max(t_{a1} + t_2, t_{a2} + t_3, t_{a3} + t_4).$$

but this can happen only in 1% of time if $t_{a1} > t_{a2} \geq t_{a3}$. The probability of WCET can be useful in the design of real-time RT systems. Those systems can be classified as hard, firm or soft [Laplante 1992]. In firm real-time systems missing a few deadlines will not lead to failure, so in that case we can specify the deadline to reach $S_4$ to be lower than WCET. In our example the probability that the deadline will be missed three times in a row is very low 0.000001. So for firm RT systems, we can soften the deadline and we get:

$$WCET = t_1 + \max(t_{a2} + t_3, t_{a3} + t_4)$$

### 5.9 Summary

We implemented a framework for model-based WCET analysis of real-time systems. From a hierarchical representation, executable code in a low level language is generated. On the generated code, WCET of transitions can be calculated by counting the number of assembly instructions execution cycles. The precise WCET determination on a complex architecture is a challenging problem, but determining the WCET on 8 and 16-bit micro-controllers is easier since features like multi-stage pipelines and
caches are not present. Invariants for improving the WCET are independent of the architecture of the target processor. The \textit{pState} architecture allows in principle external tools for WCET to be plugged in. Existing WCET analyzers like AbsInt \cite{AbsInt2015} could calculate the WCET of executable code generated by \textit{pState}, but the invariants need to be stated in the AbsInt general annotation file; the annotation could in principle be generated by \textit{pState} as well. To exclude infeasible paths, conditions specified by state invariants and conditions created by sequentialization of parallel compositions are verified by the SMT solver Yices.

From models with probabilistic transitions, we can determine the probability to reach some state as well as the costs to reach that state. In the execution time analysis, the cost is calculated as the number of processor cycles. The probability to reach some state, and the time to reach that state are calculated as quantitative properties with the PRISM model checker as the backend tool.

In our models, we do not allow general \textit{while} loops and recursion, so the calculation of WCET is always possible. Actions can contain external calls, e.g. to I/O libraries, that are out of control of \textit{pState}: for those timing constraints cannot be specified. Thus, if loops or recursion are necessary, these can be implemented by an external call.

Worst-case-response-time (WCRT) that includes the impact of other transitions is especially important in concurrent systems. Invariants could be taken into account in a similar way as they are taken for calculation of WCET. Latency is the time passed from the moment when the event is registered to the moment when execution of the event starts; it depends on the implementation of the scheduler and I/O synchronization. Quantification of execution time is important information in the design of firm and soft real-time systems.
The Best Case Execution Time (BCET) is defined as the shortest time ever observed [Lee et al., 2007]. The BCET can be of interest to analyze the potential jitter, the difference between WCET and BCET. The average-case execution time (ACET) depends on the execution time distribution of the program. The BCET and ACET are important metrics in periodic systems, and jitter is known to be an important implementation aspect because it may create a scheduling problem. We are focused on reactive embedded systems that do not perform periodic tasks (most of the time they are in the sleep mode), and in that case jitter is not a problem.
Chapter 6

Code Generation

We describe the generation of embedded code for a micro-controller from pCharts. First we show the translation of simple event and then on an example we present the full structure of executable code, the part generated from the pCharts and the part that should be prewritten.

6.1 Transformations

The code generation is based on the recursive algorithm from Section 4.2. As an example, the operation $op(E)$ of the event $E$ in Figure 6.1 is as follows:

$$op(E) = (\text{root} = S0 \rightarrow x := x + 1 \parallel \text{root} := S1 \parallel s2 := P1 \parallel s3 := Q1) \//$$

$$(\text{root} = S1 \rightarrow s2 = P1 \rightarrow x := x - 1 \parallel s2 := P2) \// \text{skip}$$

$$||$$

$$(s3 = Q1 \rightarrow 0.2 : s3 := Q2 \oplus 0.8 : s3 := Q3) \// \text{skip}$$

The full description of generalized program statements skip, stop, multiple assignment, guarded statement, nondeterministic choice, probabilistic choice, and parallel
composition used to define the meaning of events, and the grammar of chart statement is presented in Section 4.1.

6.2 Executable Code

Target code is created by further translating the intermediate code, provided that there are no probabilistic transitions in the sub-chart for which code is to be generated. The intermediate code may contain parallel compositions emerging from broadcasting (transitions in concurrent states are taken in parallel) and multiple assignments. As multiple assignments are a special case of parallel composition, both are treated uniformly by introducing auxiliary variables and sequentializing, e.g. \((x := y \parallel y := x) = (x, y := y, x) = (\text{var} h = x; x := y; y := h)\). Specifications of costs/rewards are ignored for code generation. Nondeterministic choice with guarded choices is translated as if-then-else or case statements in the target code syntax. The
abstract syntax of the executable code is presented in Section 4.1. *pState* generates code for PIC16F6xx in C or assembly language, and Libelium/Arduino code for ATmega1281 micro-controller. Both are 8-bit RISC-based micro-controllers.

**PIC C Code.** We generate an *if − else* statement for single operation events, and a *case* statement if there is more than one operation on an event. The translation of parallel composition into sequential is done automatically if the parallel composition is well-formed.

All executable files can be divided into two groups, (1) generated files and (2) pre-written files. *pState* generates the file *charts.c*, which defines the behaviour of the application. Prewritten files *main.c, setupProcessor.c, Scheduler.h, actions.h* can be divided into two groups: target independent, and target dependent files shown in Figure 6.2. Target independent files are *main.c, and Scheduler.h*. The file *main.c* defines the entry of the application, and initializes variables, chart states, and the scheduler. Then it enters an infinite loop which processes input events and scheduled actions. The file *Scheduler.h* defines a data structure which holds timed events and defines functions to schedule and cancel timed events.

![Figure 6.2: Structure of C code for PIC 16F6xx](image)
The target dependent file `setupProcessor.c` contains routines for processor input/output initialization, timer initialization etc. The file `actions.c` specifies how external events will be processed. For instance, if a digital signal is connected to PORTA bit 0 of a PIC16F6xx micro-controller, and if the presence of a signal means high voltage, then that should be defined in `actions.c` as 

```
#define SIGNAL (RA0 == 1)
```

**PIC Assembly Code.** Assembly code is created by translating the intermediate code representation. Translation of if-else or case statements is straightforward, and is done by just converting them into assembly syntax. Translation of parallel statements needs extra processing since that statement has to be converted into sequential statements. It includes the normalization and sequentialization of multiple assignments. Most micro-controllers have an instruction which allow constants to be added immediately. In this approach generation of the code is delayed until the mode of an expression is known, a technique called *delayed code generation* [Wirth, 1996].

We are using CBLOCK 0x20 or CBLOCK 0x40 to allow the variables declared within the block to automatically increment to the next general register, starting from 0x20 or 0x40. Address 0x40 and beyond are used for constants associated with state names, while 0x20 to 0x39 are used for variables.

Names of variables in assembly code are generated as lowercase letter state names. Variables are either integers or boolean. The generated code consists of state and variable declarations, assignments and expressions, state transitions, macros, statements, and timed transitions. Scheduler, initialization and I/O actions are not generated from the specification, they are write-once code. In this way we have full control over the structure of the application, similar to the approach described in [IARSystems, 1999].

As stated in [Wirth, 1996], code generation depends not only on the symbols but
also on the values of their attributes. The generated code depends on the fact if
the value is held in a register or it is a known constant. If it is a constant, the
generated code will be smaller since the value does not need to be stored to working
register before the operation is performed. Where the value is stored and how it is to
be accessed is indicated by an attribute of non-terminals factor, term or expression
in the concrete syntax. Assembly code generation is implemented only for simple
expressions given by the syntax:

\[
\begin{align*}
expression & ::= \text{simpleExpr} \\
& \quad \{ (" = " \mid " \neq " \mid " \leq " \mid " \geq " \mid " < " \mid " > " ) \text{simpleExpr} \} \\
\text{simpleExpr} & ::= [" + " \mid " - " ] \text{term} \{" + " \mid " - " \} \\
\text{term} & ::= \text{factor} \\
\text{factor} & ::= \text{identifier} \mid \text{integer} \mid " (" \text{simpleExpression}" ) \\
\text{identifier} & ::= \text{letter}(\text{letter} \mid \text{digit}) \\
\text{integer} & ::= \text{digit}\{\text{digit}\}
\end{align*}
\]

In the implementation we have following attribute modes:

- **Reg** - special function registers i.e. PORTA, STATUS etc.,
- **Var** - general purpose registers
- **Const** - constant
- **AccW** - working register or accumulator

In addition to those we have special modes for expressions like **VarPlusConst**, **VarMinusConst** which indicates operations of addition or subtraction between factors of
type variable and constant. Once we know the mode of expression, optimized code
can be generated.
Arduino Code. Arduino is a C-derived programming language; most of its syntax is similar to that of C. Arduino program has at least two parts. The structure is shown in Listing 6.1. The setup function is the preparation, and the loop function is the execution. The setup function includes the declaration of variables. It is the first function to run in the program and is run only once. The loop function runs after calling the setup function. It is then executed continuously, allowing the program to change, respond to and control the Arduino board.

Custom functions in the Arduino language can be written to perform repetitive tasks and reduce clutter in a program. They are declared like functions in the C language with function return type, name, and parameters to be passed to the function. In our implementation we assume that no value is to be returned, so the event function type is `void`.

The Arduino code generated from the example in Figure 6.3 is in Listing 6.2. All states of the hierarchical structure are nested in the root state, which is declared as variable `root`. pCharts allow direct declaration only of integer and boolean variables.

![Diagram](image)

Figure 6.3: Simple operation on event $E$

Functions that are unique to the Arduino language and used to configure, read and write specific ports of micro-controller can be called as an action on some event.
Those actions have to be prewritten. If we need to setup some pin to be INPUT or OUTPUT, that is done by the `pinMode(pin,mode)` function; to read digital pin value, which can be HIGH or LOW, the function `digitalRead(pin)` is used, and to write to pin `digitalWrite(pin,value)` is used. Handling analog pin is done by `analogRead(pin,value)` and `analogWrite(pin,value)`. It reads and writes the value from a specified analog pin with a 10-bit resolution.

### 6.3 Key-fob Executable Code Generation

This example illustrates how executable code is generated form a pChart model, which part of the code is automatically generated and which part is prewritten. Finally we show how the execution time on each event is calculated.

A key-fob is a small handheld transmitter device. The transmitter circuit comprises (1) a user replaceable long-life battery serving as power source for the transmitter, (2) an electric circuit capable of generating a code modulated electric signal.
suited to drive an infrared (IR) light emitting diode (LED), (3) a momentary contact trigger switch actuated by a user through a thumb or index button placed in a convenient position on the plastic enclosure of the transmitter, (4) an IR LED (or several LEDs for longer range) functioning as the emitter. In Figure 6.4 the model of three operational modes of the device is shown. To save power, when it is not operational, it is in the Sleep state. On wup event (press of the button) it goes from Sleep into Active state. The time to switch from Sleep into Active should not be longer than 20 cycles, indicated by $\Delta 20$. Once in Active state, the system on event key1 goes into the Transmit state to emit the signal. On this transition, the code to be transmitted is assigned to the variable key. In our example, the code is the one-byte number 156. The time from the moment when the contact trigger switch generates event key1, to the moment when the Transmit state is reached, should not take more than 20 cycles. In Transmit, the system stays 20ms and then goes back into the Active state. If there is no key1 event for one second while the system is in Active, the timed transition on 1000ms will trigger. This creates the transition into the Sleep state and executes a SLEEP assembly instruction, which puts the micro-controller into low-power sleep mode.

There are two reactive events, key1 and wup, generated on the external action (button pressed), and two timed transitions, on 20ms and 1000ms, executed by the scheduler.

WCET is always associated to some target hardware; our target is PIC16F636 micro-controller.

Table 6.7 specifies how buttons are connected to the micro-controller. Conditions that create actions have to be prewritten; they are not generated from the pChart model. The button Key1, which creates event key1, is pressed if output 4 of PORTC goes to zero. The code of a simple action on which event key1 is called is in Listing 6.3.
Event \textit{key1} moves system from \textit{Active} state into \textit{Transmit} state. If the system is not in \textit{Active}, pressing Key1 button does not have an effect. The power switch button is connected to PORTA 0, an ultra low power wake-up pin, which has to be enabled by the appropriate configuration of the micro-controller’s Power Control Register PCON. The generic code for wake-up initialization can be found in [Microchip Technology Inc., 2007]. It should call the \textit{wup} event in the similar way the event \textit{key1} is called when the Key1 button is pressed in Listing 6.3.

The executable application has three assembly files and one include file with global variables. The main run-time loop is not generated from the specification; it is prewritten and is the same for all applications. It calls initialization routines, \textit{InitDevice} to

\begin{verbatim}
BCF STATUS, 0x05
BCF STATUS, 0x06
BTFSC PORTC, 0x4
RETURN
CALL key1
RETURN
\end{verbatim}

Table 6.7: Micro-controller PIC16F636 pins connection

<table>
<thead>
<tr>
<th>Port</th>
<th>Pin</th>
<th>I/O</th>
<th>H/L</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTA</td>
<td>0</td>
<td>Input</td>
<td>L</td>
<td>Switch not pressed</td>
</tr>
<tr>
<td>PORTA</td>
<td>0</td>
<td>Input</td>
<td>H</td>
<td>Switch pressed</td>
</tr>
<tr>
<td>PORTC</td>
<td>4</td>
<td>Input</td>
<td>L</td>
<td>Key1 pressed</td>
</tr>
<tr>
<td>PORTC</td>
<td>4</td>
<td>Input</td>
<td>H</td>
<td>Key1 not pressed</td>
</tr>
</tbody>
</table>

Figure 6.4: pCharts model of Key-fob
initialize the target hardware, *InitVariables* to initialize the chart variables, *InitChart* to set the initial states of the chart and schedule timed transitions if there are any at the initial state. After the initialization, the program stays in the loop that executes due-transitions or performs an action actuated by an input signal. The code of the main run-time loop is in Listing 6.4.

Routine *InitDevice* has to be prewritten, and it is part of *setupProcessor.asm* file. It is an elementary sequence of special function register configuration instructions.

Routines *InitVariables*, *InitChart*, *Event*, and *Action* are part of the generated code and stored in *chartsApp.asm*. Variables declared in the hierarchical state structure of pCharts are initialized in *InitVariables* and the initial states of the chart are set in *InitChart*. If there are timed transitions in the initial state, they have to be scheduled.

Scheduling a timed transition involves adding *transition name*, *due-time* or time from now until transition will be executed i.e. 100ms, and *priority* to an array of scheduled transitions. Timed transitions are identified by automatically generated unique names.

Events are sorted based on due time and priority. The transition that should be executed first is stored at the end in the scheduled transitions array. Events are identified by a unique name. The time to add a new transition to the array depends on the due-time, priority of transition, and the number of already scheduled timed
transitions. A transition with low priority and late due-time will take more time to be scheduled because already scheduled transition shifted to make space for the new one.

The algorithm of `schedule` is given in Listing 6.5. Executable code for this routine is written in assembly language.

If there are no scheduled transitions ($nt = 0$), a new transition is simply added to the first location. But, if there are scheduled transitions, the due-time of the new transition is compared with the time of already scheduled transitions. Next, we compare transition priorities. Priorities are considered only for transitions with the same due-time. During this process the interrupt tick is disabled. Once the transition is scheduled, due-time is decreased on each tick. In the current implementation, the tick is performed every 1ms by an interrupt service routine. When the due-time of a transition reaches zero, the associated transition is triggered. The prewritten code is in PIC assembly language so that we can count the number of cycles and calculate the WCET. An array of transition names points to the addresses of implementation routines of the transition with the parameter specified as time. Each timed transition has a unique name, so there is no ambiguity in call routine.
The `schedule` routine puts a timed transition into the data structure. The same transition can be removed from the structure by calling procedure `cancel`. The algorithm of the `cancel` routine is given in Listing 6.6.

The timed transition to be canceled is first located and then removed from the list of transitions. Remaining transitions are shifted left in the array of scheduled transitions. Similar as for scheduling, the interrupt tick is disabled during this process.

In the example in Figure 6.4 on the event `key1`, while the system is in the `Active` state, the timed transition on 20ms has to be scheduled, and the timed transition on 1000ms canceled. A snippet of the generated code for `key1` is shown in Listing 6.7.

We specify that `wup` and `key1` events should not take more than 20 cycles by $\Delta 20$. For the event `wup`, pState calculates 16 cycles. On this event, the timed transition on 1000ms is scheduled. The number of cycles needed to `schedule` a timed transition is
equal to the number of scheduled transitions multiplied by 141 and 52 added, which is 193 cycles. The WCET of \textit{wup} is

\[ WCET(wup) = 193 + 16 = 209 \text{ cycles.} \]  

(6.20)

For the \textit{key1} event, we need to calculate (1) \( c_1 \) - the number of cycles to scan PORTC 4, (2) \( c_2 \) - the number of cycles to cancel the timed transition on 1000ms, (3) \( c_3 \) - the number of cycles to schedule the timed transition on 20ms, (4) \( c_4 \) - the number of cycles to execute the statement \( key := 156 \). The values of \( c_1, c_2, c_3 \) are known from the prewritten code. From Listing 6.3 we can see that \( c_1=5 \) cycles. It takes \( c_2=141+52=193 \) cycles to \textit{schedule} timed transition. The number of cycles needed to \textit{cancel} transition is equal to the number of scheduled transitions multiplied by 35 plus 75 cycles, \( c_3 = 110 \). The number of cycles to execute the body of \textit{key1} event is
calculated by $pState$, $c_4 = 21$ cycles. WCET of the key1 is

$$WCET(\text{key1}) = c_1 + c_2 + c_3 + c_4 = 329 \text{ cycles.}$$ (6.21)

According to the model, the WCET for both events should be less or equal to 20 cycles. Therefore, the requirements are not satisfied. That is expected since scheduling and cancelling timed transition takes more than 100 cycles. In this case, even update of the target hardware (micro-controller) will not meet the specified requirements. The designer will get this information early in the design process, so the design requirements have to be relaxed, or where it is possible, functionality decreased (i.e., by excluding some timed transition).
Chapter 7

The Visual Tool pState

Our tool pState allows visual model specification according to pCharts syntax. From the model, it generates event-centric code. We describe briefly the tool and its main components. Full API documentation is generated by JavaDoc as HTML files from the Java source code.

7.1 Architecture

The architecture of the tool is shown in Figure 1.4. Main components are editor and code generators. The editor is used for the creation of chart visual objects on the screen that are modifiable by the user by dragging, resizing, etc. The code is generated from an intermediate representation of chart structure. An implementation class of the CodeGenerator interface generates particular code. To implement the generator of new executable code or input code for the model checker, the new class that implements code generator interface has to be created. That is a straight-forward process, the tool can be extended to use some other probabilistic model checkers like MRMC [Katoen et al., 2009], or a tool from the MoDeSt [Hartmanns, 2012] toolset.
In the verification of the conditions for an event to be executed, we use the SMT tool Yices, but input code for other tools can be created in a similar way, so it is possible to add another SMT tool. Currently, the WCET is calculated by counting a number of executable cycles, but it is possible to use third-party WCET analyzer as backend tools.

The tool is written in Java, using the JHotDraw framework [Randelshofer, 2012]. It has about 40000 lines of code. The main components of the code are tested on a number of test cases created by the JUnit testing framework.

7.2 pState Editor

The editor is designed using JHotDraw 7.6 framework. As a starting point we used frameworks from org.jhotdraw.samples package. In the current version we use JHD 7.6 version which is the latest release. Figure 7.1 is a typical view of the pState graphical interface that shows some of the TV set features represented as a pCharts diagram. Components, like states and transitions are added in drag-and-drop fashion using icons in the toolbar. It is possible to modify, move or delete the components. Operations for saving and reloading are also supported. The TV control activity is partitioned into two states, the Basic state Standby and the AND state Working. The initial state is Standby.

The design tool from the Figure 7.1 consist of drawing action tool, state figure, transition figure, initial state figure, probabilistic state figure, concurrent line draw, choice node, formulae, I/O declarations and text area figure. It is straightforward to add other tools by adding the tool to the button factory. We use the standard attribute bar with all selections from the JHotDraw framework. This bar is an example how new features like a colour of the figure can be easily added to the drawing editor.
7.3 Application Model, View, Figure

For the application we use the Mac OS Application Interface (OSX). View of the figures is handled by Java swing JPanel class. All figures are created either by extending the JHotDraw TextAreaFigure class or as a JHotDraw LabeledLineConnectionFigure.

Reading and Writing a Figure. JHD OutputFormat interface encapsulates a strategy for writing drawings to output streams. We can write a Drawing using a specific format into some output stream, a Uniform Resource Identifier (URI) or a Transferable interface for classes that can be used to provide data for a transfer operation. The drawings are stored in the Extensible Markup Language (XML) format.
7.4 Implementing Code Generation

To generate code from intermediate representation or EventDeclCode we use visitor design pattern. As stated in Gamma et al., 1995, the visitor pattern represents operations to be performed on elements of an object structure. The class diagram of the visitor pattern implementation in pCharts is shown on Figure 7.2.

The translation of the effect of a transition into an atomic statement in order to eliminate parallel composition is done by the transform operation. Next, by the translate operation we create executable code (C or assembly) or model of the chart in the form of a MDP or a PTA.
7.5 Summary

There are several other frameworks that can provide similar functionality to JHotDraw. Graphical Editing Framework (GEF) from the Eclipse project that provides end-user components related to graphical applications using Abstract Window Toolkit (AWT)/Swing [Eclipse, 2011a]. Piccolo [Piccolo, 2015] project provides a structured 2D graphics framework and relies on the Java2D API for its graphics rendering. A version of Piccolo built on .NET framework allows the creation of animated graphical applications on mobile devices. The main reason why $pState$ tool uses JHotDraw is compatibility to its predecessor iState, but in future development either GEF or Piccolo framework can be used as a viable alternative.
Chapter 8

Case Studies

On the series of examples, we demonstrate how system properties can be analyzed on models rather than on executable code. Qualitative properties, correctness with respect to invariants, and timing guarantees are verified together with quantitative properties, notably power consumption and reliability.

Assumptions In next examples, we assume that there is no internal clock drift and we do not take into account effects of aging of integrated circuits. To generate timed event, we use the timer with a clock resolution of 1ms (milisecond).

8.1 Fischer’s Protocol

Fischer’s Protocol is introduced in [Lamport, 1987], and our description follows [Vereijken, 1995]. Two processes, P1 and P2, are running in parallel. The shared variable x is initialized to 0. When P1 observes that x is 0, it writes the value 1 to x. After that, it waits for some time, and if x still has value 1, it is safe to enter the critical session. If we assume that if P2 also attempted to enter the critical section, it would have done so earlier by setting x to 2. Fischer’s protocol as a pCharts is in Figure 8.1.
Figure 8.1: Fischer’s protocol in pCharts

It illustrates timed transitions, nondeterministic timing, guarded transitions, global variables, and concurrent states.

Initially, $P_1$ is in state $Start_1$ and if shared variable $x$ is 0, process $P_1$ can go to state $Assignment_1$ at some time between 1 and 10ms. If $x \neq 0$ after 1ms (1ms..[x ≠ 0]), the state $Start_1$ is reentered and the internal clock is reset. It takes 1 to 2ms to assign value to $x$, and to enter state $Delay_1$. This is specified by 1ms..2ms/ $x := 1$. It takes additional 3 to 4ms to enter $CritSection_1$ if the condition $x = 1$ is satisfied, or to go back to $Start_1$ state if $x \neq 1$. In $CritSection_1$, process $P_1$ stays at least 3ms and then goes back to $Start_1$ by setting shared variable $x$ to 0. This is done by timed event 3ms/ $x := 0$. The minimum time to enter the critical section from $Delay_1$ is $t_d = 3$, and the maximum time to stay in $Assignment_1$ is $t_a = 2$, so $t_d > t_a$. This condition must be satisfied to exclude that critical sections 1 and 2 are entered at the same time. Informally we can prove this by following reasoning. Assume that $t_d > t_a$ is
not satisfied. Since two processes are running in parallel, it may happen that states
Assignment1 and Delay2 are entered at the same time. Since $t_d > t_a$ is not satisfied,
$P_2$ first enters CritSection2, then $P_1$ enters Delay1 and assigns $x := 1$. While $P_2$
is in CritSection2, $P_1$ enters CritSection1, which violates mutual exclusion. Mutual
exclusion is verified by the state invariant

$$in\,\text{CritSection1} \neq in\,\text{CritSection2}$$

attached to state FischersProtocol. The generated PRISM input code for Fischer’s
protocol is shown in Listing 8.1. The invariant is passed to PRISM as a query and
evaluates to true. Note that the (generated) invariant in the PRISM code constrains
the clock variables as required for probabilistic timed automata to ensure that timed
transitions will be taken; while it carries the same name, it is independent of chart
invariants.

Executable code that preserves verified property is generated according to the
process described in Chapter 6.

8.2 Alternating Bit Protocol

We model the behaviour of the alternating bit protocol for sending sequence of mes-
sages from a Sender to a Receiver [Bartlett et al., 1969]. The communication
is full-duplex, but messages are sent through a half-duplex channel that can hold only
one message at a time. Also, channels can “misbehave” in a way that can duplicate a
message or lose it. Each message is sent with a control bit that is either zero or one.
This information is used for retransmission if an error occurs. The goal of modelling
Listing 8.1: Generated code for Fischer’s protocol

```verbatim
pta

const FischersProtocol=0;
const Start1=0; const CritSection1=1; const Assignment1=2; const Delay1=3;
const CritSection2=0; const Assignment2=1; const Delay2=2; const Start2=3;

module fischersprotocol1
    root :[0..1] init FischersProtocol;
p1 :[0..3] init Start1;
p1clk : clock;
p2 :[0..3] init Start2;
p2clk : clock;
x : [0..2] init 0;

    invariant
        (p1=Start1=>p1clk<=10)
        & (p1=Assignment1=>p1clk<=2)
        & (p1=Delay1=>p1clk<=4)
        & (p2=Assignment2=>p2clk<=2)
        & (p2=Delay2=>p2clk<=4)
        & (p2=Start2=>p2clk<=10)

    endinvariant

    [] (p2=Delay2) & (x!=2) & (p2clk>=3) & (p2clk<=4) => (p2'=Start2) & (p2clk'=0);
    [] (p1=Start1) & (x=0) & (p1clk>=1) => (p1'=Start1) & (p1clk'=0);
    [] (p2=Delay2) & (x=2) & (p2clk>=3) & (p2clk<=4) => (p2'=CritSection2) & (p2clk'=0);
    [] (p2=CritSection2) & (p2clk=3) => (x'=0) & (p2'=CritSection2) & (p2clk'=0);
    [] (p2=Start2) & (x=0) & (p2clk>=1) & (p2clk<=10) => (p2'=Assignment2) & (p2clk'=0);
    [] (p1=Start1) & (x=0) & (p1clk>=1) & (p1clk<=10) => (p1'=Assignment1) & (p1clk'=0);
    [] (p1=Assignment1) & (p1clk>=1) & (p1clk<=2) => (x'=1) & (p1'=Delay1) & (p1clk'=0);
    [] (p1=Delay1) & (x=1) & (p1clk>=3) & (p1clk<=4) => (p1'=CritSection1) & (p1clk'=0);
    [] (p1=Assignment2) & (p1clk>=1) & (p1clk<=2) => (p1'=Start1) & (p1clk'=0);
    [] (p2=Start2) & (x=0) & (p2clk=1) => (p2'=Start2) & (p2clk'=0);
    [] (p2=Assignment2) & (p2clk>=1) & (p2clk<=2) => (x'=2) & (p2'=Delay2) & (p2clk'=0);
    [] (p1=CritSection1) & (p1clk=3) => (x'=0) & (p1'=Start1) & (p1clk'=0);

endmodule
```


is to verify that all messages will be sent successfully (because of effective retransmission), and to find out the impact of lost and retransmitted messages to the overall cost of the transmission.

The protocol is modelled as the parallel composition of devices Sender and Receiver, and channels ChannelOut and ChannelIn. Messages are sent from Sender to Receiver through ChannelOut and acknowledgments are sent from Receiver to Sender through ChannelIn. We distinguish two type of messages, zero and one, and associated two types of acknowledgments. In our model we assume that the sequence to be sent has 20 messages, alternatively tagged with zero and one. Initially the sender is
in $SendingMsg0$, and we initialize the number of message pairs to be sent by assigning 10 to variable $j$. On event $A0$ message $sendMessage0$ is broadcasted which causes a transition from state $IdleOut$ to state $TransMsg0$ in $ChannelOut$. The cost of the transition is indicated by $\text{energy}=2$ in the state $TransMsg0$.

The behaviour of the channel is modelled by probabilistic transitions. We assume the probability to receive a single message is 80%, and when the message is received, the event $recMsg0$ is broadcast. This is indicated by $\text{@0.8}/recMsg0$ on the alternative transition arrow from $TransMsg0$ to $IdleOut$. We assume that a message will be duplicated with a probability of 10%, represented by a transition back to $TransMsg0$. If the message is lost, which we assume to happen in 10% of cases, we go back to $IdleOut$.

Broadcasted message $recMsg0$ triggers a transition in $Receiver$ from $WaitingMsg0$ state to $SendingAck0$ state and consequently to $WaitingMsg1$ on event $B0/sendAck0$. Event $B0$ broadcasts acknowledgement $sendAck0$ on the message $sendMsg0$.

When $Receiver$ receives confirmation that message zero is successfully received, it goes to state $SendingMsg1$ to transmit message one. If acknowledgment on transmitted message does not come after timeout event $to0$ (we omit the exact time specification here), the message zero is retransmitted by $to0/sendMsg0$. This retransmission will increase the cost of the transition. In the case of duplicated messages, it may happen that while waiting for an acknowledgment of zero, the acknowledgement one is received. In that case, the message zero or $sendMsg0$ is retransmitted.

In the same way as message zero, the message one is sent. Everything is symmetrical except that in the model of $ChannelIn$ we assume that the cost associated with sending an acknowledgement, indicated in states $TransAck0$ and $TransAck1$, is lower than cost in states $TransMsg0$ and $TransMsg1$ because acknowledgments are usually much shorter.
To the state *Done* two queries are attached. In the specification of the property we use the state hierarchy in the translation to a PCTL formula for PRISM. By the first query $P_{\max}$ we can verify that the maximum probability to eventually reach state *Done* is 1. The verification of the minimal probability would result in 0, which reflects the theoretical possibility that messages can be always lost or duplicated. The second query $\text{energy}_{\min}$ returns the minimum expected energy to reach state *Done*, which results in 74.06 for the values specified on the model in Figure 8.2. The maximum theoretical expected energy to reach state *Done* is infinite, because of the probability to lose or duplicate messages is not zero.

<table>
<thead>
<tr>
<th>Output Channel</th>
<th>Input Channel</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suc.</td>
<td>Dup.</td>
<td>Lost</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.9</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>0.8</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>0.8</td>
<td>0.1</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>7</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>0.8</td>
</tr>
<tr>
<td>9</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>10</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 8.8: Cost vs. channel quality

From Table 8.8 we can see that in ideal conditions, when the probability of losing or duplicating message on both channels (input and output) is zero, the minimum expected cost is 60, since there are 20 transmissions, 10 messages for the cost of 2 and 10 acknowledgements for the cost of 1. Once we start increasing the probability of losing or duplicating messages, the overall cost starts to increase. As it can be expected, the contribution of the output channel to the total costs is higher than the contribution of the input channel. For instance, in runs 3 and 4, the *quality*...
(probability of lost, duplicated, correctly received messages) of output and input channel are swapped, and we can see that the minimal expected cost is lower when the quality of output channel is better than the quality of input channel. When the probability of successful transition decreases, the energy cost increases. Finally, when the probability of losing a message on at least one channel is 1, the message will never be transmitted, and the cost of transmission goes to infinity. The generated code for the probabilistic model checker is an Markov Decision Process (MDP) module, as there are no timed transitions. All parallel processes are flattened. Executable code is generated by selecting the Sender or Receiver states of the pCharts for the C code that represents the Receiver or Sender behaviour.

8.3 RFID Tag

In this case study, we show how the pCharts model in Figure 8.3 can be used to analyze properties of postal RFID tags [Paun, 2006] and to generate code for the embedded system. This model has concurrent states ElectronicTag and Environment. In the ElectronicTag state, the basic operation of an RFID device is specified. Initially, a tag is in the StandBy state and on the FieldOn event goes into the Receive state. Local event FieldOn is broadcasted by Environment on the transition from Off to On. The environment is initially in the Off state and in time between 58s and 62s, goes into the On state. During this transition, boolean variable field is set to true, which means that the a low frequency (LF) field is present in the environment.

Whenever an electronic tag is in the range of an LF field, it tries to read the unique field identification ID number. This process takes about 1s, and on average, it is recognized in 90% of cases. If the field ID is recognized, variable frec is set to true. This is shown by a probabilistic transition from state On to state FieldID. We
assume that the field will disappear according to an exponential distribution with an exponential scale of 5s, which is specified by the transition on $\exp(5s)$. This means that the transition from state $FieldID$ to $Off$ can take between 1s and 58s. Theoretically, an exponential transition may take forever, but we assume that the transition will be taken when the probability of the transition is bigger than 99.9999%. On the transition from $FieldID$ to $Off$, the variable $field$, which represents the presence of the field, is set to $false$.

![Diagram](image-url)

Figure 8.3: pCharts model of RFID tag excitation in pCharts

On the $ElectronicTag$ side, in the $Receive$ state, system field $ID$ is read. If $ID$ is not recognized ($frec = false$), but the field is still present ($field = true$), tag goes into $StandbyLF$ or low field standby state, in which stays for the next 10s. This is done to prevent multiple excitation by a pulsating field which cannot be recognized, and to save energy since the consumption in $StandbyLF$ is lower than in $Receive$. While in $StandbyLF$, event $FieldOn$ broadcasted by environment does not take any effect.
If the reading of LF field ID is good ($freq = true$), regardless of the status of the field flag, the tag goes into Transmit state. After transmission of a preprogrammed number of messages, the tag goes back into the initial state Standby if field is no longer present ($field = false$), or goes to StandbyLF if the field is still present ($field = true$). This depends how fast Environment goes from state FieldID to Off, and that is specified by an exponential time transition on $exp(5s)$.

To each state of ElectronicTag we assign costs in the form of power consumption. By our model, we can calculate the average consumption after a number of broadcasts of FieldOn events. To count broadcasts, we introduce a counter variable $i$ and increase it on every transition from Standby to Receive. Table 8.9 shows the minimum and maximum expected costs of the tag cycle (path from initial state Standby back to that state). The values are shown for three different probabilities (0.9, 0.8, and 0.1) of the message to be lost, and for the exponential distribution $exp(5s)$ of the field disappearing. The maximum consumption in one excitation is calculated using formula

$$cons.max F (i = 1)$$

specified in Off that sums the consumption in all ElectronicTag states (Receive, Transmit, Standby, StandbyLF) when Environment reaches Off after one tag excitation (process of going form Standby to Receive state).

<table>
<thead>
<tr>
<th>Run</th>
<th>Environment</th>
<th>Electronic Tag</th>
<th>Expected Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Succ.</td>
<td>Lost.</td>
<td>Exp</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>0.1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>0.2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.9</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 8.9: Min/max expected cost vs. channel quality
To validate the model, in addition to the counter $i$ we need to add temporary *End* states to which both *Environment* and *ElectronicTag* will go at the end of the test. Without these states PRISM reports a deadlock problem. The reason is the condition on temporal variable $i$ on *FieldOn* event. For executable code generation, both counter $i$ and state *End* are taken out from the pChart. The verification of formula (8.22) for run 1 returns 26.7037 and the time for the model checking is 135.26s, on an Intel(R) Core(TM)2 Duo CPU 2.00GHz laptop. The minimum expected consumption is 1.2394. The built model has 1136223 states and 1728571 transitions.

From the selected part of the pCharts model, we can generate executable code. In our example in Figure 8.3 if we select the *ElectronicTag* state (blue state), executable code for embedded system of micro-controller for the PIC16Fxx family can be generated.

### 8.4 Hubble Telescope

This example is based on the failure model for the Hubble telescope as presented in [Oldenkamp, 2007]. Six gyroscopes are used for navigation. The telescope is designed such that it can fully operate with only three gyroscopes. When less than three gyroscopes are operational, the telescope goes into sleep mode and waits for repair. As long as at least one gyroscope works, the telescope is operational, otherwise it will crash. The goal of modelling is to find out the probability that the system will operate without failure for a given period of time.

We build a formal probabilistic model of the system as a pCharts model with 13 states. In the model, we assume that each gyroscope has an average lifetime of 10 years. Since six gyroscopes are operational, and any one can fail, we can expect that the outgoing rate is $6 \cdot \frac{1}{10} = 0.6$, which means that there is 60% of chance that at least
one gyroscope will fail in 365 days. That is modelled as a probabilistic timed transition from state \textit{SixG} to state \textit{FiveG}. If five gyroscopes are correct, the probability to have a failure in one year is 0.5, which is modelled as the probabilistic transition from state \textit{FiveG} to state \textit{FourG}. When only two gyroscopes are active, the telescope needs to go into sleep mode. The probability that the telescope will go into sleep mode and the rescue operation starts in three days is 99.8%. In that case, in approximately 60 days, with probability 0.968% all failed gyroscopes will be fixed and the system goes into the initial state \textit{SixG}. If the rescue operation fails, the system goes into \textit{FailOne} and consecutively into the \textit{SleepOne} state. This means only one gyroscope is good, and the telescope is in sleep mode. The rescue operation is taken within two months (60 days) with 98.4% chance of success. While in state \textit{TwoG}, if the system fails to go to \textit{SleepTwo} state, it will continue to work with two gyroscopes until one fails in approximately 730 days. Then, it tries to go from \textit{OneG} into \textit{SleepOne} and to
start the rescue operation. The telescope can end up in the Crash state if it can
not go into sleep mode, or if the rescue operation is not successful. From pCharts
formula $P_{\text{max}} F < 3650d$ specified in the Crash state, $pState$ generates the formula

$$P_{\text{min}} = \neg[F < 3650(root = \text{Crash})]$$

on which PRISM calculates that the crash probability in 10 years (3650 days) is 0.01226%,
or the probability that the Hubble telescope will be operational in 10 years is 99.98774%.

8.5 Waspmote Sensor Power Consumption

In this example we show how pCharts can be used to model the power consumption of
a wireless sensor or end-unit device, and how to generate the framework for the device-
executable code. In a collaborative research effort [MacWater, 2015], we use sensors
to measure acidity or basicity (pH) of lake water, read the geographic position of the
sensor by GPS, and transmit data by a ZigBee wireless device [Libelium, 2014]. In
our experiment, we also use sensors to measure water conductivity, dissolved oxygen,
dissolved ions etc., which we leave out here for brevity. We show how to specify the
impact of the environment on the working device, and how to quantitatively verify that
impact. The model in Figure 8.5 has three concurrent states, Device, Environment,
and Test. The state Device has itself four concurrent composite states Board, pH,
ZigBee, and GPS. The state Device represents the behaviour of the Waspmote water
monitoring mote [Libelium, 2014]. State Environment represents the impact of the
environment on GPS communication. We add state Test to specify queries to be
quantitatively verified by the model checker.

Initially, the state Board is in DeepSleep, state pH is in pHOff, state ZigBee is
in ZigBeeOff, and state GPS is in GPSOff. Every 10 seconds, Board wakes up, and broadcasts the event pHOn. On this event pH goes from pHOff to pHSensorOn and executes the command pHTurnOn. This command is a prewritten external function. In the PRISM model it is ignored, but it is used for executable code generation.

In the state pHSensorOn, pH stays only 5ms, to measure water acidity, and then goes back to pHOff state. During this process it broadcasts pHRead and calls Turn-pHOff. On the event pHRead, Board goes from pHWarmUp to GpsWarmUp state, and broadcasts event GpsOn.

On the event GpsOn, state GPS goes from initial state GpsOff to GpsCheck and broadcasts event Connect. On this event, Environment moves from GpsEnvIdle to InitialDelay. The GPS module is used to read the position of the device. In normal operation, based on our measurements, is takes between 1.8s and 2.2s for the GPS to get connected. No connection is possible in less than 1.8s, in 50% of the time a connection is established between 1.8s and 2s, in 60% of time between

Figure 8.5: Wireless sensor power model in pCharts
2s and 2.1s. By 2.2s the connection is always established. This is modelled by a probabilistic transitions between $GpsEvnIdle$ and $Connecting4$. When the connection is established, Boolean variable $rec$ is set to $true$. In our model the GPS module tries to acquire a signal for 4.8s, or every 200ms for 24 times. Once the connection is established, the $GPS$ goes into $GetPosition$. Consumption in $GPS$ depends on how fast the connection is established, and that is modelled by probabilistic transitions in the $Environment$ state. From $GetPosition$, $GPS$ goes back into $GPSOff$, broadcasts $GpsRead$ event and executes $TurnGpsOff$. On broadcasted event $GpsRead$, $Board$ goes from $GpsWarmUp$ to $ZigBeeWarmUp$ and broadcasts $ZigBeeOn$.

The secure ZigBee networking protocol is used to transmit the collected data to the central station. We model power consumption in transmitting and receiving states, for data transmission and acknowledgement reception. Once this process is finished, $ZigBee$ goes back to $ZigBeeOff$, broadcasts $ZigBeeRead$ and executes $TurnZigBeeOff$. On the event $ZigBeeRead$, $Board$ goes from $ZigBeeWarmUp$ to $DeepSleep$, broadcasts the event $Done$, and executes $GoToDeepSleep$. The broadcasted event $Done$ moves $Test$ from $Testing$ to $Query$ state, where the queries

"$?P.min$, $?P.max$, $?\$cons.min" 

for $min$ and $max$ probabilities ($P$), and $min$ costs of the consumption ($\$cons$) are specified. They are used for the calculation of the probability that $Board$ will go from initial state $DeepSleep$ back to $DeepSleep$, and to calculate the consumption in one cycle. Current consumption is specified in $mA$ values according to the specification from the Wasp mote technical documentation.

The verification is done by the PRISM $Digital Clock$ engine. The created model
has 17221 states and 17232 transitions. The calculated minimal and maximal probabilities ($P_{\min}$ and $P_{\max}$) to reach Query are the same and 1, which means the test always terminates. There are no nondeterministic transitions, so the min and max probabilities are equal. The calculated expected minimal consumption ($cons_{\min}$) is 248581.78mAms, and it took 126.65s to do the calculation. The maximum time of one cycle is a simple summation of deep sleep time (10000ms) and the times to read pH (5ms), get the GPS position (2210ms), and send data by ZigBee (600ms) which is 12815ms. So, the average current consumption is $\frac{248581.78mAms}{12815ms} = 19.39mA$. Waspmote devices are usually powered by the battery of 6600mAh, so according to our calculation the battery can last for approximately 340 hours, or 14.1 days. Thus, we are able to predict automatically the battery life from the model. 

All properties are verified on Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 12.0GB of RAM and on 64-bit Operating System. We assume that working conditions do not change and that the battery discharge curve is linear. In the reality that is not a case, so we expect that results of empirical measurements may be slightly different than calculated.

The executable code is generated by first selecting Device (blue state). A snippet of the generated code is shown in Listing 8.2. At the initialization, the timed event $exactly1$ is scheduled in 10000ms. When it runs, it broadcasts the event pHOn and moves Board into pHWarmUp state. Broadcasted event pHOn moves state pH from pHOff to the pHSensorOn, sets a new timed event $exactly3$ to be run in 5ms, and calls input-output action PhTurnOn. This action is hardware dependent and it is part of a prewritten input-output library.
Listing 8.2: Sensor’s code snippet

```c
/* Variables */
enum board_status {DeepSleep, pHWarmUp, ZigBeeWarmUp, GpsWarmUp} board;
enum zigbee_status {ZigBeeOff, Receiving, Transmitting} zigbee;
enum ph_status {pHSensorOn, pHOff} ph;
enum gps_status {Continue, GetPosition, GpsOff, Transit, GpsCheck} gps;

int i;
#define MAX_TIME_EVENTS 8
void exactly4 (long t);
void exactly5 (long t);
void exactly6 (long t);
void exactly7 (long t);
void exactly0 (long t);
void exactly1 (long t);
void exactly2 (long t);
void exactly3 (long t);

int main(void){

    pthread_t threadRun;  // thread variable
    /* Initialize data to pass to thread */
    eventdata.n = 0;
    eventdata.tr = false;
    eventdata.ir = true;
    /* Create schedule run thread */
    pthread_create (&threadRun, NULL, (void *) &run, (void *) &eventdata);
    /* Initialization */
    board = DeepSleep;
    schedule( &exactly0, 10000, 1);
    zigbee = ZigBeeOff;
    ph = pHOff;
    gps = GpsOff;
    i=0;

    return 0;
}

void exactly0(long t){
    pHOn();
    board = pHWarmUp;
}

void pHOn(long t){
    if ((ph == pHOff)) {
        pHTurnOn();
        ph = pHSensorOn;
        schedule( &exactly3, 5, 1);
    }
}
...
```
Chapter 9

Analysis of the DASH7 RFID-standard

We illustrate the holistic approach by a case study in which we analyze the tag collection and collision arbitration of the DASH7 open standard protocol. In this protocol, message collision is allowed to some extent. First, we create a model of tag collection to calculate the collision probability and then we use the calculated collision probability to estimate the average tag power consumption. Finally, we show how the code for a tag micro-controller can be generated directly from the system model.

9.1 DASH7 Protocol

We present the first formal analysis of DASH7 Mode 2 [DASH7-Alliance 2013], the radio frequency identification (RFID) protocol. DASH7 is a non-profit industry consortium to promote interoperability among DASH7 compliant devices. In a similar way as the WiFi alliance follows IEEE 802.11 protocol, the DASH7 alliance follows the ISO 18000-7 standard for wireless sensor networking. ISO/IEC FCD 18000-7.2 is
an open standard for the license-free 433MHz industrial, scientific and medical (ISM) band. DASH7 technology is used in a wide range of applications from automotive industry to location based services (smart-cards, key-fobs, tickets).

Two main components of radio-frequency identification systems are *tags* and *readers* or *interrogators*. Tags are attached to the objects to be identified. Readers send a signal to tags and read their response. Tag to tag communication in traditional RFID systems is not supported, but DASH7 technology supports tag to tag communication as well as a long range communication to up to 10Km. It is another technology for *mesh* sensor networking. The technology of DASH7 is so-called BLAST (Bursty Light ASynchronous Transitive) networking technology.

*Bursty* means that data transfer is abrupt, and does not include content isochronous (video, audio) forms of data. *Light* indicates that packet sizes are limited to 256 bytes and multiple consecutive packets are usually not transmitted. *Asynchronous* method of communication is by command-response which does not require periodic network ”hand-shaking” or synchronization between devices. *Transitive* means that a DASH7 is system of mobile devices. In this study we consider modelling and analysis of a variety of QoS properties for a complex, real world RFID network protocol and to our knowledge, this is the first formal analysis of DASH7 technology.

The ISO/IEC 18000-7.2 [DASH7-Alliance, 2013] standard provides an air interface implementation for wireless, non-contact information system equipment for *item management* applications. We study a *system* with active tags, i.e. tags with an own source of energy (battery). Each tag has an unique serial number and other data. It is intended for attachment to a managed item. An interrogator is a device that communicates to tags in its RF communication range. The interrogator controls the master-slave protocol, reads information from the tag, directs the tag to store data,
ensures message delivery and validity. We present the method by which the interrogator identify and communicate with one or more tags present in the operating field of the interrogator over a common radio frequency channel. Tags do not transmit unless commanded to do so by the interrogator, and an interrogator can communicate with tags individually, or with the tag population as a whole.

For three tags and five time slots as in the system shown in Section 1.3, Figure 1.6, in the first communication period, we assume that tags #1 and #3 transmit in the same time slot, so there will be a collision. In the acknowledgement period there is an acknowledgement only for the message of tag #2. In the second communication period tags #1 and #3 will retransmit the message, and we assume that tag #1 transmits in the time slot 1, and tag #3 transmits in time slot 4, so there is no collision, and in the acknowledge period there are two acknowledge messages.

**Collision Model.** We can calculate the collision probability by calculating the number of possible transmissions without collision and divide it by the total number of possible transmissions. For \( n \) transmitting tags, the first tag can transmit in any of the \( m \) time slots, the second tag should transmit in any of the \( m - 1 \) slots to avoid collision and so on. The number of transmissions without collision is:

\[
NC = m \cdot (m - 1) \cdot \ldots \cdot (m - n + 1) \quad (9.23)
\]

while the number of all possible transitions is:

\[
AT = m \cdot m \cdot \ldots \cdot m = m^n \quad (9.24)
\]
The probability that a collision will happen is simply:

\[ 1 - \frac{N_C}{AT} \]  

(9.25)

We assume a model of three tags \((n = 3)\) and five time slots \((m = 5)\). The number of possible transmissions without collisions is \(N_C = 5 \cdot 4 \cdot 3 = 60\), and the number of possible transmissions is \(AT = 5 \cdot 5 \cdot 5 = 125\). The probability of at least one collision according to (9.25) is 0.52.

The collision model represented by pCharts is shown in Figure 9.1. The number of tag is three \((N = 3)\), and the number of time slots is five \((M = 5)\). In state Collision, by the formula "? P.min" we query the collision probability, or probability to go to Collision state, which is 0.52. To calculate the collision probability for a different number of time slots or a different number of tags, all we have to do is to assign new numbers to \(M\) or \(N\) in the Tag state declaration.
Listing 9.1: Generated code for the collision model, three tags, five time slots

```plaintext
mdp
const M=5;
const N=3;
const Tag=0;
const Collision=0; const NextTS=1; const TS=2;

module test2
    root :[0..1] init Tag;
tag :[0..2] init TS;
c :[0.. M] init M;

    [t2] (tag=NextTS)&(c>(M−N)) −> (c’=(c−1))&(tag’=TS);
    [t1] (tag=TS)&(c>(M−N)) −> (1−(c/M)):(tag’=Collision) + (c/M):(tag’=NextTS);
endmodule
```

Figure 9.2: Collision probability for tags $N = [2, 3]$ and time slots $M = [3..9]$.

Translation of the model into input code for the model checker is shown in Listing 9.1. The collision model in Figure 9.1 is without timed transition and the generated input code for model checker is MDP.

**Power Consumption Analysis.** However, in the power consumption model, we need to know how long tags stay in states and the current consumption in those states. The model of power consumption is shown on Figure 9.3. From this model, `pState` generates a PTA model as input code for the probabilistic model checker.

In the PTA model, we can query the average power consumption in one collection
Figure 9.3: Power consumption of collection period

period (CP), taking into account the collision probability calculated on the collision model. Current consumption for a typical active tag during transmission is $9.2\,mA$, in receiving mode $0.2\,mA$, and in standby $0.0024\,mA$ [Paun, 2006]. In the construction of the transitions we use collision probability data from Figure 9.2. In the case of three tags and five time slots, the collision probability is $0.52$, or $52\%$, so the transition from state $Tx$ to $Next$ is probabilistic with $52\%$ probability. That means that if a collision happens, the tag needs another collection period to perform the operation. In the case of collision, the probability that all three tags select the same time slots is $4\%$ which is modelled by a probabilistic transition from $Tx$ to $ThreeCollisions$ state. If there is collision of two tags, in the next collection period, on five time slots, two only tags are transmitting. According to Figure 9.2, the collision probability is $20\%$, and that is represented by a probabilistic transition from $TwoTags$ to $Next$ state. The maximum number of collection periods in the model is three.

Properties are verified over the formula $?\cons\max$ specified in the state $End$. The calculated value is $216.69\,m\text{Ams}$, with an electrical charge of $0.216\,m\text{As}$. We can
Listing 9.2: Generated PTA code for the power consumption model, three tags, five time slots

pta

const Next=0; const Rx=1; const Tx=2; const TwoTags=3; const End=4; const ThreeCollisions=5;

module powercons

tagconsum:[0..5] init Rx; tagconsumclk : clock;

i :[0..3] init 0;

invariant

(tagconsum=Next=>tagconsumclk<=1)
& (tagconsum=Rx=>tagconsumclk<=68)
& (tagconsum=Tx=>tagconsumclk<=15)
& (tagconsum=TwoTags=>tagconsumclk<=1)
& (tagconsum=ThreeCollisions=>tagconsumclk<=1)

endinvariant

[] (tagconsum=ThreeCollisions)&(tagconsumclk=1) --> 0.52:(tagconsum'=Next)&(tagconsumclk'=0) + 0.48:(tagconsum'=End)&(tagconsumclk'=0);
[] (tagconsum=Tx)&(i=0)&(tagconsumclk=15) --> 0.52:(tagconsum'=Next)&(tagconsumclk'=0) + 0.48:(tagconsum'=End)&(tagconsumclk'=0);
[] (tagconsum=Next)&(tagconsumclk>=0)&(tagconsumclk<=1) --> (tagconsum'=(i<2)?Rx:End)&(tagconsumclk'=0);
[] (tagconsum=Tx)&(i!=0)&(tagconsumclk=15) --> 0.96:(tagconsum'=TwoTags)&(tagconsumclk'=0) + 0.04:(tagconsum'=ThreeCollisions)&(tagconsumclk'=0);
[] (tagconsum=Rx)&(i<3)&(tagconsumclk=68) --> (i=(i+1))&(tagconsum'=Tx)&(tagconsumclk'=0);
[] (tagconsum=TwoTags)&(tagconsumclk=1) --> 0.8:(tagconsum'=End)&(tagconsumclk'=0) + 0.2:(tagconsum'=Next)&(tagconsumclk'=0);

endmodule

rewards "cons"

(tagconsum=Next): 0.2;
(tagconsum=Rx): 0.5;
(tagconsum=Tx): 9.2;
(tagconsum=TwoTags): 0.2;
(tagconsum=ThreeCollisions): 0.2;

endrewards
also calculate the probability of not receiving all tags in three collection periods by the formula \( P_{\text{max}} F (i = 3) \). It is 0.046, or 4.6%, so in more than 95% of the time all tags are read in three collection periods.

**Executable Code Framework Generation.** To generate code for a particular target we need to know the detailed hardware design, e.g. which micro-controller is used, how pins are connected etc.

![Figure 9.4: Tag model](image)

Figure 9.4 gives the tag operation model. It has two parallel processes \( Tag \), which represent the tag operation, and \( Mode \) which represents the tag transition from sleeping to working mode and back. Initially, \( Mode \) is in the \( Sleep \) state, in which periodically, or every 1ms, the presence of the wakeup signal is checked. If there is no signal \( Mode \) goes back to sleep and repeats the process after 1ms. If there is a wakeup signal, the indicator \( field \) is set to \( true \), and the system goes first into \( Field \) and immediately to \( FieldON \). From that state it goes to \( Work \), and broadcasts event \( WakeUp \). The broadcasted event moves \( Tag \) from \( Start \) into \( Preamble \) state. On that transition,
the WAKEUP procedure is called. It has to recognize the WP preamble, Figure 1.6, and has to be executed in 2.45 to 4.8 seconds. Next, Tag goes into CP in which it receives commands from the interrogator, and goes into listen period LP, in which the tag transmits its message in a randomly selected time slot. In the Ack state, Tag waits for confirmation or acknowledgement message. If the received command informs the tag that communication is done, it goes to Finished and then back to Start. On the transition from Finished to Start, local event GoToSleep is broadcasted. This forces the parallel process Mode to go from Work to Sleep mode. Another way to go from Work to Sleep is on a timeout which is 30 seconds according to the protocol specification.

Generated files are target independent, while pre-written files are target dependent as described in Section 6.2. In this example, target dependent file setupProcessor.c contains routines for processor input/output initialization, timer initialization, etc. The file actions.c specifies how external events will be processed. For instance, if the wakeup signal is connected to PORTA bit 2 of the micro-controller, and if the presence of field means low voltage on the pin, then in the actions.c this has to be defined as "#define WUP SIGNAL (RA2 == 0)".

From the specification we generate the framework for tag application according to DASH7 protocol. For full implementation it is necessary to implement following procedures (1) WAKEUP to detect a low frequency wake up signal, (2) RECEIVECP to receive broadcast and point-to-point commands, (3) LISTENPERIOD to send a packet message which contains unique tag identification number to interrogator in the selected time slot, (4) ACKPERIOD to receive acknowledge from an interrogator. Those four procedures should be developed according to International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) 18000-7 standard [ISO/IEC 2008] that satisfy timing requirements from the general
framework model. Each transition can be transferred automatically into assembly
code and WCET can be calculated in terms of processor cycles.

Listing 9.3: Generated code from tag model

```c
#include <pic.h>
#include <stdio.h>
#include <stdlib.h>
#include <htc.h>
#include "scheduler.h"
#include "actions.h"

enum tag_status {Finished, LP, Preamble, Ack, CP, Start} tag;
enum mode_status {Work, FieldON, Field, Sleep} mode;

// Global Variables
bit done;
bit field;

// Timed events
void between8 (unsigned int);
void exactly4 (unsigned int);
void exactly5 (unsigned int);
void between0 (unsigned int);
void between2 (unsigned int);
void exactly1 (unsigned int);
void between7 (unsigned int);
void after6 (unsigned int);
void exactly3 (unsigned int);
```
/* Initialize variables */

void InitVariables(void) {
    done = 0;
    field = 0;
    // Init variables related to timed event scheduler
    n = 0;
    tr = 0; // false
    ir = 1; // true
}

// Execute timed event
void Event(void) {
    // Timed event
    if (run) {
        run = 0;
        timedEvent[event] (tm[event]);
    }
}

// Input actions
void Action(void) {
}

// Chart initialization
void InitChart(void) {
    tag = Start;
    done = 0;
    mode = Sleep;
    schedule(between8, 1, 1);
}

void between8(unsigned int t) {
    if (WUP SIGNAL) {
        field = 1;
    }
mode = Field;
schedule(between7, 0, 1);
}

void exactly4(unsigned int t){
tag = LP;
schedule(exactly1, 57, 1);
LISTENPERIOD();
}

void exactly5(unsigned int t){
mode = Sleep;
schedule(between8, 1, 1);
}

void GoToSleep(){
if ((mode == Work)) {
    mode = Sleep;
    schedule(between8, 1, 1);
cancel(exactly5);
}
}

void WakeUp(){
if ((tag == Start)) {
    WAKEUP();
tag = Preamble;
schedule(between2, 2450, 1);
}
}

void between0(unsigned int t){
GoToSleep();
tag = Start;
done = 0;
}
void between2(unsigned int t) {
    RECEIVECP();
    tag = CP;
    schedule(exactly4, 5, 1);
}

void exactly1(unsigned int t) {
    ACKPERIOD();
    tag = Ack;
    schedule(exactly3, 20, 1);
}

void between7(unsigned int t) {
    if (!(field)) {
        mode = Sleep;
        schedule(between8, 1, 1);
    }
    else {
        if (field) {
            mode = FieldON;
            schedule(after6, 0, 1);
        }
    }
}

void after6(unsigned int t) {
    WakeUp();
    mode = Work;
    schedule(exactly5, 30000, 1);
}

void exactly3(unsigned int t) {
    if (done) {
        tag = Finished;
        schedule(between0, 0, 1);
    }
Test. The generated framework of the code is tested on a PICDEM™ board with a PIC16F636 micro-controller. The code is compiled by the XC8 v1.35 compiler of the MPLAB X IDE v1.85. The micro-controller is programmed by MPLAB IPE v3.05.

To test timed event execution and how the system in Figure 9.4 goes from state to state, six LEDs are connected to the PORTC on RC0-RC5 output pins through 470Ω resistors. One push button is connected to RA2 input pin over a 10KΩ resistor [Microchip Technology Inc., 2011].

Initially, the device is in state Sleep. It wakes up every 1ms to scan if there is LF field. This mode is indicated by the first green LED. The external LF field is simulated by the push button event. That is indicated by turning the red LED on (third from left to right in Figure 9.5). In some states the system stays shortly, that

<table>
<thead>
<tr>
<th>Port</th>
<th>Pin</th>
<th>I/0</th>
<th>H/L</th>
<th>Meaning</th>
<th>Connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTC</td>
<td>0</td>
<td>Output</td>
<td>H</td>
<td>Tag in LP</td>
<td>Red LED</td>
</tr>
<tr>
<td>PORTC</td>
<td>1</td>
<td>Output</td>
<td>H</td>
<td>Tag in Ack</td>
<td>Red LED</td>
</tr>
<tr>
<td>PORTC</td>
<td>2</td>
<td>Output</td>
<td>H</td>
<td>LF field detected</td>
<td>Red LED</td>
</tr>
<tr>
<td>PORTC</td>
<td>3</td>
<td>Output</td>
<td>H</td>
<td>Scan LF field</td>
<td>Green LED</td>
</tr>
<tr>
<td>PORTC</td>
<td>4</td>
<td>Output</td>
<td>H</td>
<td>Mode Work</td>
<td>Green LED</td>
</tr>
<tr>
<td>PORTC</td>
<td>5</td>
<td>Output</td>
<td>H</td>
<td>Tag in CP</td>
<td>Green LED</td>
</tr>
<tr>
<td>PORTA</td>
<td>2</td>
<td>Input</td>
<td>H</td>
<td>Wakeup</td>
<td>Push button</td>
</tr>
</tbody>
</table>

Table 9.10: Micro-controller PIC16F636 test pins connection
is indicated by the associated LED blinking. In the end, both Tag and Mode go into initial states and all LEDs are turned off.

![Testing generated executable code framework](image)

**Figure 9.5: Testing generated executable code framework**

Based on our test the timing is according to the timed transitions specified on the model in Figure 9.4. If a valid field is detected, the whole cycle from Sleep back to Sleep is between 2533 and 2536ms, otherwise the system goes back into Sleep on a timeout event, which is 30s. To create an active tag transponder, the appropriate hardware LF receiver and RF transmitter/receiver has to be added. The procedure that reads the signal from LF field WAKEUP should take at least 2400ms, the function RECEIVECP that starts the collection period by synchronization takes 5ms, LISTENPERIOD is procedure that takes 57ms in which the RF signal from the interrogator is received, and ACKPERIOD takes 20ms as shown in Figure 1.6. The test shows how the generated framework runs on the target processor.
9.2 Discussion of Holistic Development

We show in this example how three steps, system property analysis, device property analysis, and executable code generation of an embedded software development process can be performed by a single tool. All those steps can be done by other tools separately, but that approach decreases engineering efficiency. That process may be more expensive, especially if there is need for redesign or optimization. Documentation of the project becomes more difficult, too.

**System Property Analysis.** It is possible to analyze system properties like a collision probability shown in this example by some spreadsheet tools, or some mathematical software. That approach is manual and may not be well documented as an integral part of the development process.

**Device Property Analysis.** Properties of the system with probabilistic transitions can be analyzed manually by somebody who understands in depth probability theory, with a help of commercial tools like Matlab, or by probabilistic model checkers.

**Executable Code Generation and WCET Analysis.** There are many tools capable of generating executable code, but we believe that code generation has to be integrated with probabilistic model checking, such that the generated code preserves the verified qualitative and quantitative properties.

**Documentation.** The documentation that can be written in parallel with the system design so that design decisions can be described when they are made. From the beginning to the end of software engineering process, looking at the big picture increases engineering efficiency.
Chapter 10

Conclusions

The development of pState was motivated by the analysis of a specific kind of RFID tags [Nokovic and Sekerinski, 2010]. Our overall goal is to support a holistic approach in which qualitative properties, notably structural well-formedness, correctness with respect to invariants, and timing guarantees, can be verified together with quantitative properties, notably resource consumption, reliability, and performance. These properties cannot be analyzed by considering exclusively the computerized part; rather, its environment has to be considered to certain extent. In this thesis, we have demonstrated how that kind of a tool can be created.

10.1 Thesis Summary

We presented the operation structure of event-centric transitions and the formal syntax of pCharts, a variation of statecharts extended with probabilistic transitions, timed transitions, costs/rewards, and state invariants.

In Chapter 5 we described and implemented a framework of model based WCET analysis for reactive systems. From a hierarchical pCharts representation, executable
code in a low-level language can be generated. On the generated code, WCET of an event can be calculated by counting the number of assembly instruction execution cycles. The precise WCET determination on complex architecture is a challenging problem, but the determination of WCET on simple 8 and 16-bit micro-controller is easier since features like multi-stage pipelines and cashes are not present. We showed how in the earlier stage of software design, during the specification process, WCET analysis can be performed on basic blocks. Also, we showed how to take into account invariants and the probabilistic nature of transitions. In our specification, we do not have general loops, so the calculation of WCET is straight-forward. The modular design of our tool pState allows us to integrate easily probabilistic model checker like PRISM and SMT solvers like Yices as supporting tools. In Chapter 6 we described the process of executable code generation and the structure of the generated code.

Chapter 7 presented how the tool is created using the jHotDraw framework, which provides drawing tools as well as facilities for saving and printing figures.

In Chapter 8 we showed some case studies to illustrate how we can specify the impact of an environment. pCharts model can be used to optimize device hardware (i.e. power consumption) and software design for particular working environment. The thesis is concluded with the analysis of the DASH7 RFID-standard.

10.2 Future Work

The following topics would provide useful extensions to the main contributions made in this study:

- We use PRISM as backend model checker, but other probabilistic model checkers like Fortuna or MRMC can be easily added. The general problem of model checkers is the state-space explosion. One of the ways to handle this problem
is to use approximate or statistical model checkers and estimate the correctness of a design. Some of the probabilistic statistical model checkers like APMC or Ymer can be also added as backend verification tools.

- Assembly code generation is implemented only for a limited number of arithmetical operation (summation and substraction) to demonstrate the technique of delayed code generation. Full implementation of assembly code generation is part of future work.

- A set of states that have the same structure can be specified by indexing the state by a parameter. Implementation of parametrized states may simplify model representations.

- Processor behaviour analysis gathers information on the processor components that influence the execution times, such as memory, caches, and pipelines. The inclusion of processor behaviour analysis will make WCET analysis possible on a wider range of microprocessors.

- pState is developed as the standalone application with supporting tools. The full integration of a probabilistic model checker and an SMT solver, and the development as Eclipse plugin would make it easier to install and use.
Appendix A

Basic Definitions
A.1 Properties Specification

Temporal logic for the formal specification of quantitative properties of PTAs is based on the probabilistic computational tree logic PCTL [Baier and Katoen, 2008; Bianco and Alfaro, 1995]. The syntax of PCTL is given by the following grammar:

\[
\begin{align*}
\phi & ::= \text{true} \mid a \mid \chi \mid \phi \land \phi \mid \neg \phi \mid P_{\triangleleft p}[\psi] \mid R_{\triangleright q}[\rho] \\
\psi & ::= \phi U^{\leq k} \phi \mid \phi U \phi \\
\rho & ::= I = k \mid C \leq k \mid F \phi
\end{align*}
\]

where \( \phi \) is a state formula, \( \psi \) path formula, \( a \in AP \) is an atomic proposition, \( \chi \in CC(\mathcal{X}) \) is a clock of constraint, \( \triangleleft \in \{\leq, <, \geq, >\} \), \( p \in \mathbb{Q} \cap [0,1] \), \( q \in \mathbb{Q}_{\geq 0} \), \( \rho \) is reward function on reward structure \( r \) and \( k \in \mathbb{N} \) [Norman et al., 2013]. The set of clock constraints over \( \mathcal{X} \), denoted \( CC(\mathcal{X}) \), is defined by the syntax:

\[
\chi ::= \text{true} \mid x \leq d \mid c \leq x \mid x + c \leq y + d \mid \neg \chi \mid \chi \land \chi
\]

where \( x, y \in \mathcal{X} \), and \( c, d \in \mathbb{N} \). A clock valuation \( v \) satisfies a clock constraint \( \chi \), if \( \chi \) resolves to true when substituting each occurrence of clock \( x \) with \( v(x) \). The set of valuations satisfying a clock constraint is called a zone. A reward structure is defined using a pair \( r = (r_L, r_{Act}) \), where \( r_L : L \rightarrow \mathbb{R}_{\geq 0} \) is a function assigning to each location the rate at which rewards are accumulated as time passes in that location and \( r_{Act} : L \times Act \rightarrow \mathbb{R}_{\geq 0} \) is a function assigning the reward of executing each action in each location. This logic extends propositional logic with a probabilistic operator \( (P) \) and a reward operator \( (R) \). A property of the form \( P_{\triangleleft p}[\psi] \) states that the probability of path formula \( \psi \) being true satisfies the bound \( \triangleleft p \). A property of the form \( R_{\triangleright q}[\rho] \) means that the expected value of reward function \( \rho \) on reward
structure $r$ meets the bound $\triangleright q$. As can be seen from the grammar above, formulae in the logic are always state formulae since path formulae only occur inside the $P$ operator. For some state $s$ and state formula $\phi$, we write $s \models \phi$ to denote that $\phi$ is satisfied in $s$. The reward operator $I^=k$ refers to the reward of the current state at time instant $k$, $C^\leq k$, to the total reward accumulated up until time point $k$, and $F \phi$ to the total reward accumulated until a state satisfying $\phi$ is reached.
A.2 MDP

Markov decision processes are a variant of Markov chains that permit both probabilistic and nondeterministic choices. This presentation of MDP follows [Fruth, 2011; Baier and Katoen, 2008; Norman et al., 2013].

Definition 1 A labelled Markov decision process is a tuple \( M = (S, \bar{s}, A, p, l, r) \) where

- \( S \) is countable nonempty set of states
- \( \bar{s} \) is the set of initial states
- \( A \) is the finite set of actions
- \( p : S \times A \to \text{Dist}(S) \) is the transition probability function
- \( l : S \to 2^{AP} \) is the labelling function
- \( r : S \times A \times S \to R \) is the reward function

and \( AP \) is a set of atomic propositions. We assume that \( S, A, p, l \) and \( r \) do not vary over time, and that \( S \) and \( A \) are discrete.

Example. The pChart of a simple MDP and the generated PRISM code are shown in Figure A.1 and Figure A.1. There are two transitions on \textit{wakeup} from \( S_0 \), the initial state, the choice between them being nondeterministic. One of the transitions

![Figure A.1: State-transition diagram of the MDP model](image-url)
Listing A.1: MDP PRISM code generated by \textit{pState}

```prism
mdp
const S0=0; const S1=1; const S2=2; const S3=3;

module mdpexample
    root :[0..3] init S0;
    [send] (root=S1) -> (root'=S2);
    [wakeup] (root=S0) -> 0.3:(root'=S0) + 0.7:(root'=S1);
    [wakeup] (root=S0) -> (root'=S1);
    [recv] (root=S2) -> 0.1:(root'=S1) + 0.9:(root'=S3);
endmodule

rewards "r"
    (root=S0): 0.1;
    (root=S1): 3;
    (root=S2): 2;
    (root=S3): 0;
endrewards
```

is probabilistic, in which with 70\% probability state $S1$ is reached, and with 30\% probability the system stays in the initial state. The other transition from $S0$ to $S1$ is deterministic, where on event \textit{wakeup}, state $S1$ is always reached. The transition on event \textit{send} is deterministic and the transition on event \textit{recv} is probabilistic. Rewards are assigned to the states by $r = e$, where and $e \geq 0$ is a real expression.

In $S3$ we specify two queries. The query $?sr.max$ returns maximum cost to reach $S3$ and the query $?P.max$ returns the maximum probability to reach state $S3$. Those two properties are translated into PCTL formulas $R''r''max = ?[F(root = S3)]$ and $Pmin = ?[F(root = S3)]$ respectively. The calculated maximum costs to reach $S3$ is 5.6984, and the maximum probability to reach $S3$ is 0.9999, that is 1. The error comes from floating point rounding of the model checker.
A.3 PTA

Timed automata (TA) provide a natural way for expressing timing delays of real-time systems [Alur and Dill, 1994]. On a TA, we can prove the correctness of real-time systems using trace semantics originally proposed in a model for communicating sequential processes (CSP) [Hoare, 1978]. Probabilistic timed automata (PTA) are an extension of TA used for formal modelling and analysis capabilities for systems with probabilistic, nondeterministic and real-time characteristics [Baier and Katoen, 2008]. PTA augmented with quantitative information in the form of costs or reward are called priced probabilistic timed automata [Kwiatkowska et al., 2009]. On a PTA model, two main classes of properties can be analyzed, the minimum/maximum probability of reaching a target, possibly within a time bound, and the minimum/maximum expected reward accumulated until a target is reached, using quantitative abstraction refinement and statistical model checking verification methods [Forejt et al., 2011].

Definition 2 A probabilistic timed automaton (PTA) is a tuple

\[ P = (S, \bar{s}, X, A, \text{inv}, \text{enab}, \text{prob}, l) \]

where

- \( S \) is countable nonempty set of states;
- \( \bar{s} \) is the set of initial states;
- \( X \) is a finite set of clocks;
- \( A \) is the finite set of actions;
- \( \text{inv} : S \rightarrow CC(X) \) is an invariant condition, a clock constraint for each state;
- \( \text{enab} : S \times A \rightarrow CC(X) \) is an enabling condition;
- \( \text{prob} : S \times A \rightarrow \text{Dist}(2^X \times S) \) is a (partial) probabilistic transition function;
- \( l : S \rightarrow 2^{AP} \) is the labelling function.
Example. The pChart of a simple PTA and the generated PRISM code are shown in Figure A.2 and Figure A.2. The PTA has clock \( rootclk \) with initial value 0. In the state \( S_0 \), the system waits for the \( wakeup \) event for 1s (second). \( S_0 \) also allows a transition to state \( S_1 \) when \( rootclk = 1 \) and the invariant \( root = S_0 \Rightarrow rootclk \leq 1 \) forces the transition to be taken when \( rootclk \) reaches 1. In the state \( S_0 \), the system waits for a maximum of 1s. If the event does not occur, it goes to the next state on the timed transition. On the model, properties like the expected time to reach state \( S_3 \) or the probability of reaching state \( S_3 \) in a given number of time units can be verified. In state \( S_3 \), we specify two queries. The query \( ?P.max F < 10s \) returns 0.9, the maximum probability to reach state \( S_3 \) in 10s and the query \( ?P.min F < 10s \) returns 0.819, the minimum probability to reach \( S_3 \) in 10 seconds. Those properties cannot be verified on an MDP model. While PRISM uses abstract time units, in pCharts the time unit, here \( s \), must be explicitly specified.

Figure A.2: State-transition diagram of the PTA model

A PTA in PRISM is verified by one of two model checking engines, *digital clocks* and *stochastic games* [Kwiatkowska et al., 2006, 2009]. In the digital clock engine, clock variables are allowed in \( P \) (probability) operator expressions, as well as in \( F \) (eventually) and \( U \) (until) expressions. However, this engine does not support time-bounded reachability properties and clock constraints cannot use strict comparison
operators, e.g. $rootclk < 2$. Also, comparison between clock variables is not allowed. Automata with such constraints are called closed, diagonal-free probabilistic timed automata. The digital clocks method is based on a language-level translation from a PTA model to an MDP model. In the stochastic games engine, properties cannot contain references to clocks. Only unbounded or time-bounded probabilistic reachability properties are allowed. For this, only the $P$ operator is used. The basic types of path properties that can be used inside the $P$ operator are: $X$ (next), $U$ (until), $F$ (eventually), $G$ (always), $W$ (weak until), and $R$ (release), but PTA stochastic game model checking currently (V 4.2.1) only supports the $F$ path operator. The $S$ operator, used to reason about the steady-state behaviour of model, and the $R$ operator, used to calculate reward properties, are not supported.

<table>
<thead>
<tr>
<th>Listing A.2: PTA PRISM code generated by $pState$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pta</strong></td>
</tr>
<tr>
<td><strong>const</strong> $S0$ = 0; <strong>const</strong> $S1$ = 1; <strong>const</strong> $S2$ = 2; <strong>const</strong> $S3$ = 3;</td>
</tr>
<tr>
<td><strong>module</strong> ptaexample</td>
</tr>
<tr>
<td><strong>root : [0..3]</strong></td>
</tr>
<tr>
<td><strong>init</strong> $S0$;</td>
</tr>
<tr>
<td><strong>rootclk : clock</strong></td>
</tr>
<tr>
<td><strong>invariant</strong></td>
</tr>
<tr>
<td>$(root=S1 =&gt; rootclk &lt;= 1) &amp; (root=S2 =&gt; rootclk &lt;= 4)$</td>
</tr>
<tr>
<td><strong>endinvariant</strong></td>
</tr>
<tr>
<td><strong>[wakeup]</strong> $(root=S0) -&gt; (root’=S1) &amp; (rootclk’=0)$;</td>
</tr>
<tr>
<td><strong>[wakeup]</strong> $(root=S0) -&gt; 0.3:(root’=S0) &amp; (rootclk’=0) + 0.7:(root’=S1) &amp; (rootclk’=0)$;</td>
</tr>
<tr>
<td>**[] (root=S2) &amp; (rootclk=4) -&gt; 0.1:(root’=S1) &amp; (rootclk’=0) + 0.9:(root’=S3) &amp; (rootclk’=0)$;</td>
</tr>
<tr>
<td><strong>endmodule</strong></td>
</tr>
<tr>
<td><strong>rewards &quot;r&quot;</strong></td>
</tr>
<tr>
<td>$(root=S0): 0.1; $(root=S1): 3; $(root=S2): 2; $(root=S3): 0;</td>
</tr>
<tr>
<td><strong>endrewards</strong></td>
</tr>
</tbody>
</table>
Bibliography


Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements
of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.


