6-20-2014

Formal Modeling and Analysis Techniques for High Level Petri Nets

Su Liu
sliu002@fiu.edu

DOI: 10.25148/etd.FI14071134

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

FORMAL MODELING AND ANALYSIS TECHNIQUES FOR HIGH LEVEL PETRI NETS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Su Liu

2014

To: Dean Amir Mirmiran
School of Computing and Information Science

This dissertation, written by Su Liu, and entitled Formal Modeling and Analysis Techniques for High Level Petri Nets, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Shu-Ching Chen

Peter J Clarke

Jinpeng Wei

Armando Barreto

Xudong He, Major Professor

Date of Defense: June 20, 2011

The dissertation of Su Liu is approved.

Dean Amir Mirmiran
School of Computing and Information Science

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

DEDICATION

To my parents and my girlfriend.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my advisor Professor Xudong He for his excellent and persistent guidance and caring in the last six years. He offered me a great atmosphere for doing research: not only he can always provide a solution to me when I have a research problem but also he guides me to find the solution with great patience. Without his help this dissertation will never be finished.

I also want to thank all of my dissertation committee members Professor Shu-Ching Chen, Professor Peter J Clarke, Professor Jinpeng Wei and Professor Armando Barreto for their questions and suggestions from different angles and appreciate their patience to help me and encourage me to finish my PhD study.

Besides, I want to thank Reng Zeng and Zhuo Sun for their great help as research companions and as roommates. I can always discuss problems in research and in life with them.

Last but not least, I have to thank my parents and my girlfriend for supporting me all the time.

ABSTRACT OF THE DISSERTATION

FORMAL MODELING AND ANALYSIS TECHNIQUES FOR HIGH LEVEL PETRI NETS

by

Su Liu

Florida International University, 2014

Miami, Florida

Professor Xudong He, Major Professor

Petri Nets are a formal, graphical and executable modeling technique for the specification and analysis of concurrent and distributed systems and have been widely applied in computer science and many other engineering disciplines. Low level Petri nets are simple and useful for modeling control flows but not powerful enough to define data and system functionality. High level Petri nets (HLPNs) have been developed to support data and functionality definitions, such as using complex structured data as tokens and algebraic expressions as transition formulas. Compared to low level Petri nets, HLPNs result in compact system models that are easier to be understood. Therefore, HLPNs are more useful in modeling complex systems.

There are two issues in using HLPNs - modeling and analysis. Modeling concerns the abstracting and representing the systems under consideration using HLPNs, and analysis deals with effective ways study the behaviors and properties of the resulting HLPN models. In this dissertation, several modeling and analysis techniques for HLPNs are studied, which are integrated into a framework that is supported by a tool.

For modeling, this framework integrates two formal languages: a type of HLPNs called Predicate Transition Net (PrT Net) is used to model a system's behavior and a first-order linear time temporal logic (FOLTL) to specify the system's properties.

The main contribution of this dissertation with regard to modeling is to develop a software tool to support the formal modeling capabilities in this framework.

For analysis, this framework combines three complementary techniques, simulation, explicit state model checking and bounded model checking (BMC). Simulation is a straightforward and speedy method, but only covers some execution paths in a HLPN model. Explicit state model checking covers all the execution paths but suffers from the state explosion problem. BMC is a tradeoff as it provides a certain level of coverage while more efficient than explicit state model checking. The main contribution of this dissertation with regard to analysis is adapting BMC to analyze HLPN models and integrating the three complementary analysis techniques in a software tool to support the formal analysis capabilities in this framework.

The SAMTools developed for this framework in this dissertation integrates three tools: PIPE+ for HLPNs behavioral modeling and simulation, SAMAT for hierarchical structural modeling and property specification, and PIPE+Verifier for behavioral verification.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Motivation

Nowadays, hardware and software systems are becoming larger than ever and their complexities are growing even faster. However, errors are intolerable in some of the critical systems, such as astronomy control systems, electronic commerce, highway and air traffic control systems, and medical instruments. In 2010, the giant automaker Toyota's reputation dropped significantly due to its stuck accelerator problem. Toyota has to recall vehicles up to 2.3 million in USA, 1.8 million in Europe and 75000 in China [5]. In order to prevent these accidents caused by subtle errors, reliable hardware and software systems are desired. Furthermore, as the involvement of such systems into our lives increases, producing reliable systems becomes urgent.

Formal methods have been developed to tackle this problem [29]. Formal methods are mathematically based languages, techniques and tools for specifying and verifying systems. A method is formal if it has sound mathematical basis [114]. Formal methods build a mathematically rigorous model of a complex system and use mathematical proof to ensure correct behavior of the system. Unlike traditional system designs that use extensive testing to test system behaviors but can only draw limited conclusions, formal methods offer further insurance as they only accept systems that have been proved correctly [21]. Along with the development of formal methods, some well known formal specification languages are developed, such as Z [102], VDM [66], FSMs [80], Statecharts [52], CSP [58], LOTOS [62], Alloy[64] and Petri Nets [91].

Among these formal languages, Petri nets are a promising tool for modeling and analyzing information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. Petri nets play a unique role for its graphical modeling and dynamic executable characteristics. A Petri net model is a directed bipartite graph that consists with simple graphical elements, such as places, transitions and directed arcs and they use tokens to simulate dynamic behaviors. This dynamic semantics of Petri nets make them powerful to describe dynamic systems and becomes widely recognized in industry. In addition, Petri nets are a formal language that can be defined with the integration of mathematical tools, such as state equations and algebraic equations. Thus, they can be used by both practitioners and theoreticians and can be a bridge between the two.

There are many types of Petri nets, which can be classified into two categories:

1. Low level Petri nets are [98] simple nets with only graphical elements such as places, transitions, directed arcs and tokens, which are suitable to model control flows but cannot effectively model data and functionality in complex systems.

2. High level Petri nets (HLPNs) [6] are a more expressive formalism developed to handle data and functionality in addition to control flows. The high level concepts in HLPNs include complex structured data as tokens and algebraic expressions as transition conditions.

As data is getting critical in our real world system and can influence the behavior of the system, HLPNs become more popular in modeling high level systems such as software systems. Compared to low level Petri nets, HLPNs use less graphical elements to represent richer information and are more closely matched to real world systems, thus using HLPNs simplifies our modeling process by building a more understandable and more compact system model. Therefore, HLPNs become more pragmatic in modeling complex and data oriented models.

In this dissertation, we introduce a framework for HLPNs modeling and analysis. An overall structure of this framework is shown in Figure 1.1. The framework can be applied to model a complex system formally in HLPNs and to analyze the models through different automatic methods. The framework is based on the integration of Predicate Transition Net (PrT Net) [53] (a class of HLPNs) and first-order linear time temporal logic (FOLTL). PrT Nets are used to model the behaviors of a system and FOLTL is used to specify its properties. Besides, the model in this framework can be analyzed through various automatic analysis methods, including simulation, explicit state model checking by traditional model checkers [67] and bounded model checking through SMT solvers [26, 11]. The bounded model checking method for PrT Nets is also refined by removing redundant subformulas. In addition, we present our prototype tool set, SAMTools, not only from an user's view but also from a developer's view so that the tool set becomes open source to the formal method worldwide research community.

1.2 Contributions

The main contribution of this dissertation is a framework and a supporting software tool set that can model a system in PrT Nets formally and analyze the model automatically using three alternative methods.

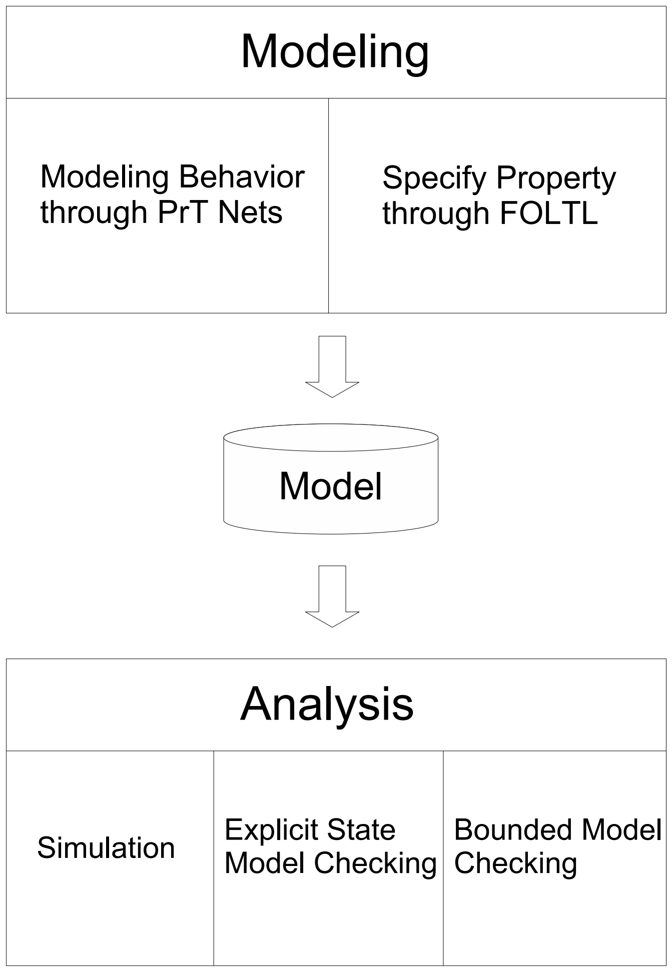

Modeling
Modeling Behavior through PrT Nets
Specify Property through FOLTL
Model
Analysis
Simulation
Explicit State Model Checking
Bounded Model Checking

Figure 1.1: Overall Structure of Framework

1.2.1 Modeling

For modeling, we developed a software tool PIPE+ [83] with a graphical editor that allow a user to model a system into PrT Nets graph with drag and drop actions. And the high level concepts can be specified on the net graph directly. In addition, the system's propertys can also be integrated into the model with PIPE+'s graphical editor.

For hierarchical modeling, we developed another tool SAMAT [84] that can support SAM, a framework for hierarchical software architecture modeling. Using SAMAT modeling editor, models are drawn via drag and drop actions and specified through components and connectors. SAMAT supports modeling in multiple layers of components thus models can be specified hierarchically using a top down manner. SAMAT also integrates PIPE+ to develop behavior models in PrT Nets.

1.2.2 Analysis

For analysis, we integrate three methods: simulation, explicit state model checking and bounded model checking.

1. Simulation: a simulator is developed specifically for PrT Nets that can execute the model by randomly firing a transition or generating a transition firing sequence. A report of the simulation is generated after each simulation;
2. Explicit state model checking: an adapter is developed to automatically convert a PrT Net model into a PROMELA [60] model and then checked by SPIN [60] model checker;
3. Bounded model checking: adapting this checking technique to PrT net with satisfiability module theories (SMT) and developed an automatic method to

convert a PrT net model into a first order logic formula and then checked by an SMT solver. This method is implemented in a software tool called PIPE+Verifier [82]. Besides, a refinement method is developed and presented. The refinement method aims to produce a reduced model that preserves reachability properties of the original model but removes redundant subformulas, thus it prevents unnecessary checking time by SMT solvers. The proof of the equivalence under reachability properties between the original model and the reduced model is presented.

1.2.3 Tools - PIPE+, SAMAT and PIPE+Verifier

Three independent tools are developed for supporting the techniques described above:

1. PIPE+ is developed for PrT Nets behavior modeling and simulation;
2. SAMAT is developed for hierarchical architecture modeling and analysis with explicit state model checking;
3. PIPE+Verifier is developed for bounded model checking PrT Net models.

Figure 1.2 presents the functionalities of our tools supported:

1.2.4 SAMTools: A Tool Set for Formal Modeling and Analysis of PrT Nets

An software tool set that integrates all the functionalities mentioned above is developed and open sourced on GIT. A high level view of SAMTools is shown in Figure 1.3.

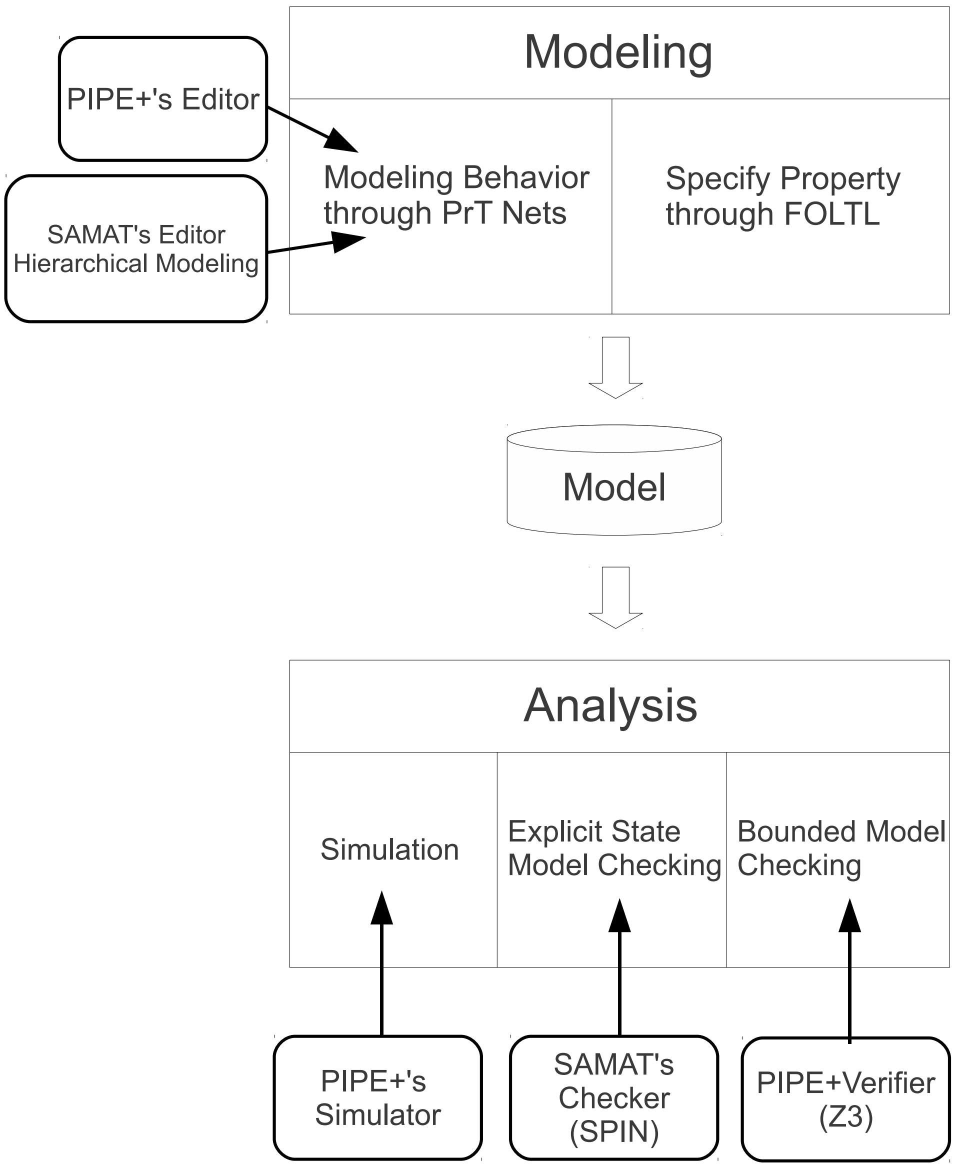

Modeling
PIPE+'s Editor
SAMAT's Editor Hierarchical Modeling
Modeling Behavior through PrT Nets
Specify Property through FOLTL
Model
Analysis
Simulation
Explicit State Model Checking
Bounded Model Checking
PIPE+'s Simulator
SAMAT's Checker (SPIN)
PIPE+Verifier (Z3)

Figure 1.2: Tools for Our Framework

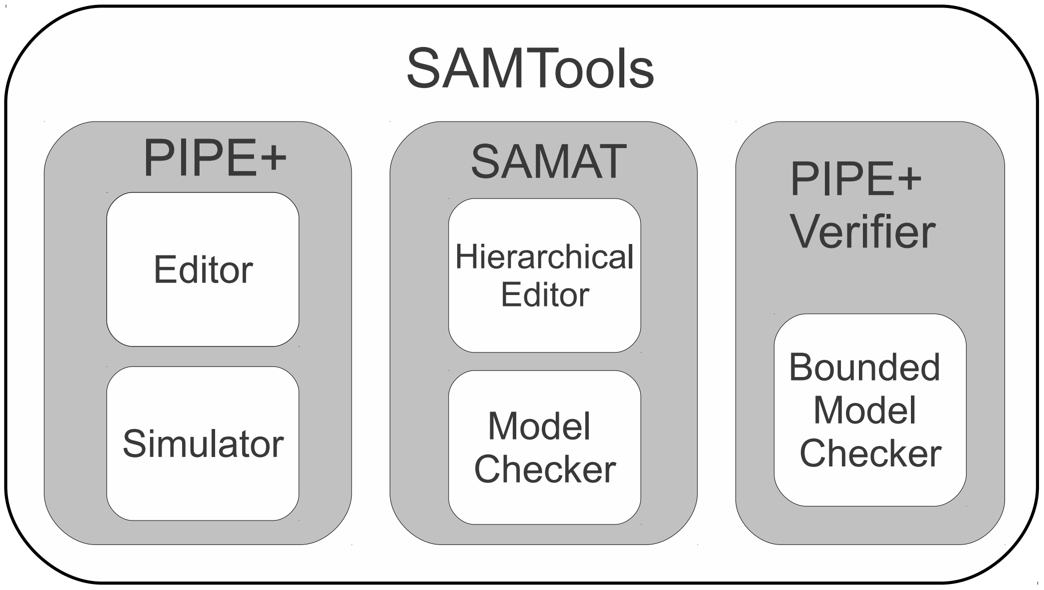


Figure 1.3: An Overview of SAMTools

1.3 Outline

In Chapter 2, we first give an overview of Petri nets and its applications. We formally define both low level and high level Petri nets and then show some examples to give a brief idea of Petri nets.

In Chapter 3, we list some related HLPNs modeling and analysis tools, symbolic model checking tools and some software architecture modeling and analysis frameworks.

Chapter 4 describes modeling based on PrT Nets, including behavior and property modeling. We present our tool PIPE+ [83] to implement and realize the ideas in modeling with PrT Nets. Besides, for scalability, we also describe a hierarchical way of modeling software architecture, which is also based on PrT Nets and software architecture components. And we present another tool implemented by ourself called SAMAT to do hierarchical modeling.

Chapter 5 describes three analysis methods to PrT Nets models, including simulation, explicit state model checking and bounded model checking. They are complimentary analysis methods that have advantages to specific requirements. These methods are all automated and supported by our tools PIPE+, SAMAT [84] and PIPE+Verifier [82] respectively. In addition, we defined a situation to refine the bounded model checking method on PrT Nets by solving a reduced formula and proved the equivalence between the original model formula and reduced model formula.

Chapter 6 presents a integration of our tools into a tool set called SAMTool.

Finally in Chapter 7, we summarize this dissertation, discuss the usefulness and future improvements to this work.

CHAPTER 2 BACKGROUND

2.1 An Overview to Petri Nets

Petri Nets were firstly introduced by Carl Adam Petri [23]. And high level Petri nets were proposed by Hartmann Genrich and Kurt Lautenbach [48]. According to [91], after years of researching in Petri Nets, they became popular and can be applied to in a lot of areas. For example, performance evaluation [8, 96], communication protocols [37, 68], distributed software systems [45, 75], distributed database systems [94, 112], concurrent and parallel programs [49, 71], flexible manufacturing/industrial control systems [34, 92], discrete event systems [72], multiprocessor memory systems [88, 101], data flow computing systems [81], fault-tolerant systems [95], programmable logic and VLSI arrays [32, 97], asynchronous circuits and structures [69, 70], compiler and operating systems [13, 93], office information systems [43, 59], formal languages [33] and logical program [47].

A Petri net structure consists of a finite set of places (drawn as circles), a finite set of transitions (drawn as bars), a finite set of directed arcs (drawn as arrows), and a set of tokens (drawn as dots) to define an initial marking. The arcs connect from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. The places can contain multiple tokens and thus are of multi set type (or bag). A distribution of tokens over the places of a net is called a marking. A transition may fire whenever there are enough tokens in all input places.

According to the international standard [6], a high level Petri net graph comprises: a net graph, place types, place marking, arc annotations, transition condition and declarations. The net graph is the net structure; place types are non-empty sets, restrict the data structure of tokens in the place; place markings are collection of elements (data items) associated with places, called tokens; arc annotations are inscribed with expressions which may comprise constants variables (e.g., x, y) and function images (e.g., f(x)); transition conditions are Boolean expressions inscribed in; declarations comprising definitions of place types, typing of variables and function definitions. For net execution, the most important is transition enabling. Enabling a transition involves the marking of its input places. When an enabled transition occurs, the enabling tokens from input place's are subtracted and the resulting tokens of the transition Boolean expression are added to the output places.

2.2 Formal Definitions

2.2.1 Low Level Petri Net

Since HLPNs are developed based on Low level Petri nets (LLPNs), we first present a formal definition of LLPNs. The formal definition is adopted from [91].

A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)\, where:$

$P = \{p_1, p_2, \cdots, p_m\}$is a finite set of places (represent as circles),

$T = \{t_1, t_2, \cdots, t_n\}$is a finite set of transitions (represent as bars),

$F \subseteq (P \times T) \cup (T \times P)$is a set of arcs,

$W : F \to \{1, 2, 3, \cdots\}$is a weight function,

$M_0 : P \to \{0, 1, 2, 3, \cdots\}$is the initial marking,

$P \cap T = \oslash$and $P \cup T \neq \oslash$.

Markings are shown by placing tokens within circles. A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by N. A Petri net with the given initial marking is denoted by (N, M_0).

The behavior of systems can be described in terms of system states and their changes. A state of marking in a Petri net model is changed according to a transition $t \in T$ is enabled and fired. A state marking sequence is denoted as $\pi = M_0 M_1 M_2 \cdots$. Since the marking can only be changed by firing transition, a state transition sequence is denoted as $\varphi = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots$.

The place $p \in P$ from which an arc runs to a transition t is called the input place of the transition t; the place $p \in P$ to which an arc run from a transition t is called the output place of the transition t. A transition t is said to be enabled if each input place p of t is marked with at least of $w\,(p, t)$ tokens, where $w\,(p, t)$ is the weight of the arc from p to t. A firing of an enabled transition t removes $w\,(p, t)$ from each input place p of t, and adds $w\,(p, t)$ tokens to each output place p of t.

Figure 2.1 illustrates a simple LLPN model and a transition firing. In Figure 2.1 (a), there are three places P_{H2}, P_{O2} and P_{H2O}, one transition t. To enable transition t, it requires two tokens from P_{H2} and one token from P_{O2} as the weight of the arc from P_{H2} to t is 2. After firing transition t, shown in Figure 2.1 (b), t consumed two tokens from P_{H2} and one tokens from P_{O2}, then produced two tokens to P_{H2O}.

2.2.2 Predicate Transition Net

Predicate Transition Nets (PrT Nets) are a class of HLPNs defined in [55, 53]. They are based on low level Petri nets structure N by incorporating high level definitions. The syntax and static semantics of HLPNs defined by a tuple $(HLPN = (N, Spec, ins))$,

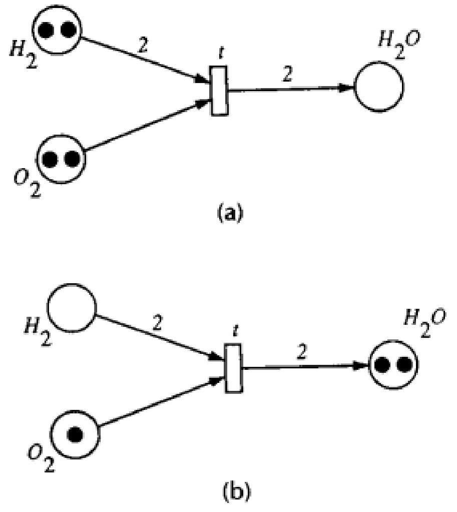


Figure 2.1: An illustration of a transition (firing rule): (a) The marking before firing the enabled transition t. (b) The marking after firing t, where t is disabled.

1. where N is partial similar to low level Petri nets representing the net structure $N = (P, T, F)$.
2. $Spec = (S, OP, Eq)$ is the underlying specification, and consists of a signature $\mathbf{S} = (\mathrm{S}, \mathrm{OP})$ and a set equations Eq. The signature $\mathbf{S} = (\mathrm{S}, \mathrm{OP})$ includes a set of sorts S and a family $OP = (OP_{s_1,\ldots,s_n,s})$ of sorted operations for $s_1, \ldots, s_n, s \in S$.
3. $ins = (\varphi, L, R, M_0)$ is a net inscription that associates a net element in N with its denotation in $Spec$:
   (a) φ is a data definition of N associates each place p.
   (b) L is a label of the net represents the relation of two elements connected with arcs.
   (c) R is well-defined constraint associates each transition in T, which defined in a first-order logic formula.
   (d) M_0 is an initial marking assigns a multi-set of tokens to each place in P.

Dynamic semantics of PrT Nets are:

1. Markings of a Petri net N are mappings $M : P \to MCON_s$.
2. An occurrence mode of N is a substitution $\alpha = \{x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n\}$, which instantiates typed label variables.
3. Given a marking M, a transition $t \in T$ and an occurrence mode α, t is enabled at M iff the predicate is true: $\forall p : p \in P. (L(p,t) : \alpha) \subseteq M(p) \wedge R(t) : \alpha$.
4. If t is enabled at M, t may fire in occurrence mode α. The firing of t with α returns the marking M' defined by $M'(p) = M(p) - L(p,t) : \alpha \cup L(p,t) : \alpha$ for $p \in P$.
5. A state transition sequence $M_0T_0M_1T_1\ldots$of N is either finite when the last marking is terminal (no more enabled transitions in the last marking) or infinite, in which each is an execution step consisting of a set of non-conflict firing transitions.

2.2.3 A PrT Net Model

2.2.3.1 Dining Philosophers Problem

Developed by [38], Dining Philosophers problem is a famous model often used in concurrent algorithm design to illustrate an inappropriate use of shared resources generating deadlocks. Five philosophers are sitting at a round table around a bowl of spaghetti but only one chopstick is placed between each pair of adjacent philosophers. The philosophers are only in two states, thinking and eating. Initially, they are in thinking state. Each philosopher must pick up chopsticks from both his left and right sides to enter the eating state and he can only pick up the chopstick on his left or the one on his right as they become available. A philosopher cannot enter

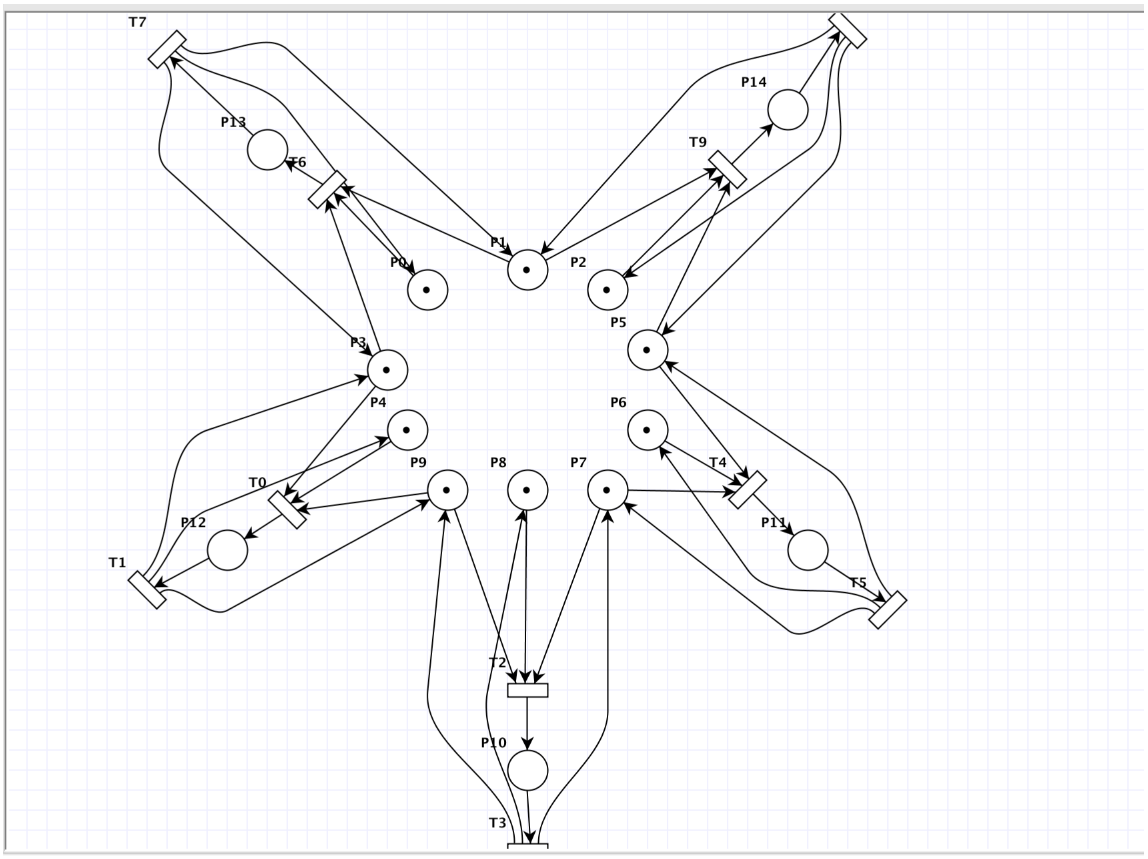


Figure 2.2: 5-Dining Philosophers Problem in Low Level Petri Net

eating state before he get both chopsticks. After he finishes eating, he needs to put down both forks so they become available to others.

2.2.3.2 Modeling the 5-Dining Philosophers Problem in Low Level Petri Nets

In order to see the advantage of using PrT Net, we first see the problem modeled with low level Petri nets illustrated in Figure 2.2.

2.2.3.3 Modeling the 5-Dining Philosophers Problem in PrT Net

Figure 2.3 illustrates a 5-Dining Philosophers problem modeled in PrT Net. The net consists of three places $P_{Phil\_Thinking}$, $P_{Chopsticks}$, P_{Phil_Eating} and two transitions

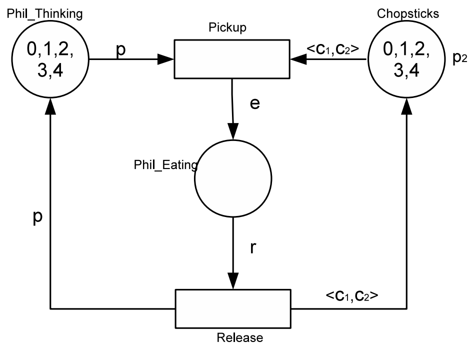


Figure 2.3: 5-Dining Philosophers Problem in PrT Net

T_{Pickup} and $T_{Release}$. All the places' token type is $\langle int \rangle$. $P_{Phil\_Thinking}$ and $P_{Chopsticks}$ are both initiated with markings that have five tokens $\{\langle 0\rangle,\langle 1\rangle,\langle 2\rangle, \langle 3\rangle,\langle 4\rangle\}$. T_{Pickup}'s transition condition is $p = c_1 \wedge (p+1)\%5 = c_2 \wedge e = p$. $T_{Release}$'s transition condition is $p = r \wedge c_1 = r \wedge c_2 = (r+1)\%5$.

2.3 Nondeterminism in PrT Nets

In low level Petri nets, nondeterminism is when multiple transitions are enabled at the same state, anyone of them may fire. But in high level Petri nets, tokens are not identical any more, in addition to the nondeterminism in low level Petri nets, the consuming of the tokens or multiple tokens that can enable a transition is nondeterministic. For example in Figure 2.3, T_{Pickup} may consume token {[0]} from $P_{Phil\_Thinking}$ and {[0], [1]} from $P_{Chopsticks}$, and T_{Pickup} may also consume token {[1]} from $P_{Phil\_Thinking}$ and {[1], [2]} from $P_{Chopsticks}$... The nondeterminism results in different markings in P_{Phil_Eating}.

CHAPTER 3 RELATED WORK

3.1 High Level Petri Net Tools

In [4], there are a number of Petri Net tools developed in the past decades. Unfortunately, many of the tools described in the database as well as in literature are no longer maintained or available and few of them support HLPNs, especially the high level Petri net definitions and notations proposed in 2001 international standard [6]. Model Checking Contest @ Petri nets (MCC) [73, 74] is held annually to assess Petri nets based formal verification tools and techniques. Some of the tools participating in MCC are quite actively maintained tools. We select some well maintained tools and list them in Table 3.1.

Some of them are further introduced below.

3.1.1 Colored Petri Nets Tool

Colored Petri Nets (CPNs) [65] are a kind of high level Petri nets that use tokens with typed values and functional programming language Standard ML [107] to define the guards of transitions. CPN Tools [1] is an industrial strength tool that is widely used to analyze modeled systems through simulation and model checking. CPN Tools integrates a model checking engine that explicitly searches the whole state space of a model.

3.1.2 ALPiNA

ALPiNA [61] is a model checker for algebraic Petri nets (APNs), which use algebraic abstract structured data type (AADTs) to define data and term equations to define

Table 3.1: Tools for High Level Petri Nets

Name	**Petri Net Type**
ALPiNA	Algebraic Petri Nets, Timed Petri Nets
CPN Tools	Colored Petri Nets
Cunf	Contextual Net
GreatSPN	High Level Petri Nets, Stochastic Petri Nets, Timed Petri Nets
ITS-Tools	(Time) Petri Nets, ETF, DVE, GAL
LoLA	High-level Petri Nets, Place/Transition Nets
Maria	High LevelPetri Nets, Modular High level nets, Labelled state transition systems
Neco	High Level Petri Nets
PEP	High Level Petri Nets, Place/Transition Nets, Timed Petri Nets
PetriSim	High Level Petri Nets, Place/Transition Nets, Timed Petri Nets
PROD	High Level Petri Nets, Place/Transition Nets
Sara	Place/Transition Nets
Renew	Object-oriented PNs, High Level Petri Nets, Place/Transition Nets, Timed Petri Nets
YAWL	High Level Petri Nets

transition guards. To symbolically model checking APNs, ALPiNA uses an extended binary decision diagrams (BDDs) to represent state space.

3.1.3 Neco

Neco [46] is a Unix toolkit that checks the reachability and other properties of high level Petri nets. Neco supports high level Petri nets annotated with Python objects and Python expressions. For model checking, Neco explicitly builds state space .

3.2 Symbolic Model Checking Tools

3.2.1 Alloy

Alloy analyzer [64] is a software tool for analyzing a system defined in Alloy specification language. The analysis in Alloy is based on reducing a model to a propositional formula and leveraging a SAT solver to solve the formula.

3.2.2 Java Path Finder

JPF [111] is a verification and testing environment for Java that integrates techniques such as model checking, program analysis and testing. Despite its state compression technique, JPF still cannot avoid the state explosion problem especially in terms of memory and time in checking high level data structures such as array.

3.2.3 CBMC and SMT-CBMC

C Bounded Model Checker (CBMC) [27] is an SAT based bounded model checker on C programs. SMT-CBMC [11] is an SMT based model checker that has significant improvement over the traditional SAT based model checkers. SMT-CBMC encodes sequential C programs into more compact first-order logic formulas that can be solved by SMT solvers.

3.2.4 MCMT: A Model Checker Modulo Theories

MCMT [50] is a fully declarative and deductive symbolic model checker for safety properties of infinite state systems whose state variables are arrays. The idea is to use classes of quantified first-order formula to represent an infinite set of states of the system so that the computation of pre-images boils down to symbolic manipulations. By performing backward reachability search procedure, fix-point can be found accurately by recursively calling underlying SMT solver. However, limitations on the MCMT tool are found when states contain complex data structures that are hard to represent and when state space getting too large that cause failure while calculating fix point.

3.3 Software Architecture Modeling and Analysis Frameworks

In the past decades, many software architecture modeling and analysis frameworks were proposed and their supporting tools were built. Some of them are:

1. Wright [9] is an architectural specification language that uses CSP [58] to specify the interactions among components as well as temporal properties.

Wright leverages FDR as its model checking engine that uses compression functions for reducing the number of states and transitions.

2. Darwin/FSP [85] is a software architecture framework for modeling and analyzing behaviors of architectures of concurrent and distributed systems. The software architecture is specified in finite state process (FSP) algebra in terms of labeled transition systems and FLTL [51] that express fluent-based properties with LTL. In addition this framework uses LTSA tool for model animation and model checking (deadlock detection and liveness properties).

3. CBabel is a declarative language that describes software architecture with modules and connectors. CBabel ADL is mostly used to model evolvable and reconfigurable architectures and focus only on safety properties. CBabel ADL specifications can be translated to Maude [42] input language and checked by Maude model checker.

More related frameworks are included in some comparative studies[121, 40]. Table 3.2 presents a list of architecture modeling and analysis frameworks and their supporting tools.

Table 3.2: Software Architecture Frameworks

Framework Name	**Hierarchical Structure**	**Formalism**	**Tool**	**Verification Engine**
Wright	No	CSP	Wright	FDR
Darwin/FSP	Yes	FSP and FLTL	Darwin	LTSA
Archware	Yes	Archware ADL and Archware AAL	ArchWare	CADP
CHARMY	No	State and Sequence Diagram and PSC	Charmy	SPIN
CBabel	No	CBabel ADL and LTL	CBabel	Maude
Auto FOCUS	Yes	Model-based	AF3	NuSMV and Cadence SMV
PoliS	No	PoliS and PoliS TL	N/A	PolisMC
Fujaba	Yes	UML and LTL/CTL	Fujaba	UPPAAL
SAM	*Yes*	*Petri Nets and FOLTL*	*SAMTools*	*SPIN, Z3*

CHAPTER 4 MODELING

In this chapter, we present some fundamental concepts of our framework, such as behavior modeling based on PrT Net and property specification with first-order linear time temporal logic. We not only present the theoretical concepts of a PrT Net model, but also use an example of 5 Dining Philosophers problem 2.3 to illustrate the process via practical modeling leveraging a supporting tool developed by us, PIPE+ [83].

4.1 Modeling with PrT Nets

Modeling a system is to create an abstraction of the system that can be investigated to find defects and potential improvements. Modeling a system often requires considerable knowledge and experience and very time consuming. PrT Nets [53] are a type of HLPNs and are formally defined that are good at modeling distributed and concurrent systems. Using PrT Nets as a foundation can facilitate the modeling process. In addition, by leveraging high level concepts such as structured tokens and algebraic formulas, PrT Nets are capable of modeling data-oriented systems.

4.1.1 Behavior Modeling

Behavior modeling in PrT Net includes building a low level Petri net graph and integrating high level concepts into the net model. Since a PrT net is a class of HLPNs, and according to [6], a HLPN comprises: a net graph, place types, place marking, arc labels, transition condition and declarations.

4.1.1.1 A Net Graph

Building the net N in PrT Net is the same as that in low level Petri nets. Since N consists of a finite set of places (drawn as circles), a finite set of transitions (drawn as bars), a finite set of directed arcs (drawn as arrows), and a set of tokens to define an initial marking. Directed arcs connect only between a place and a transition.

In general, places are used to describe conditions, data or resources; transitions are used to describe events, tasks or computation steps. They can be specified into different object depends on the specific systems. Places can only connect with transitions by directed arcs.

For example in Figure 2.3, place $P_{Phil\_Thinking}$ and $P_{Chopsticks}$ are input places for transition T_{Pickup}. $P_{Phi\_Thinking}$ denotes a condition that which philosopher is in thinking status and $P_{Chopsticks}$ provides resources of available choices of chopsticks to pick. T_{Pickup} describes an event that a philosopher in thinking status pick two chopsticks and then enter eating status P_{Phil_Eating}.

This net can be modeled by PIPE+ tool graphically by drag and drop actions (Figure 4.1).

4.1.1.2 Place Type (Token Type)

The main difference between high level and low level Petri nets is that tokens are no longer black dots, but complex structured data. Places are non-empty sets served as container to tokens. Place type is the type of the place container that restrict token's data type (the data structure of tokens) in the it. In PrT Net, each place has a place type, they can be distinct from other places.

For example, in the Dining Philosophers problem in Figure 4.1, place $P_{Phil\_Thinking}$ contains a set of philosophers ID, so its place type can be defined by [Int]. Figure

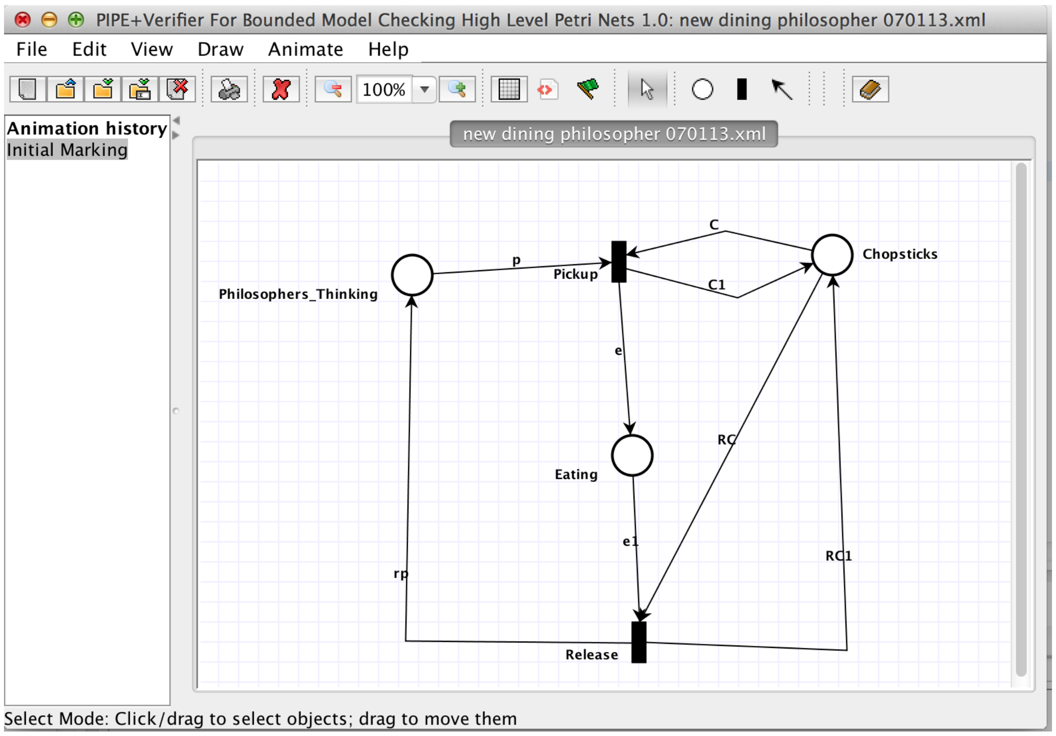

PIPE+Verifier For Bounded Model Checking High Level Petri Nets 1.0: new dining philosopher 070113.xml
File
Edit
View
Draw
Animate
Help
100%
Animation history
Initial Marking
new dining philosopher 070113.xml
C
Chopsticks
p
Pickup
C1
Philosophers_Thinking
e
RC
Eating
e1
RC1
rp
Release
Select Mode: Click/drag to select objects; drag to move them

Figure 4.1: Model Dining Philosophers Problem in PIPE+: The Net Structure

4.2, shows an editing panel in PIPE+ tool that allows users to create a type with two primitive data types, Integer and String. $P_{Phil\_Thinking}$'s place type is defined as a singleton type [int].

4.1.1.3 Transition Conditions

Transition conditions control the flow of tokens. PrT Nets use a subset of first-order logic to define transition condition formulas. Typically a formula consists of two parts: preconditions and postconditions. Preconditions are conditions that need to be satisfied to enable a transition; postconditions are assignments that if preconditions are satisfied, the transition is fired that tokens are built and distributed to related output places.

Variables in the formula are consist of variables from arc labels. Because arc label variables are instantiated by connected tokens from connected input places,

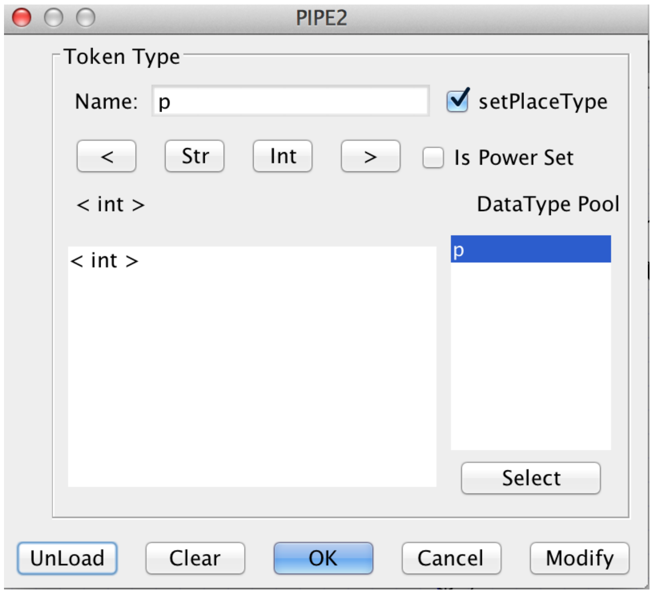


Figure 4.2: Define Place Type in PIPE+

variables in transition formula can also be instantiated by arc label variables. Thus a transition formula cannot contain any free variable that does not appear in the connected arc labels, otherwise no instantiation values can be found for free variables thus the calculation result is non-determined.

A transition formula is specified as a whole and does not explicitly separate its preconditions and postconditions. However, the pre and post conditions can be separated implicitly. Since formula variables are mapping to connected arc labels and arcs can be differentiated with input arcs and output arcs, a clause with only input variables (from input arc label) are preconditions and a clause with output variables (from output arc) involved are postconditions.

The grammar for a user to build a transition formula is defined in BNF and can be referred to Appendix 7.2.2. In the 5-Dining Philosophers Problem in Figure 4.1, the transition T_{Pickup} specified in PIPE+ Tool transition editing panel is shown in Figure 4.3. By convention, the regular letter denotes simple variable and capital letter

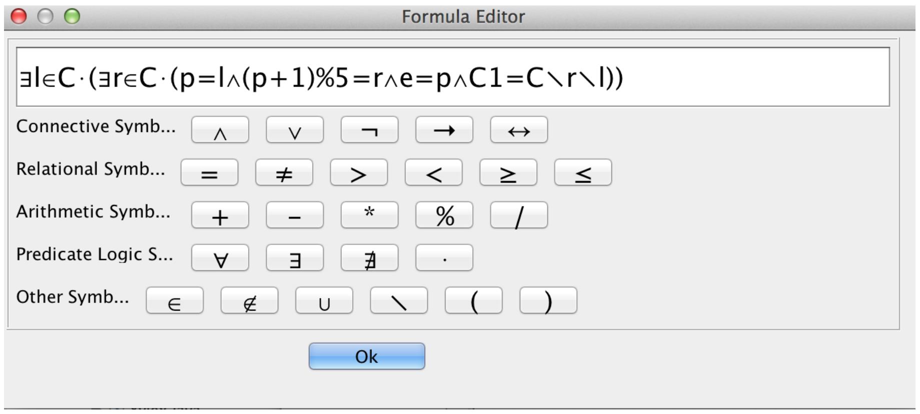


Figure 4.3: Define Transition Pickup in PIPE+

represents a set variable, the difference of them are discussed below. In T_{Pickup}'s formula, e and $C1$ are output variables, thus the clauses of them are postconditions.

4.1.1.4 Arc Label

Arc label indicates the type of variables to be instantiated with tokens. We denote the variable on arc label as arc variable. There are two types of arc variables:

1. Simple variable: denotes the token is instantiated with one regular token from connected place;

2. Set variable: denotes the token is instantiated with a token set from connected place. A token set may contain 0 or more tokens;

For example in Figure 4.1, the arc variable p on the arc connect $P_{Phil\_Thinking}$ and T_{Pickup} is simple variable and C on the arc connect $P_{Chopsticks}$ and T_{Pickup} is set variable.

4.1.1.5 Declarations:

In the standard [6], declarations include the place type definitions, type definitions, and and function definitions. In PIPE+, the declaration mechanism is realized in the modeling process through defining place data types, transition condition formulas and arc annotations.

For Dining Philosophers problem in Figure 4.1, $P_{Phil\_Thinkng}$ defines the token in this place means the philosopher is in thinking state, P_{Phil_Eating} defines the token in this place means the philosopher is in eating state and $P_{Chopsticks}$ defines the token in this place indicates the chopstick is in idle state and available for a philosopher to pick up.

4.1.1.6 Tokens and Abstract tokens

As mentioned above, there are two types of arc variables in PrT Net. Similarly, there are two types of token:

1. Regular token: we also called it token in brief. Regular token is to be instantiated with simple variable in arc label. It is a tuple that constructed by different primitive types of data such as integer or string. It often used to describe individual elements with a list of properties. For example, a student may be defined as a token with properties [ID, Name, Gender, AccountNumber];
2. Abstract token (abToken): abstract token is to be instantiated with set variable in arc label. It is a set container that has a list of tokens. Using abToken is a little special because it instantiates a token set, the correlated transition formula needs to use quantifier like $\exists$ and $\forall$. For example, a library booklist is specified using a place P_{books} and the arc variable is B, the correlated transition formula must be $\exists x \in B$ or $\forall x \in B$, where x is a user defined variable.

However, the nested sets are flattened by duplicating some fields. For example, in a library system, a user may borrow a list of books, so that the database (power set) in library system is $username, password, books, borrowedbook1, book2, ...$ that is converted into $username, password, book1$, $username, password, book2$. This design trades space for simplicity and can be improved in the future.

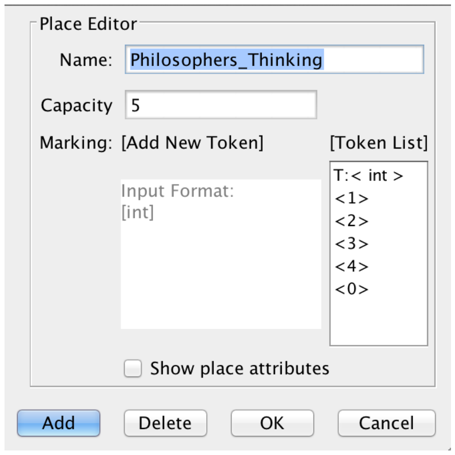
While specifying a model in PrT Net, adding tokens/abTokens to place is to specify the initial marking M_0. In modeling dining philosophers problem using our PIPE+ Tool, we add tokens to places shown in Figure 4.4 and 4.5. In $P_{Phil\_Thinking}$, tokens are integers that declared as philosophers' ID. In $P_{Chopsticks}$, tokens are integers that declared as chopsticks' ID. In the initial marking, P_{Phil_Eating} is empty shown in Figure 4.6.

4.1.1.7 Place Bound

Theoretically, a place is a multi-set that can contain unlimited number of tokens. However, for analysis reason, sometimes we need to define a bound to it that limit the maximum number of tokens in each place. In practical, every place contain has a limit size. Thus, by leveraging model checking, these bounds limits the state space of the model and improves the efficiency. Otherwise, the model checking method can suffers from the state explosion problem easily.

4.1.2 Property Modeling

As the goal of our modeling is to check whether a model N satisfies property f, denotes as $N \models f$. During modeling process, properties f need to be specified. In this framework, first-order linear time temporal logic (FOLTL) is used to specify the properties (or constraints).


Place Editor

Name: Philosophers_Thinking

Capacity 5

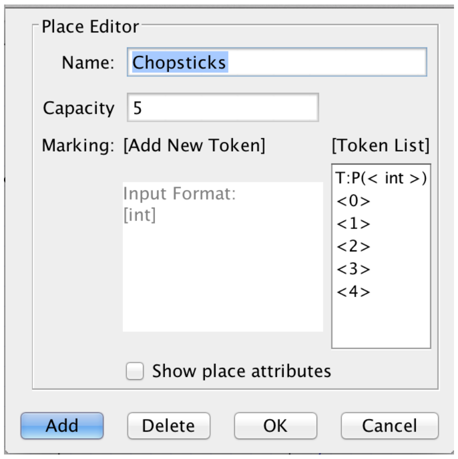
Marking: [Add New Token]

Input Format: [int]

[Token List]

```
T:< int >
<1>
<2>
<3>
<4>
<0>
```

Figure 4.4: Specify Initial Marking in Place Philosophers_Thinking in PIPE+


Place Editor

Name: Chopsticks

Capacity 5

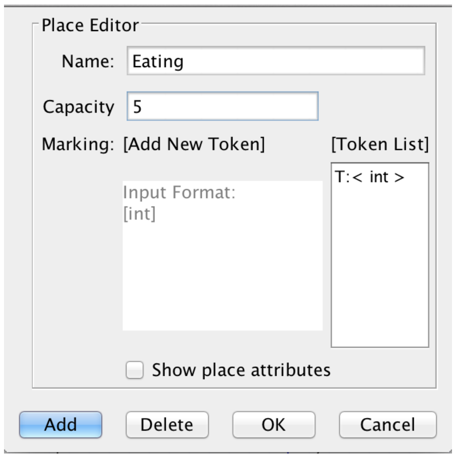
Marking: [Add New Token]

Input Format: [int]

[Token List]

```
T:P(< int >)
<0>
<1>
<2>
<3>
<4>
```

Figure 4.5: Specify Initial Marking in Place Chopsticks in PIPE+


Place Editor

Name: Eating

Capacity 5

Marking: [Add New Token]

Input Format: [int]

[Token List]

```
T:< int >
```

Figure 4.6: Specify Initial Marking in Place Philosophers_Eating in PIPE+

4.1.2.1 Temporal Formulas

Following the definitions in [56], a predicate is a boolean valued function. A predicate p is said to be true under certain state S if $S[\![p]\!]$ is true. A transition is a special predicates that contain primed state variables that indicates the next state, denoted as $S'[\![p]\!]$. A transition relates to two consecutive states, the current state and the next state, where the unprimed state variables refer to the current state and primed ones refer to the next state. Thus, a transition connects the relation of two states.

Temporal logic formulas are built from elementary formulas (predicates and transitions in PrT nets) using logical connectives $\neg$ and $\wedge$ (and derived logical connectives $\vee$, $\Rightarrow$ and $\Leftrightarrow$), the existential quantifier $\exists$ (and derived universal quantifier $\forall$) and the temporal always operator $\Box$ (and the derived temporal sometimes operator $\Diamond$). The vocabulary and models of FOLTL used in PrT Nets are based on the high-level Petri net formalism and follow the approach defined in [79]. An example FOLTL formula is $\Box$((x>y) $\Rightarrow$ $\Diamond$ (b=1)), where variables are restricted to those arc variables in the underlying PrT nets.

The semantics of temporal logic is defined on a sequence of states in the behavior of the net model. As the behavior reflects the execution sequence of the net model, a temporal logic formula defines the execution sequences in a net model.

Let u and v be two arbitrary, p be an n-ary predicate, t be a transition, $x_1, x_2, ..., x_n$ are variables. $\sigma = [\![M_0, M_1, ...]\!]$ be a behavior, we define the semantics of temporal formulas recursively as follow:

1. $\sigma[\![p(x_1, x_2, ..., x_n)]\!] \equiv M_0[\![p(x_1, x_2, ..., x_n)]\!]$;
2. $\sigma[\![t]\!] \equiv M_0[\![t]\!]M_1$;
3. $\sigma[\![\neg u]\!] \equiv \neg\sigma[\![u]\!]$;
4. $\sigma[\![u \wedge v]\!] \equiv \sigma[\![u]\!] \wedge \sigma[\![v]\!]$;

5. $\sigma[\![\forall x.u]\!] \equiv \forall x[\![u]\!]$;

6. $\sigma[\![\Box u]\!] \equiv \forall n \in Nat\sigma^n[\![u]\!]$;

7. $\sigma[\![uUv]\!] \equiv \exists k\sigma^k[\![v]\!] \wedge \forall 0 \leq n < k\sigma^n[\![u]\!]$.

A temporal formula u is satisfiable, denoted as $\sigma \models u$, iff there is an execution σ such that $\sigma[\![u]\!]$.

4.1.2.2 Defining Properties

For example in Dining Philosophers problem, a property indicates the two neighbor philosophers cannot eat at the same time can be specified by the following equation:

$$\Box\neg\left(marking\left(Eating\right) = Phil_i \wedge marking\left(Eating\right) = Phil_{i+1}\right)$$

4.1.3 PIPE+ Tool

To support the practical applications of Petri nets formalism, tools for designing and executing Petri nets are necessary. Although there are many existing tools for supporting low level Petri nets, few tools are available for high level Petri nets. There is especially a lack of tools to support high level Petri net notation proposed in the international standard [6]. In this section, we present a tool, called PIPE+, to support a subset of HLPNs. PIPE+ can modeling a HLPN model graphically with drag and drop and it provides an editor to specify high level concepts into the graphical net, such as customized data structures and transition formulas. Besides, PIPE+ provides a simulator for analyzing models by simulation.

PIPE+ is built upon an existing low level Petri net tool PIPE (Platform Independent Petri Net Editor) [20] and is an open source tool and thus is available for various enhancements from worldwide research community.

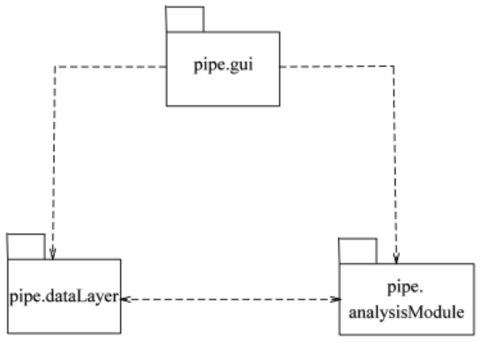


Figure 4.7: Package Diagram for PIPE

4.1.3.1 PIPE

PIPE [20] is a Platform Independent Petri net Editor to edit, animate and analyze low level Petri nets, which has clear design and incorporates the latest XML Petri net standards of storing format, the Petri Net Markup Language (PNML). It is implemented in Java and can be logically divided into three major components, shown in Figure 4.7: the graphical user interface (GUI), a layer managing the interactions between the GUI and the modules (DataLayer), and analysis modules.

Graphical User Interface PIPE's graphical user interface is developed using Java Swing API as it provides full GUI functionalities and mimics the platform it runs on. Besides, as PIPE is a cross platform application this was deemed useful for providing a native look and feel. The GUI component includes GUIFrame, GUIView and classes such as action, handler and widgets supporting Swing APIs. From a user perspective, there are two major parts: Editor and Simulator.

• Editor: Users are able to edit a low level Petri net by clicking and drawing Petri net graphical elements through the menu bar, toolbar. On the toolbar, it lists all the Petri net element thumbnails, such as place, transition and arc, which can be selected and added to the white canvas (tabbed pane) of the editor. Besides, these added elements' annotations and attributes can be defined by selecting one of the elements and pop up an editing dialog box.

• Simulator: There is a switcher button between editor mode and simulation mode. Using the simulator, a user is able to fire a random transition or fire a number of transitions randomly selected among enabled ones. The simulation process includes subtracting tokens from input places and adding them to output places while firing a transition. Besides, the animation history is displayed on the left bottom of the interface frame by listing transition's label orderly.

Internal Architecture of PIPE—The DataLayer The core component of PIPE is the data layer, which maintains states and contains all the classes used to represent a Petri net. Figure 4.8 shows the hierarchy of important Petri net object classes, including Arc, Place and Transition classes inherited from PetriNetObject because they have common variables and methods, such as id, name, location, etc.

In the data layer component, each Petri net is encapsulated by an instance of the DataLayer class, which contains all the Petri net objects stored in a list enabling the easy addition of new objects. It contains not only methods to access all its internal objects and to return its internal lists, but also methods to calculate the current markup, initial markup, forwards incidence matrix, backwards incidence matrix, combined incidence matrix and enabled transitions.

In addition, PIPE has analysis module to do analysis and conclusions on the properties of Petri net model, such as boundedness, liveness, reachable markings and so on.

Formatted Input and Output PIPE is capable of saving and loading nets and writing the Petri net data layer into a Petri net Markup Language (PNML). An

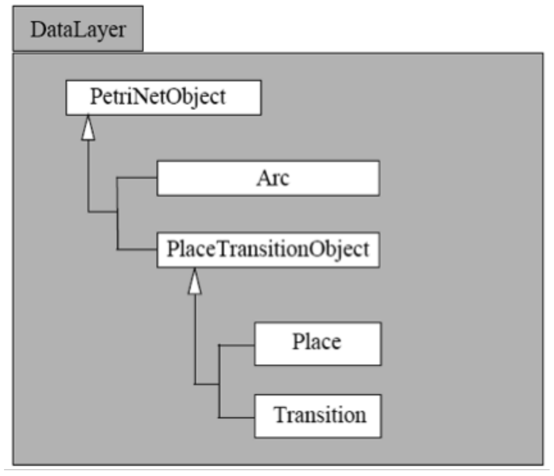


Figure 4.8: The Hierarchy of PetriNetOjbect Classes

Extensible Stylesheet Language Transformation (XSLT) is used to transform it between PNML and XML files.

4.1.3.2 PIPE+: Extension on PIPE

Similar to PIPE, PIPE+ is also an editor and a simulator. The editor is to model a system visually through a graphical interface. The goal is to utilize all the benefits that a HLPN provided with convenience. The details are presented below according to the HLPN concept's six elements in [6]. The simulator is no longer a simple black dot token animation game but to manage the movement of meaningful data. We developed a mandatory compiler with an interpreter to process token data inside transition conditions, which are defined using restricted first-order logic. Besides, a simulation algorithm is applied to ensure its fairness and improve its performance.

A Net Graph Since the graphical elements of a HLPN are the same as low level ones, PIPE's graphical editor is retained.

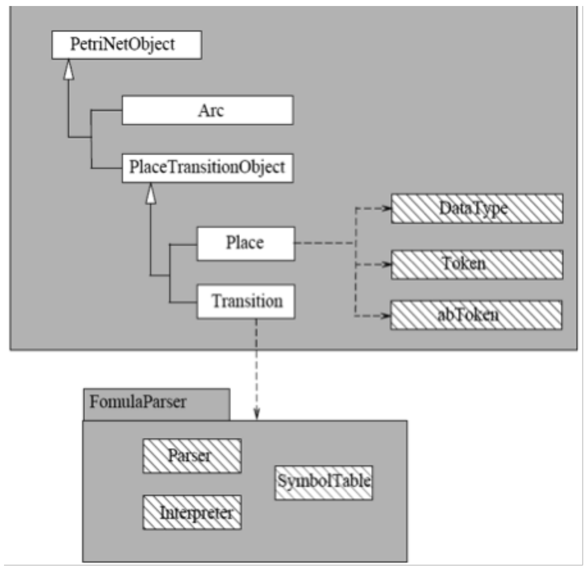


Figure 4.9: Extensions on DataLayer for PIPE+

Place Type and Place Marking The main difference between high level and low level Petri nets is that tokens are no longer black dots, but complex structured data. Place types are non-empty sets that restrict the data structure of tokens in the places. The data structure is an array of basic types, such as integer and string, and defined by user. For example, assuming a log in user account as a token has two elements, username and password, which are represented by two basic data types, string and integer. In a HLPN's place, a place data type is inscribed to restrict the data structure of tokens. In another way, the data type of tokens can be added into the place has been already defined beforehand.

To implement the concept that tokens with data structure, a data storage system is needed. Based on PIPE, the data layer package is modified by adding three classes: DataType, Token, abToken (Figure 4.9).

1. DataType: The main data structure in class DataType is a list storing basic types' name, which is used to show what data structure the token or place holds. The data structure consists of an array of basic types, such as string,

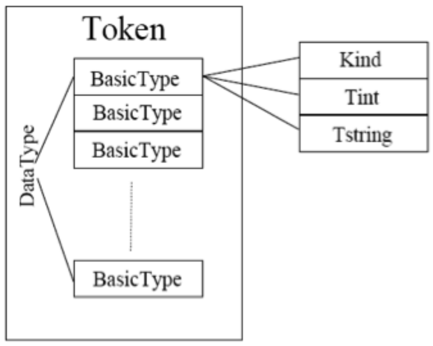


Figure 4.10: Structure of Class Token

integer, etc. For our tool, basic types are limited to strings and integers for the simplicity but are adequate for most of applications. For the convenience of extension on basic types, we introduce a new structure BasicType to data layer. The structure BasicType (see Figure 4.10) includes a flag data field "Kind" to indicate which type it is (in PIPE+, 0 represents integer, 1 represents string). Space is allocated to both integer and string since it is undecided before the “Kind” is defined. Further extension on basic types needs to enhance the class BasicType by allocating extra space and redefine “Kind”.

2. Token: Class Token is added to the data layer to maintain data value. The important field is a list storing instances of value with type of the BasicType, see Figure 4.10. Token is a basic data storage element in the places and its value is calculated by the transitions. The simulation process is fetching data value from the token’s BasicType and fill the calculated result value to another token’s BasicType.

3. Abstract Token: Since first-order logic covers quantification, the whole collection of tokens in a place need to be checked by transition condition expressions. For example, if an expression includes “$\exists x \in X$”, all the tokens in “X” needs to be checked to see whether a “x” exists, so the whole collection of tokens

is fetched while checking enabledness of a transition. The tokens in this type of place are defined as a power set. A new class abToken (abstract token) is added into the data layer to store the power set. It has a field storing a list of regular tokens with the same data type, so it also has a data type to restrict the tokens data structure. We flatten the nested power sets by duplicating some fields. For example, in a library system, one user may borrow a list of books, so that the database (power set) in library system is {username, password, books_borrowed{book1, book2,...} } is converted into {username, password, book1}, {username, password, book2}. This design sacrifices the space for the convenience of implementation, which can be further improved.

As a result, the places in PIPE+ stores a list of regular tokens or an abstract token that contains a collection of regular tokens. Whether the connected transition can fetch a regular token or an abstract token depends on the place is a power set or not. The user can add, edit and delete tokens from places to create a net marking.

In PIPE+, a place stores tokens by List container, the place's capacity is built as unbounded (remember it has nothing to do with the number of different tokens that may appear in a particular place). However, in the discussion of [57], bounded and unbounded places have the same expressive power. A bounded place is preferable for the reason of visualization and redundancy.

In PIPE+, copies of token are allowed to store in the same place. Since whether the place needs to remove its copies of token depends on what the model it is, this can be further improved by supporting an option of copy remove.

Transition Conditions and Arc Annotations Transition conditions are guards controlling the flowing of the tokens. PIPE+ use first-order logic to define transition condition formulas, which, syntactically, consists of variables and logic operators.

Variables in the formula are predicates that can be instantiated by value from input tokens. Combined with logic operators the formula can be calculated. Semantically, as transition is a guard to control token flows, it has to check the value of tokens from input places and formulate new tokens conform to the output place type, the formula consists of two parts: pre-condition and post-condition. However, in PIPE+, the user is not supposed to separate the two conditions explicitly, because the interpreter can differentiate them by the type of variables.

In PIPE+, arc annotations are variables to assist transition expression calculation by mapping token values to expression's predicate variables. Arc variables are restricted to be appeared in the connected transition expression's variables for the mapping. Since a transition is connected by input and output arcs and arcs are connected to places, the predicate variables in the transition expressions are classed into input variables and output variables. For example, in Figure 2.3, p is input variable while e is output variable.

Unless the transition formula cannot be satisfied by the value of the tokens fetched into symbol table, the tokens from input places (both regular token and abstract token) are consumed. However, a power set place with an abstract token usually has a backward arrow that makes it an output place as well, so the abstract token is returned to the place according to the post condition of the transition formula. In the case when the formula is unsatisfied, the currently fetched tokens in the symbol table are not consumed, so they are returned to the input places.

- Restricted First Order Logic Transition Formula Expression: In PIPE+, it is called restricted because the grammar we built for the tool has limitations. Since each predicate variable has to be instantiated, the user cannot use free variable that does not appear in the arc annotation, otherwise the calculation result is undetermined. Also, it does not support predefined function, like f(n), since the meaning of the function has to be declared beforehand, which is equivalent to define its operations in a single logical sentence by using the connecting operator "$\bigwedge$", which simplifies the implementation of expression interpreter. However, the restricted version of first-order logic is still very powerful, because it does support complex expressions, such as:

$$(a = b) \wedge \exists c \in C((c[1] > c[2]) \wedge (C' = C \backslash \{b\} \cup \{[a[1], a[2]]\})) \qquad (4.1)$$

In Equation 4.1, lower case letters represent regular tokens, upper case represent power set; C' by convention represents output variables and also is a power set (upper letters); it further indicates the clause is a post-condition because output variables at the left side of the equation means assignment; c[n] means the nth element value in c's data structure.

- Declarations: In [6], it comprising definitions of place types, typing of variables and function definitions. In PIPE+, the declarations are already in the modeling process by defining place data types, transition condition formulas and arc annotations.

Extensions On GUI The GUI package in PIPE mainly consists of a GUIFrame, a GUIView, and some supporting classes. The GUIFrame is PIPE's graphical frame includes a menu, a toolbar and a status bar. The GUIView is the panel to draw Petri net graphical elements. Since requirements and concepts for PrT Nets are token storage and flow, our modification to PIPE's GUI is focused on Petri net elements places, transitions and arcs. The common procedure to extend PIPE's GUI is adding new selections on graphical elements' property setting menu for new features. In PIPE+, after modifying the gui.handler package for each Petri net element class, the new selections are shown in a popup menu by right clicking a

Petri net element. The places now have the choices of defining data type and editing tokens; the transitions can contain logical formulas; the arcs can be labeled by variable key. These new features are triggered by additional selections on GUI and used through customized panels or dialogs.

4.1.4 Modeling In PIPE+

We selected some benchmark models from Model Checking Contest @ Petri nets (MCC) [73, 74] and other resources and modeled them in PIPE+. In the following chapters, these model will also be analyzed.

4.1.4.1 Shared Memory Model

In [24], a shared memory model involving P processors was given. These processors can access their local memories as well as compete for shared global memory using a shared bus. P is the number of processors and is scalable. It can be modeled as P number of tokens.

4.1.4.2 Token Ring

A token ring [39] model shows a system with a set of M machines connected in a ring topology. Each machine can determine if it has the privilege (the right) to perform an operation based on its state and its left neighbor.

4.1.4.3 Abstract State Machine Model

In [110], a method for checking symbolic bounded reachability of abstract state machines was presented. An abstract state machine written in AsmL was trans-

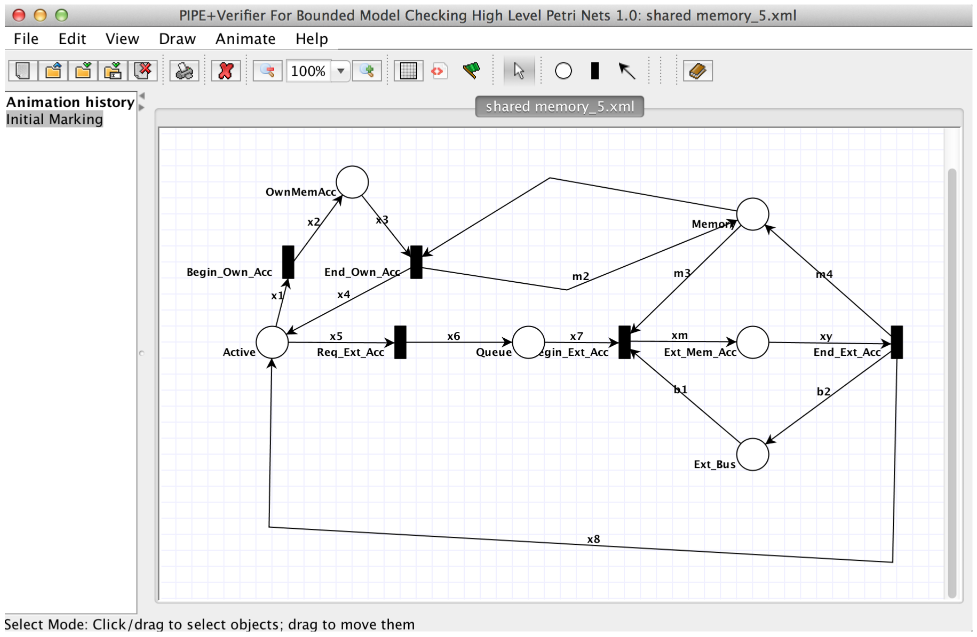


Figure 4.11: Shared Memory Model

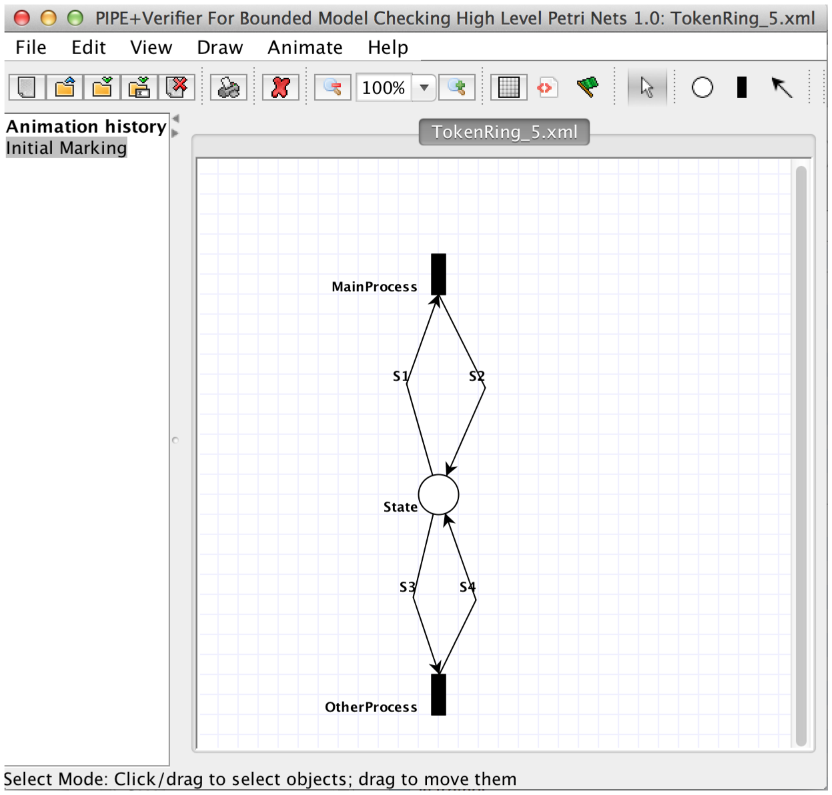

PIPE+Verifier For Bounded Model Checking High Level Petri Nets 1.0: TokenRing_5.xml
File
Edit
View
Draw
Animate
Help
100%
Animation history
Initial Marking
TokenRing_5.xml
MainProcess
S1
S2
State
S3
S4
OtherProcess
Select Mode: Click/drag to select objects; drag to move them

Figure 4.12: Token Ring Model

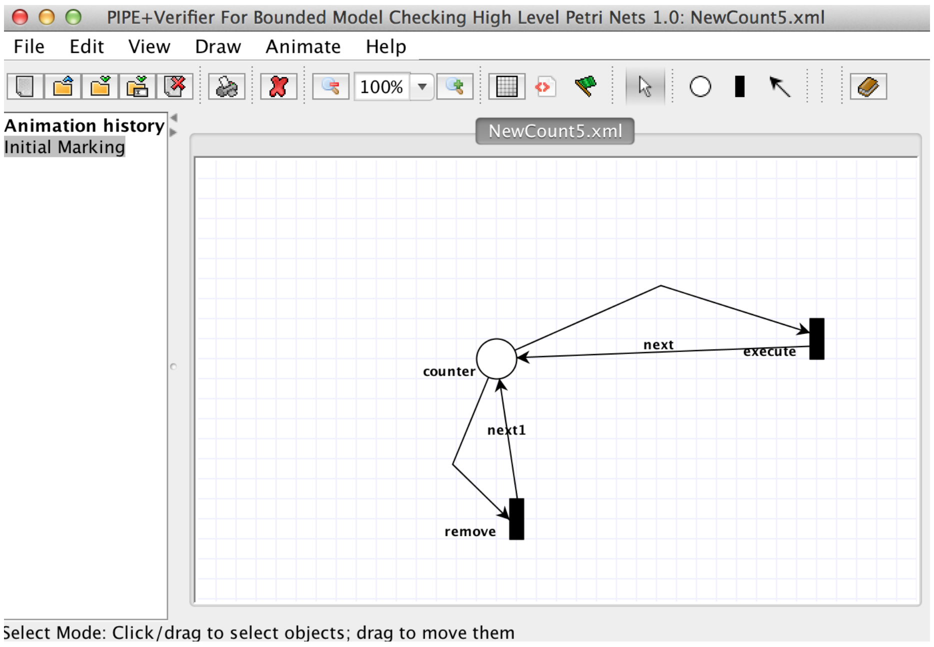

PIPE+Verifier For Bounded Model Checking High Level Petri Nets 1.0: NewCount5.xml
File
Edit
View
Draw
Animate
Help
100%
Animation history
Initial Marking
NewCount5.xml
next
execute
counter
next1
remove
Select Mode: Click/drag to select objects; drag to move them

Figure 4.13: Abstract State Machine Model

lated into a logic formula checked by an SMT solver with rich background theories including set comprehensions.

4.1.4.4 A Seabed Rig Robotic System

Seabed Rig and Energid Inc. are developing a graphical tool for constructing motion sequences that require the cooperation for multiple robots called Rig Drill Floor (RDF) sequences generation tool. Our tool PIPE+ have been used for assisting detecting design errors for the generated motion sequences for several robots. It is still an on going project as we are showing a small example of how it can be benefited.

A workflow of two robots, Robot and Roughneck, is showing in Figure 4.14. Each column is a workflow sequence of a robot. Atomic tasks (box) for the robots are stacked in the appropriate columns and are executed in a FIFO manner. "Cues"

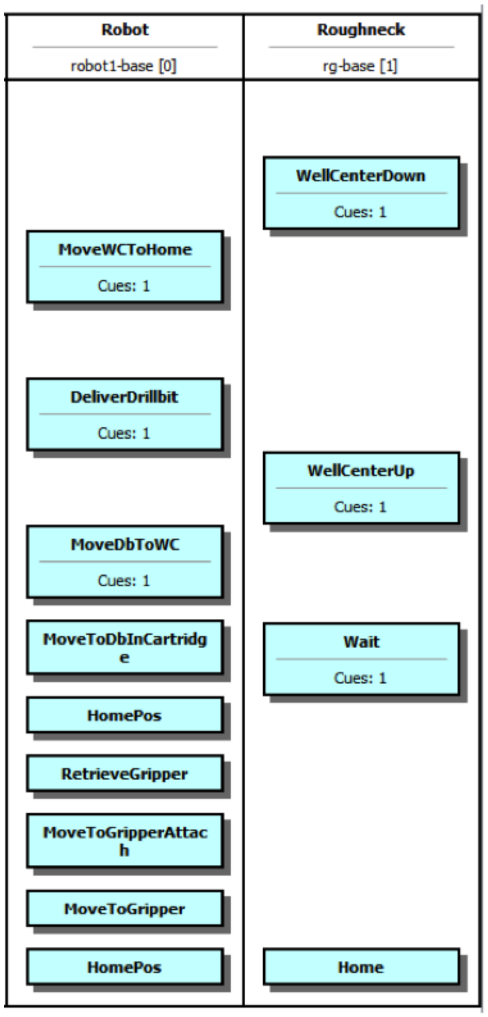


Figure 4.14: Robots motion sequences workflow

in the box indicates synchronization of two tasks among robots. A HLPN model is built by PIPE+ (Figure 4.15).

4.2 Hierarchical Modeling

Although HLPNs is a compact form of low level Petri nets, it becomes difficult to manage as the systems are getting larger and more complex. Hierarchy mechanisms allows us to decompose the target systems into modules thus simplify our modeling

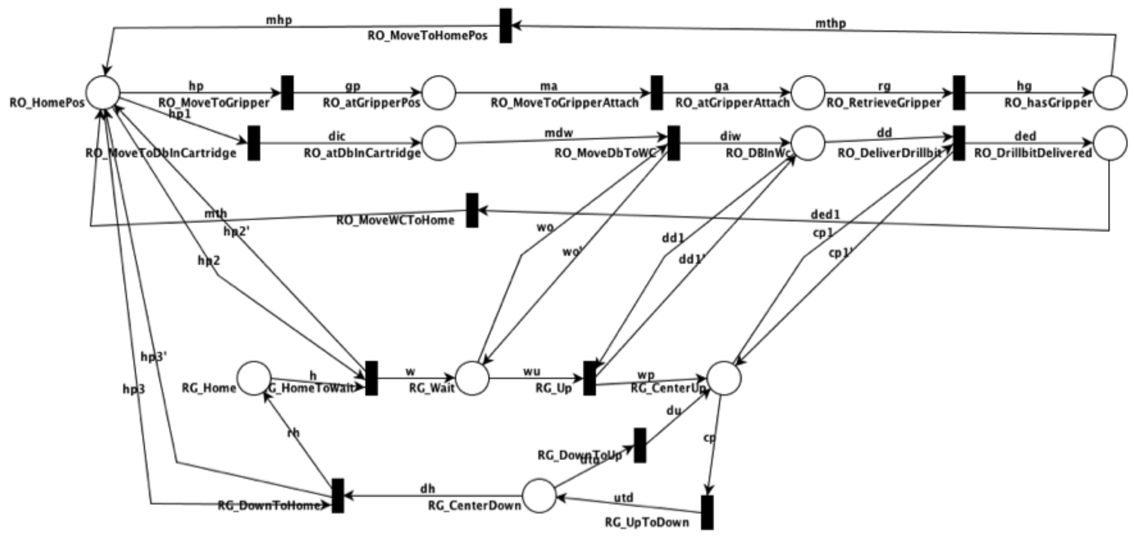


Figure 4.15: PrT Net Model for the Roboic Motion Workflow

process. By leveraging hierarchical modeling in PrT Nets, we are not only able to handle systems with bigger size, but also producing a more understandable model that easier for future development and maintenance.

SAM [113][55] is a general formal framework for specifying and analyzing software architecture and has been developed in Florida International University for years. SAM supports hierarchical modeling and analysis of software architecture. The formal foundation of SAM is based on PrT Net and FOLTL, thus SAM is able to hierarchically and modulely models and analyzes PrT Net.

4.2.1 The SAM Framework

The architecture in SAM is defined by a hierarchical set of compositions, in which each composition consists of a set of components (rectangles), a set of connectors (bars) and a set of constraints to be satisfied by the interacting components. The component models the behavior (Petri net) and communication interfaces (called ports, represented by semicircles). The connectors specify how components interact with each other . The constraints define requirements imposed on the components and connectors, and are defined by temporal logic formulas.

Figure 4.16 shows a hierarchical SAM specification model. The boxes, such as A1 and A2, are called compositions, in which A1 is component and A2 is connector. Each composition may contain other compositions, for example A1 containing three compositions: B1, B2 and B3. Each bottom-level composition is either a component or a connector and has a property specification (a temporal logic formula). The behavior model of each component or connector is defined using a Petri net. Thus, composing all the bottom-level behavior models of components and connectors implicitly derives the behavior of an overall software architecture. The intersection among relevant components and connectors such as P1 and P2 are called ports. The ports form the interface of a behavior model and consist of a subset of Petri net places.

4.2.2 The Foundation of SAM

The foundation of SAM is based on two complementary formal notations: predicate transition nets [48, 53] (a class of high-level Petri nets) and a first-order linear-time temporal logic [87].

1. Predicate transition nets (PrT nets), a class of high-level Petri nets (HLPNs), are used to define the behavior models of components and connectors. A PrT net comprises a net structure, an underlying specification and a net inscription [53]. A token in PrT nets contains typed data and the transitions are inscribed with expressions as guards to move or stop the tokens.

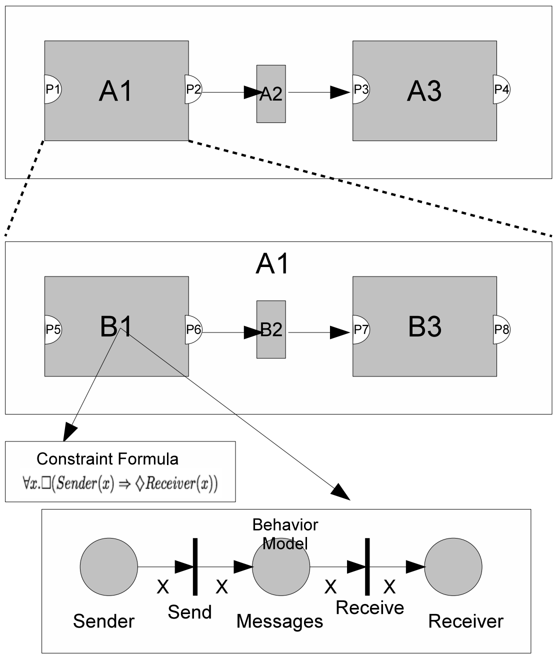


Figure 4.16: Hierarchical SAM Specification Model

2. First-order linear time temporal logic (FOLTL) is used to specify the properties (or constraints) of components and connectors. The vocabulary and models of our temporal logic are based on the PrT nets. Temporal formulae are built from elementary formulae (predicates and transitions) using logical connectives $\neg$ and $\wedge$ (and derived logical connectives $\vee$, $\Rightarrow$ and $\Leftrightarrow$), the existential quantifier $\exists$ (and derived universal quantifier $\forall$) and the temporal always operator $\Box$ (and the derived temporal sometimes operator $\Diamond$).

SAM supports the behavior modeling using PrT nets [83] and property specification using the FOLTL. SAM supports structural as well as behavioral analysis of software architectures. Structural analysis is achieved by enforcing the completeness requirement imposed on the components and connectors within the same composition, and the consistency requirement imposed on a component and its refinement.

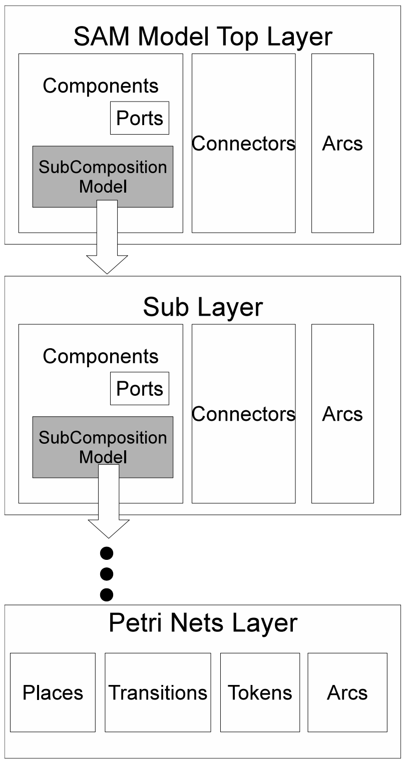


Figure 4.17: The SAM Hierarchical Model

Behavioral analysis requires the checking of system properties in FOLTL satisfied in the behavioral models in PrT nets.

4.2.3 Hierarchical Structure Modeling

In SAMTools, the graphical editor allows users to build a model through drag-and-drop actions of drawing and connecting components and connectors. Horizontally, components are connected with each other through connectors and arcs. Vertically, components can be decomposed and refined by a new layer that consists of, again, components, connectors and arcs. The bottom level component is a PrT net capturing the behavior of a subsystem. The hierarchical model stored in SAMTools is shown in Figure 4.17. A user can model an arbitrary number of levels through this recursive layered structure.

4.2.4 Behavior Modeling

4.2.4.1 Modeling Modulely and Hierarchically

The behavior of SAM model is specified modulely. Each component in SAM's architecture contains a PrT Net model and they can communicate with each other through connector, so we specify PrT Net separately for each component but need to take into account the outputs and inputs by interacting with output and input connectors.

The behavior of SAM model is specified hierarchically. Since each component may contain sub-components and each sub-component contains a PrT Net model, this allows us to specify a model hierarchically either from top down direction or from bottom up direction.

Therefore, with these mechanisms, we can decompose complex behaviors of a big model into small behavior models and can specify the behavior model through hierarchical layers. Besides, building a model in this way, we get a model with better maintainability that can be easily understood and extended.

4.2.4.2 An Example

For example, in an electronic transaction system relates to two counter parties. Each one of them should generate a request, one for send money and one for receive money, and a communication channel needs to be built. Thus, we can model a system's top layer behavior with three components, counter party 1, counter party 2 and a channel. Then, for counter party 1 and 2, it has a behavior model for generate send money request, a behavior model for receive money receiver and a transaction verification process. For the channel, it has a behavior model for receive message, processing message and send message. Therefore, all the behavior models constructs the whole transaction behavior model.

4.2.4.3 Communications among Components

The behavior models are specified modulely in different components. They can communicate with other components through connectors. For each behavior model in a component, a place is defined as input interface place and mapped to the input port of the component, a place is defined as output interface place and mapped to the output port of the component. The port are served as window of the component and connect connectors. Thus the behavior models from different components can be connected through connectors. Since some properties to be analyzed may relates to the whole system, a strategy to unite all the behavior models together is described in Section 5.5.1.

4.2.5 Property Modeling

4.2.5.1 Composition Level Properties

Since in each composition, a property may be defined and must be satisfied regardless of other compositions. This composition-level property specification is obtained by conjoining the property specifications of all components and connectors.

4.2.5.2 System Level Properties

System level properties are defined regard to the whole system, so the property may related to any place in a behavior model. Considering the system as a big model, the system level property is specified just similar to the properties in regular model. Before analyzing a system property, it is necessary to build the overall model by connecting all the behavior models distributed in different components and layers. The building process is explained in the Section 5.5.1.

4.2.6 Modeling in SAMAT Tool

We implemented SAMAT tool to support SAM Framework modeling and analysis.

4.2.6.1 Functional View of SAMAT

SAMAT is comprised of a modeling component, a SAM model, and an analysis component (Figure 4.18). The modeling component has three functions: structure modeling creates hierarchical compositions, behavior modeling specifies behaviors of software components/connectors using Petri nets, and property modeling defines property specifications using temporal logic. The SAM specification is a hierarchical structure integrating the results of structure, behavior, and property modeling, which can be transformed into XML format. The analysis component contains a translator to generate a model suitable for model checking.

4.2.6.2 Design View of SAMAT

SAMAT is a platform independent (implemented in Java) and visual software architecture modeling and analysis tool. As shown in Figure 4.19, SAMAT is designed using the Model-Vew-Control pattern.

1. The model of SAMAT includes a hierarchical layer of SAM compositions that builds the SAM model in Figure 4.18. It also include the functionalities of generating flat Petri net model and conjunctions of FOLTL formulas for analysis purpose.

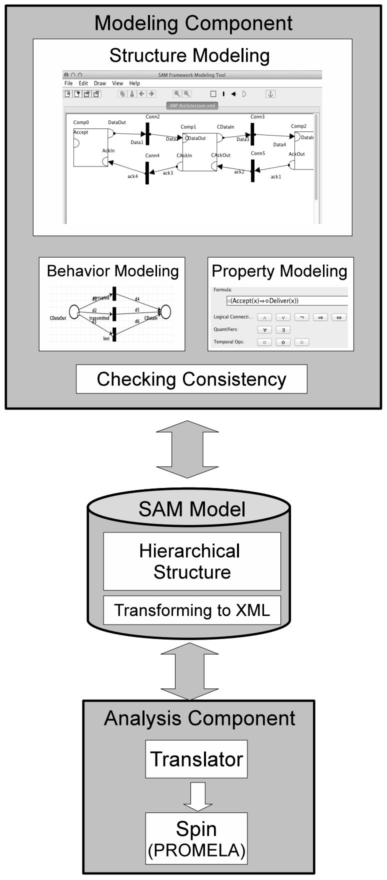


Figure 4.18: The Functional View of SAMAT

2. The graphical interface of SAMAT is developed using Java Swing API as it provides full GUI functionalities and mimics the platform it runs on. It consists of a SAM composition editor, a PIPE+ editor, a FOLTL editor and an analysis displayer. The composition editor is used for modeling the SAM compositions into a hierarchical structure; the PIPE+ editor is used for modeling the behaviors of SAM model via PrT nets; the FOLTL editor is used for defining the properties into FOLTL formulas; the analysis displayer is used for showing the analyzing result generated by SPIN [60].

3. The controller is comprised of composition controllers, a XML transformer and a PROMELA translator. The composition controllers provide options to specify detailed properties of a SAM composition; the XML transformer transforms SAMAT model into hierarchical XML format for storage purpose; the PROMELA translator translates the generated flat Petri net model and the conjunction of FOLTL formulas into PROMELA language, which is the input to SPIN.

SAMAT integrates two external tools: PIPE+ [83] for behavior modeling and SPIN [60] for model analysis.

4.2.6.3 SAM Hierarchical Model in SAMAT

SAMAT stores the SAM model in a hierarchical way. As we can see in Figure 4, the SAM model's data structure are in layers. In addition to the SAM compositions, the top layer contains a sub-composition model called sub-layer that has the same elements of the parent one except the bottom layer, which instead of a sub-composition model, is a Petri nets model. Therefore, each sub-composition model also has allocated space for its own sub-composition and a user can model arbitrarily number of levels by this recursive layer structure.

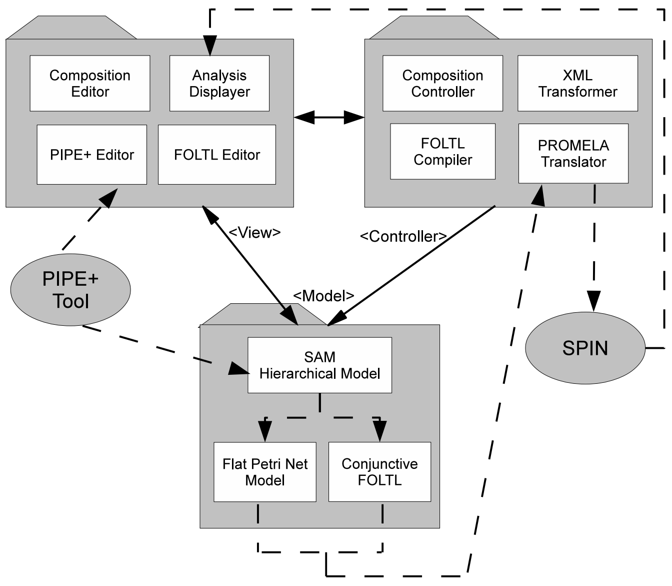


Figure 4.19: The Design View of SAMAT

The Petri nets layer in the bottom of Figure 4.17 is the behavior model of its parent composition. In this case, it is a high-level Petri net formalism modeled in PIPE+ editor. Once a Petri net model is created, it is transformed and saved in XML format and is appended to its parent SAM composition.

In this way, SAMAT is capable of storing hierarchical layers of the SAM architecture model. SAMAT supports a top-down approach to develop a software architecture specification by decomposing a system specification into specification of components and connectors and by refining a higher level component into a set of related sub-components and connectors at a lower level. From the SAMAT's GUI, each component provides options for a user to define a sub layer or a behavior model. If the sub-layer is selected, a new tab of drawing canvas is built in the mainframe editor with designated title of "parent name :: sub composition name". Furthermore, if the sub composition can be further decomposed, another new tab will be built. If the behavior model option is selected, PIPE+ is triggered for the user to build a

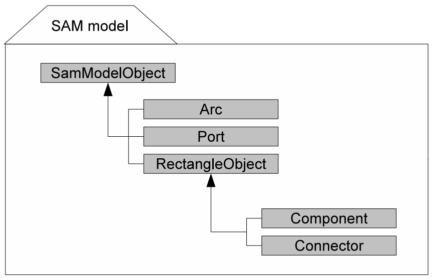


Figure 4.20: The Architecture of SAM Model Package

behavior model using Petri nets. Therefore, the top-down decomposition process is straightforward.

4.2.6.4 Inheritance Class Design in SAMAT

The design of the SAM model package in SAMAT must include all the SAM's graphical elements (i.e. components, connectors, arcs and ports). Figure 5 illustrates the class design hierarchy diagram. For the reusability and extensibility purpose, all of the SAM graphical elements are derived from SamModelObject class that holding basic features of a graphical object such as position, label and handler. Furthermore, Arc, Port and RectangleObject classes are inherited from SamModelObject, and Component and Connector classes are inherited from RectangleObject class.

4.2.6.5 FOLTL Editor

One of the underlying formalism in SAMAT is FOLTL. The vocabulary and models of FOLTL used in SAMAT are based on the high-level Petri net formalism and follow the approach defined in [79]. An example FOLTL formula is $\Box((x>y) \Rightarrow \Diamond\ (b=1))$, where variables are restricted to the underlying behavior models' arc variables. Since in each composition, SAMAT integrates a FOLTL formula editor where a user can specify system properties, the composition-level property specification is obtained by conjoining the property specifications of all components and connectors. The

FOLTL compiler checks the syntax of a FOLTL formula and the translator generates constraint code in PROMELA.

4.2.6.6 PIPE+

The other formalism in SAMAT is PrT nets, which are a class of high-level Petri nets. We integrate an existing open source high-level Petri net tool PIPE+ [83] to specify the behavior model of the SAM architecture. PIPE+ is capable of specifying and simulating high-level Petri nets proposed in [6]. SAMAT leverages PIPE+'s editing mode in which a high-level Petri net behavior model can be developed graphically with dragging and dropping actions. The high-level Petri net model is comprised of:

1. A net graph consists of places, transitions and arcs.
2. Place types: These are non-empty sets restricting the data structure of tokens in the place.
3. Place markings: A collection of elements (tokens) associated with places. For analysis purpose, a bound of tokens' capacity on each place is necessary, so that verification run on SPIN can always stop.
4. Arc annotations: Arcs are inscribed with variables that contributes to the transition expression formula variables;
5. Transition conditions: A restricted first-order logic formula Boolean expression is inscribed in a transition. It is called restricted because the grammar doe not permit free variables.

With all of the above high-level Petri net concepts specified, the behavior model is formally defined and can be verified by model checking engines.

4.2.6.7 XML Transformer

SAMAT transforms a SAM structure model into a XML model based on its hierarchical structure; and then appends the high-level Petri net XML model generated by PIPE+ to it. In this way, the SAM structural and behavior models are complete and are stored and loaded via XML saver and loader.

4.2.6.8 An Example - Alternating Bit Protocol

The alternating bit protocol (ABP) [106] is a simple yet effective protocol for reliable transmission over lossy channels that may lose or corrupt, but not duplicate messages. It has been modeled and analyzed in [56]. ABP consists of a *Sender*, a *Receiver* and a *Channel*. The *Sender* sends messages through lossy *Channel* to the *Receiver* and the *Receiver* reply with confirm number also through the *Channel*. In this section, we present how to build a model for ABP in SAMAT.

Since ABP consists of a *Sender*, a *Receiver* and a *Channel*, we can naturally use three components to model them. Both *Sender* and *Receiver* connect with *Channel* and use *Channel* to communicate, thus we connect the components with connectors. The model is graphically built in Figure 4.21.

For each component, we need to specify its behavior models in PIPE+ graphical editor that integrated into SAMAT. For example in Figure 4.22, the behavior model is specified by PrT Net. The details of specifying PrT Net is the same as one mentioned before. Besides, for analysis reason, we need to choose the place from the behavior model for the port in the upper level component *Sender* as interface to communicate with other components. We denote them port place. In *Sender*, the port Accept is P_{Accept}, the port DataOut is $P_{DataOut}$ and the port ActIn is P_{ActIn}.

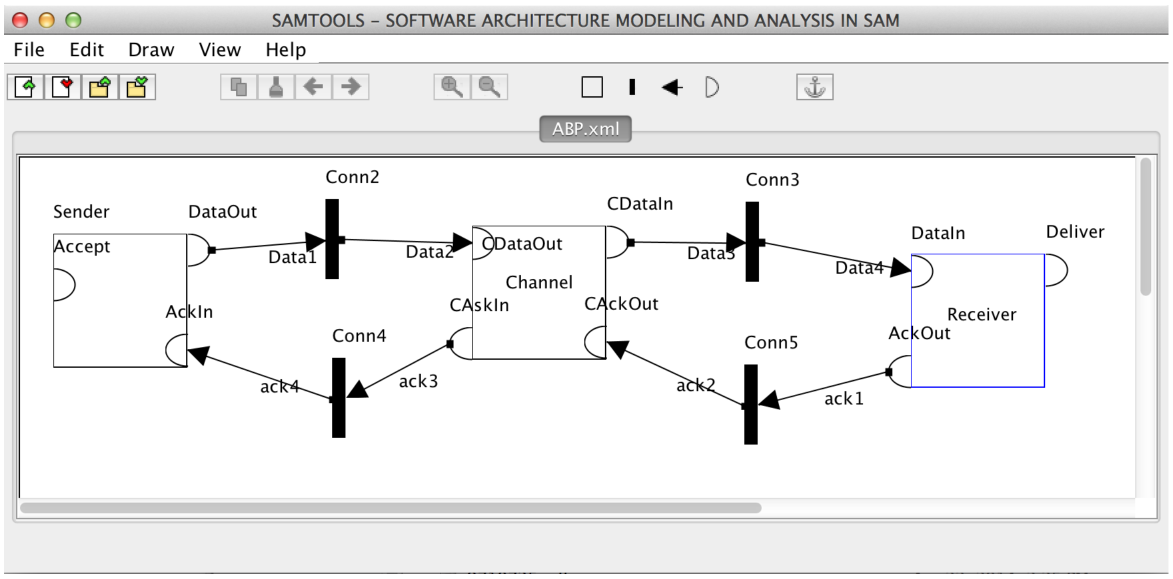


Figure 4.21: ABP Model in SAMAT

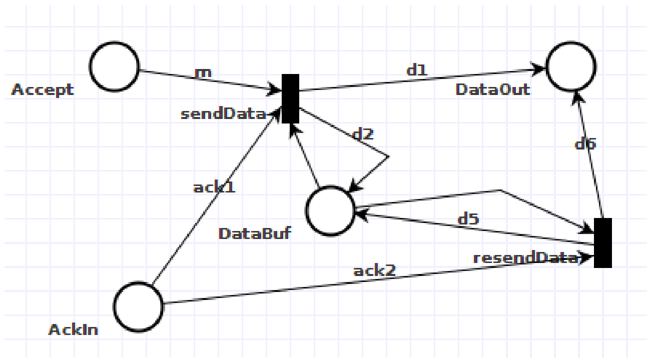


Figure 4.22: Behavior Model of ABP's Component Sender

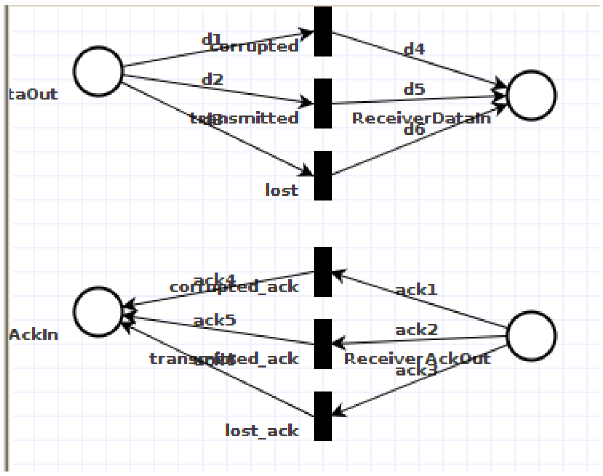


Figure 4.23: Behavior Model of ABP's Component Channel

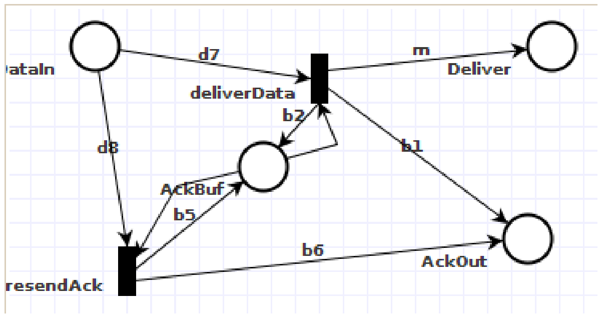


Figure 4.24: Behavior Model of ABP's Component Receiver

At last, since the connector in the top layer ABP model only pass messages but do not modify data, so the condition formula in the connector is $OutputMessage = InputMessge$. The message of connector's arc label is instantiated by tokens from port place.

CHAPTER 5 ANALYSIS

The goal of modeling a system is to investigate it more conveniently and systematically. In this chapter, we present three analysis methods in our framework for PrT Nets models, simulation, explicit state model checking and bounded model checking (BMC). Simulation reflects dynamic semantics of Petri Nets, which shows a behavior of a Petri net model. However, similar to test method for a software, simulation only explores one state transition sequence of a model a time. Thus, simulation is speedy but with less path checking coverage. On the other hand, explicit state model checking explores all the state transition sequence in a model, but since HLPNs' high level concepts brings in big complexity, this method sometimes becomes inefficient because of the state explosion problem. BMC only covers a certain level of state transition sequence, such as exploring paths within length k , but more efficient. BMC trades off some coverage for efficiency, but it becomes useful when we know the coverage predefined is enough.

5.1 Simulation

Petri nets models are executable, which means by making simulations, it is possible to visualize the behavior of the systems modeled by Petri nets. The behavior is reflected by token flows. Simulating a Petri net model for one step means firing a transition in the model, denoted as $M \xrightarrow{t} M'$, which means a token sent from t's input place to t's output place. A simulation run results in a state transition sequence $\varphi = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots$, where the net is modified by a sequence of transition firings.

HLPNs are executable as well [6]. Simulation in HLPNs is similar to low level ones as it reflects the behavior of a model with token flows. However, in HLPNs, tokens are not identical as they represent different data, its simulation algorithm need to consider the enabling of transition formula and the fairness of token to be fired in a place.

Currently, there are many tools equipped with simulator that can run simulation on specific HLPNs models. Such as CPN tools[76], Maria [86], Renew [78]. However, simulation go through only one path per execution, which means it does not provide guarantee to any verification property. Simulation often used in non-critical and fault tolerant systems. We implemented PIPE+ with a simulator to graphically specify and simulate PrT nets model. PIPE+ simulator can execute the PrT net step by step. The execution results in a sequence of fired transitions and generates a sequence of state markings, which can be used to investigate the behavior of the system. A simulator not only needs to execute the net model visually, but also has to ensure the correctness, fairness and good performance. The PIPE+ simulator has the following features.

5.1.1 Graphical Simulation

Since in a low level Petri net, tokens are just black dots flowing from one place to another and the animation is visible to the user. In contrast, tokens in HLPNs are complex structured data, and thus are inappropriate to be displayed graphically during the simulation. A user can view the result stepwise by looking into the contents of places and by checking the summary of the simulation. In PIPE+, to view the tokens in the Places, a user can open the place editing panel to display the value of tokens under the text area of token list. Furthermore, the firing history and summary is retained in PIPE+ by listing the fired transition name orderly and updated instantly after each transition firing.

5.1.2 Transition Occurrence Scheduling Algorithm

A scheduler is needed to coordinate the simulator's token flow strategy efficiently. Since the performance of the simulator is mostly affected by the checking of transition constraints, PIPE+ chooses the scheduling algorithm from [89] to minimize the recalculation of transition constraint checking. The idea is to keep track of disabled transitions discovered during the search of enabled transitions using the locality principle, that is an occurring transition only affects the marking of immediate neighboring places and hence enabling only a limited number of neighboring transitions. In the implementation, an unknown list and a disabled list are maintained. All transitions initialized as unknowns will be randomly picked and checked for enabling status. If the status of a transition is disabled, the transition will be moved to the disabled list. Upon the firing of a transition, the status the neighboring transitions in the disabled list will be changed to unknown. Therefore, using the disabled transition list avoids the unnecessary costly transition constraint checking.

5.1.2.1 Fairness of Picking a Token to Fire

Unlike low level Petri nets that tokens are identical black dots, in HLPNs, tokens are structured data whose values may be different. A place in HLPNs is a set of tokens with no order. Theoretical, if a number n of the tokens in place P can enable a transition T, one of the tokens is picked randomly to instantiate the transition condition and is consumed after firing the transition. However, a token may not able to be fired forever if the newest added tokens to P is fired every time checking T.

Because of this problem, we need to maintain the liveness of every token in P, thus in this simulation algorithm we maintain a FIFO list, that the newest token to P is always added to the tail of the list. To prevent the situation that a transition cannot be enabled as the head token in its input place cannot enable a transition, we move the head token to the tail every time a head token is checked by a transition condition and cannot enable the transition. In this way, every token got a chance to be checked and fired.

5.1.2.2 Instantiating a Transition Constraint

In high level Petri nets, tokens are meaningful data. In evaluating a transition constraint the variables are to be instantiated. Since a transition may connect to a number of input places, where each place contains a list of tokens, to see whether the transition is enabled or disabled,all the possible combinations of instantiated tokens from its input places need to be checked. For example, if there are three input places and each place has 3 tokens, the total number of combinations are $3 \times 3 \times 3 = 27$. If one of the three input places is a set, it contains only one abstract token. So the combinations reduce to $3 \times 3 \times 1 = 9$.

5.1.2.3 Enabling and Firing a Transition

A transition enabling calculation process is shown in Figure 5.1: In step (1), each token in the connected place is firstly bounded to the connected arc variable; a pair, {variable, token}, is fetched into a symbol table of the transition (note the pair with output variable's token value is temporarily empty and to be filled by the result of the expression calculation). In step (2), the input variables in the

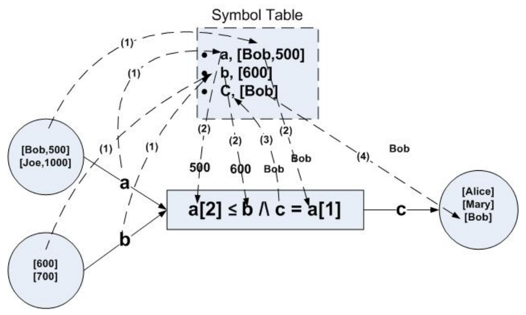


Figure 5.1: An Enabled Transition Formula Calculation Process

transition expression can locate token values through the symbol table. In step (3), after transition expression calculation, the output variables are assigned the resulting values and the symbol table's output variable pairs are filled with the values. In step (4), the output pairs' tokens are added to the connected output places according the arc's variables. For example, c is instantiated with[bob] from the symbol table and token [bob] is added to the output place.

5.1.3 Parser and Interpreter

Because logical formulas need to be parsed and interpreted, we build a compiler with a parser and an interpreter for the restricted first-order logic. The parser includes a scanner, which is built by a lex file and generated by jflex 1.4.3 [3]. A BNF grammar is built in cup file and generated by leveraging the tool jcup v11 [2]. Since a transition formula does not explicitly separate pre and post conditions, while only pre-conditions need to be calculated to determine the transition is enabled or not, the interpreter has to differentiate pre and post conditions. The key is to identify the post-condition, which usually starts with an output variable equaling to an expression, for example, in (1), $C' = C \setminus \{b\} \cup \{[a[1], c[2]]\}$ is a post-condition because C' is an output variable. Therefore the interpreter checks a clause with an "=" operator, if the left hand side of "=" is an input variable, this clause is a pre-condition and "=" is interpreted as relational equality operator; on the other hand, if it is an output variable, the clause is a post-condition and "=" means an assignment.

5.1.4 The Complete Simulation Process

1. All transitions in the net are initially stored in an unknown list, and a disabled list is initialized to be empty;
2. A transition is randomly selected from the unknown list and its constraint is checked;
3. During the checking process of the selected transition, all the connected arcs and places of the transition are found;
4. Combinations of tokens from the transition's input places are orderly chosen to fill in its symbol table. Since symbols in symbol table are pairs of [key, object]. The keys are from arcs label and the objects are regular tokens. If the input place has a power set type, an abstract token is sent as an object; otherwise only the first token is sent and the remaining tokens are still in place. The object of any key from an output arc is empty ;
5. The parser checks the syntactical correctness of a formula. Then, the interpreter evaluates the formula and returns a boolean result: if it is true, the transition is enabled and fires immediately; if it is false, the transition is not enabled under the current marking, the tokens in symbol table will go back to the input places. If all the combinations of input tokens cannot enable the transition, the transition is moved into the disabled list.
6. After firing a transition, the tokens in the symbol table are sent to the output places according to the variables of output arc sand are added to the tail of

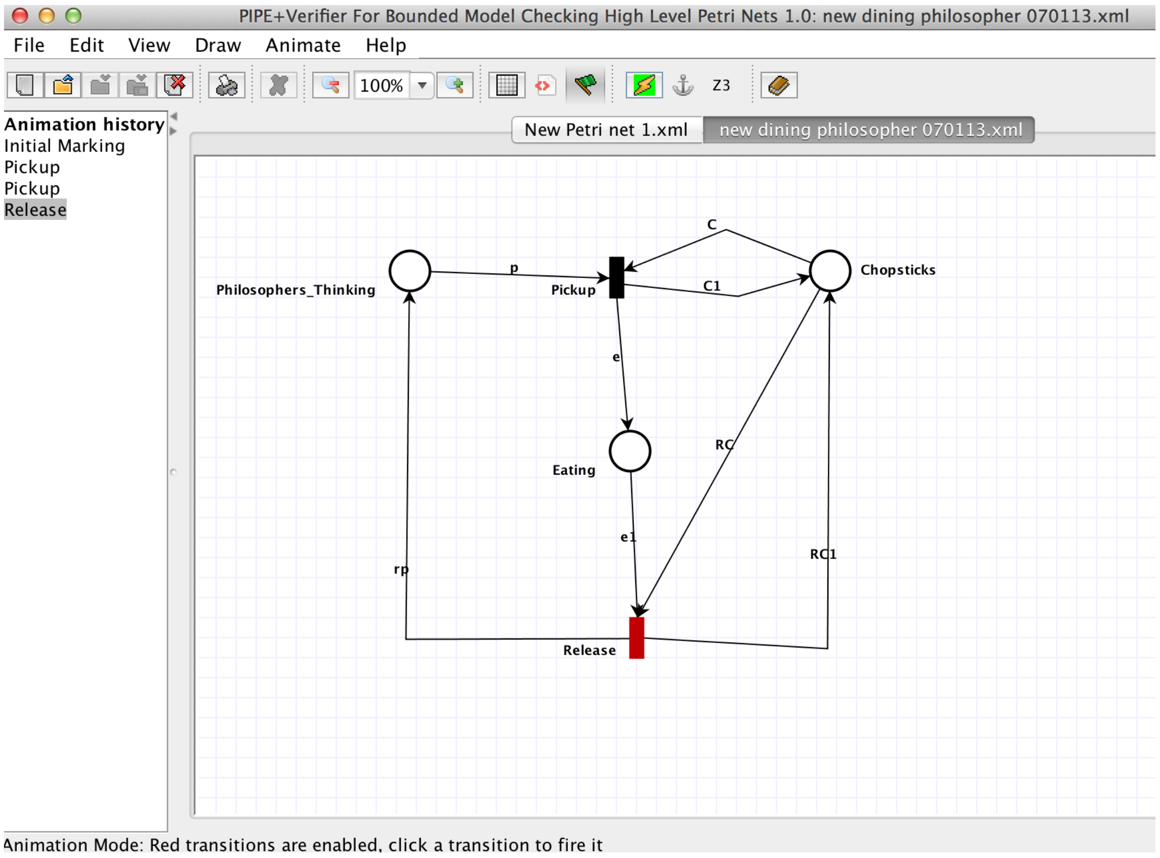

PIPE+Verifier For Bounded Model Checking High Level Petri Nets 1.0: new dining philosopher 070113.xml
File
Edit
View
Draw
Animate
Help
100%
Z3
New Petri net 1.xml
new dining philosopher 070113.xml
Animation history
Initial Marking
Pickup
Pickup
Release
Philosophers_Thinking
p
Pickup
C
C1
Chopsticks
e
Eating
RC
e1
RC1
rp
Release
Animation Mode: Red transitions are enabled, click a transition to fire it

Figure 5.2: Simulate Dining Philosophers Problem in PIPE+

output places' token list. The scheduler moves dependent transitions from the disabled list to the unknown list. Return to step 2.

7. In step 2, when unknown list is empty, the simulation process ends.

5.1.5 Simulating 5-Dining Philosophers Problem in PIPE+

After the Dining Philosophers problem specified completely, user can click the green flag toggle button to get into the simulation mode. By clicking the button of randomly fire a HLPN transition, PIPE+ will check all the transition and fire one of them. The fired transition will turn red indicating it is fired and will be added to the firing history queue.

In Figure 5.2, a transition firing sequence is shown in left column $\sigma = T_{Pickup} \to T_{Pickup} \to T_{release}$. By firing the first two T_{Pickup} transition, philosopher 0 and 2 picked up chopsticks and P_{Phil_Eating} now has token $\{[0], [2]\}$. After firing the third transition $T_{release}$, philosopher 0 released chopsticks and goes to thinking status. Thus, the final marking after firing σ in P_{Phil_Eating} is philosopher number 2.

5.2 Explicit State Model Checking

We call this explicit state model checking to differentiate with bounded model checking. Explicit state model checking is actually a classic model checking method, thus we sometimes use model checking for short. Model checking [67] refers to exhaustively and automatically checking a system model and see whether a property is satisfied on this morel or not. As model checking verifies a model completely, it can be used to prove whether a model N meets a specification f, denoted as $N \models f$. Another advantage of this method is the verification process can be automated, thus compared to manual analysis, it improves the efficiency of verification process and avoid potential mistakes made by human beings.

5.2.1 Overview

Investigating a system by simulation is similar to test a system, it is convenient, efficient and straightforward but it cannot be used to verify properties on all the behaviors of a system. For example in Figure 5.2, simulate the model may generate an infinite sequence $T_{Pickup} \to T_{Release} \to T_{Pickup} \to T_{Release} \cdots$, but never find a state where no philosopher can get two chopsticks at the same time. Thus, formal verification is required, especially when we already have a formal model (PrT Nets model). Model checking [30] is a verification process that exhaustively and automat-

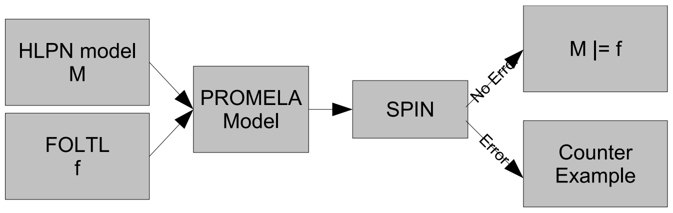


Figure 5.3: Model Checking PrT Nets Process

ically check whether a model M meets its given specification f, denoted as $M \models f$. Thus, using model checker as backend engine to verify PrT Nets is a natural way to prove properties on a model.

Our framework integrates a model checker SPIN [60] to verify specified properties defined in linear temporal logic [79] formula. A HLPN model combined with LTL formula is first translated into PROMELA (Process or Protocol Meta Language) model [60], which is input language to SPIN, and then checked by SPIN. If a the property formula is satisfied, then $M \models f$ is proved; otherwise, a counterexample is produced. The model checking process is shown in Figure 5.3.

We implemented a tool SAMAT that is able to automate the whole analysis process. SAMAT automatically translate HLPN model and FOLTL formula into a PROMELA model and leverages its integrated SPIN model checker to check the model.

5.2.2 The SPIN Model Checker and PROMELA

SPIN [60] is a model checker for automatically analyzing finite state concurrent systems. SPIN has been used to check logical design errors in distributed systems, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. A concurrent system is modeled in the PROMELA (Process or Protocol Meta Language) modeling language [60] and properties are defined as linear temporal logic formulas. SPIN can automatically examine all program behaviors to decide whether the PROMELA model satisfies the stated properties. In case a property is not satisfied, an error trace is generated, which illustrates the sequence of executed statements from the initial state. Besides, SPIN works on-the-fly, which means that it avoids the need to pre-construct a global state graph or Kripke structure, as a prerequisite for the verification of system properties.

A SPIN model in PROMELA consists of three types of objects: processes, message channels and variables. Processes specify the behavior, while the channels and variables define the environment for processes to run. The processes are global objects and can be created concurrently, which communicate via message passing through bounded buffered channels and via shared variables. Variables are typed, where a type can either be primitive or composite in the form of arrays and records.

5.2.3 Translating PrT Net Models to PROMELA

The translation process maps a high-level Petri net model to a PROMELA model. The resulting PROMELA model needs to capture the concepts of HLPNs defined in [6] and preserves the dynamic semantics of a given HLPN model. The PROMELA program's major parts are definitions of places and place types, transition enabling and firing inline functions, a main process and an init process that defines the initial marking.

The translation map is shown in Table 5.1:

- **Translating places**: We predefine each message type (place type) into a new structure. Places and place types are mapped to PROMELA's buffered channels and predefined message types. In addition, structured tokens are

Table 5.1: Mapping from High-level Petri net to PROMELA

High-level Petri net	**PROMELA**
Place	Channel
Place Type	Typedef Structure
Token	Message
Transition	Inline Function
Initial Marking	Message in Channel

mapped to typed messages in PROMELA. Because SPIN verifies a model by exhaustive state searching, we set bounds to limit the number of tokens in places. The bounds are then mapped to the lengths of the channels. A sample PROMELA program resulted from place translation is shown below:

```
#define Bound_Place0 10
typedef type_Place0 {
        int field1;
        short field2
};
chan Place0 = [Bound_Place0]
        of {type_Place0};
```

Translating transitions: In PrT Nets, a transition expression consists of a precondition and a postcondition. The precondition defines the enabling condition of the transition, and the postcondition defines the result of the transition firing. Each precondition and postcondition are translated into two inline functions, is_enabled() and fire(), respectively. To check the precondition of a transition expression, we first consider a default condition that whether each of the input place has at least one token by checking the emptiness of each mapped channel. The evaluation process includes non-deterministically receiving a message from an input channel to a local variable, and instantiating and evaluating the expression. A sample PROMELA program from transition translation is shown below:

```
inline is_enabled_Transition0{
}
inline fire_Transition0{...}
inline Transition0{
        is_enabled_Transition0();
        if
        :: Transition0_is_enabled
                -> fire_Transition0
        :: else -> skip
        fi
}
```

- **Defining main process**: The dynamic semantics of a Petri net is to non-deterministically check and fire enabled transitions, so the main process is defined by including all the transitions in a loop, "do ... od". Since PROMELA has finer granularity and a transition firing process includes multiple sub-steps, we aggregate them into an atomic construct. A sample PROMELA program for an overall PrT net structure is shown below:

```
proctype Main(){
        ...
        do
        :: atomic{Transition0}
        :: atomic{Transition1}
        od
}
```

- **Defining initial marking**: PROMELA has a special process "init{}", which is often used to prepare the initial state of a system. Therefore, the initial marking is defined in init process by declaring typed messages and send them into buffered channels. A PROMELA prototype is shown below:

```
init{
        type_Place0 P0;
        P0.field1 = 1;
        P0.field2 = 0;
        Place0!P0;
        run Main()
}
```

- **Using basic data types**: The basic data types supported in PIPE+ are integer and string, which are mapped to "int" and "short" in PROMELA respectively.
- **Handling non-determinism**: In PrT Nets, tokens are meaningful data and usually different from each other and thus different firing orders result in different markings. Therefore, a non-deterministic inline function is defined and is called to non-deterministically pick a token from an input place each time a precondition is evaluated.
- **Supporting power set**: PIPE+ supports quantifiers in restricted first-order logic formulas in transition expression, the domain of each quantified variable is a list of tokens as a power set contained in a place. For this type of places, we are not dealing with one message but all the messages in the channel, we do

not put all received messages into a local variable but directly manipulate the channel. The strategy is when the first message is received from the channel, it is used and then is sent back immediately.

5.2.4 FOLTL Formula

Since SPIN recognize LTL as inputs, and PrT Nets specify properties using FOLTL formula, it is natural to covert FOLTL to LTL without quantifier $\exists$ and $\forall$. Since first-order logic is undecidable, but as the FOLTL formula we specified in PrT Net model has restricted domain, when come across quantifiers, the translation need to instantiate the subformula behind quantifiers. Therefore, the instantiated formula can be wrapped by following PROMELA's syntax:

```
ltl f{/*formula*/}
```

5.2.5 Translation Correctness

The translation correctness is ensured by the following completeness and consistency criteria [117, 10]

Let N be a given HLPN and P_N be the resulting PROMELA program from the translation.

- **Completeness** Each element in N is mapped to P_N according to the mapping rules listed in Table 1.
- **Consistency** The dynamic behavior of N is preserved by P_N as follows:
  - A marking of N defines the current state of N in terms of tokens in places, our place translating rule correctly maps each marking into a corresponding state in P_N;

– The initial marking of N is correctly captured by the initial values of variables in the init{} process of P_N;

– The enabling condition and firing result of each transition t in N is correctly inline functions "is_enabled_Transition_i" and "fire_Transition_i" respectively;

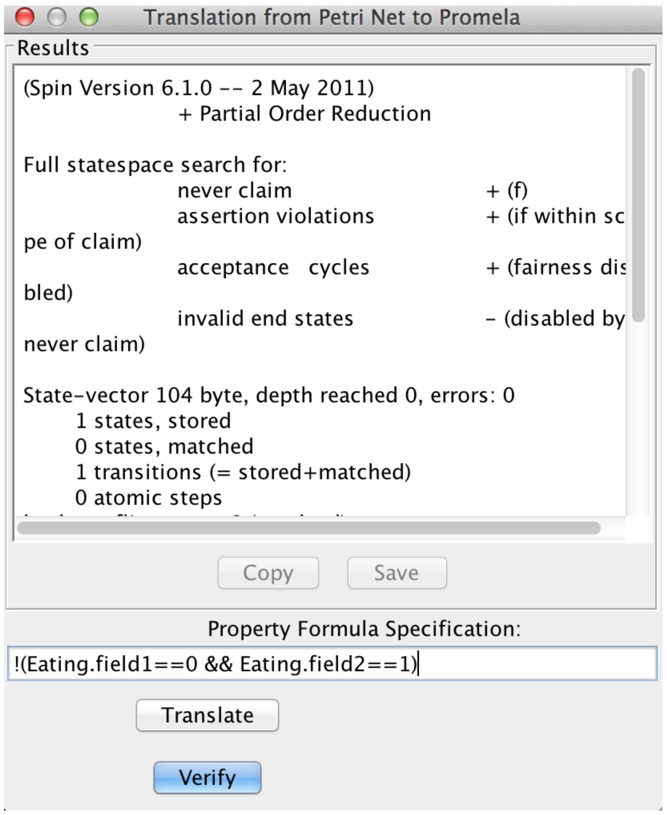
– The atomicity of enabling and firing a transition in N is preserved in P_N by language feature "atomic{}".

– An execution of N is firing sequence $\tau = M_0 t_0 M_1 t_1 ... M_n t_n ...$, where $M_i (i \in nat)$ is a marking and $t_i (i \in nat)$ denotes the firing of transition t_i. Each execution is correctly captured by the construct "do ... od" in the "Main" PROMELA function, which produces an equivalent execution sequence $\sigma = S_0 T_0 S_1 T_1 ... S_n T_n$, where $S_i (i \in nat)$ is a state and T_i denotes the execution of inline function "Transition_i".

The proofs of the completeness and consistency are straightforward and can be found in [117, 10] .

5.2.6 Verification using SPIN

The two inputs to SPIN are a PROMELA model and a property formula. SPIN performs verification by going through all reachable states produced by the model behaviors to check the property formula. If the property formula is unsatisfied, it produces a trail file indicating the error path. SPIN also provides a simulation function to replay the trail file so that any error path that leads to the design flaw can be visualized. Our framework encapsulates the verification process in SPIN and displays the verification result as well as captured error path by SPIN.



```
(Spin Version 6.1.0 -- 2 May 2011)
                    + Partial Order Reduction

Full statespace search for:
                    never claim                    + (f)
                    assertion violations           + (if within sc
pe of claim)
                    acceptance   cycles            + (fairness dis
bled)
                    invalid end states             - (disabled by
never claim)

State-vector 104 byte, depth reached 0, errors: 0
       1 states, stored
       0 states, matched
       1 transitions (= stored+matched)
       0 atomic steps
```

Property Formula Specification:

```
!(Eating.field1==0 && Eating.field2==1)
```

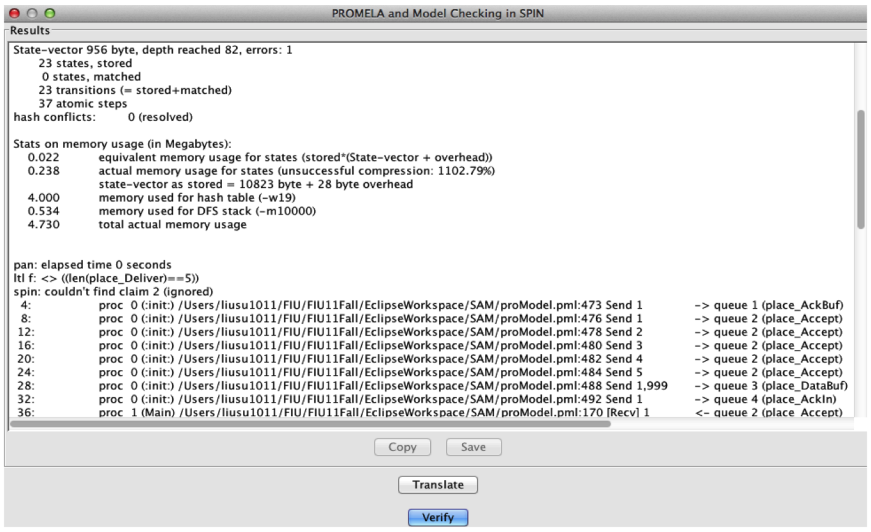
Figure 5.4: Model Checking 5-Dining Philosophers Problem

5.2.7 Checking 5 Dining Philosophers Problem

The 5 dining philosophers problem is checked with the property neighbor philosophers are not eating at the same time. We set the bound of each place as 5. The result in Figure 5.4 says the property is satisfied with 0 error; which took 2 seconds and 2.539Mb memory.

5.2.8 Checking Seabed Rig Robotic Workflows

The Seabed Rig robotic workflow specified in Figure 4.15 is checked by SPIN with properties that two robots finished their tasks and moved back to the initial position.



```
State-vector 956 byte, depth reached 82, errors: 1
      23 states, stored
       0 states, matched
      23 transitions (= stored+matched)
      37 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.022         equivalent memory usage for states (stored*(State-vector + overhead))
    0.238         actual memory usage for states (unsuccessful compression: 1102.79%)
                  state-vector as stored = 10823 byte + 28 byte overhead
    4.000         memory used for hash table (-w19)
    0.534         memory used for DFS stack (-m10000)
    4.730         total actual memory usage


pan: elapsed time 0 seconds
ltl f: <> ((len(place_Deliver)==5))
spin: couldn't find claim 2 (ignored)
  4:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:473 Send 1        -> queue 1 (place_AckBuf)
  8:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:476 Send 1        -> queue 2 (place_Accept)
 12:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:478 Send 2        -> queue 2 (place_Accept)
 16:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:480 Send 3        -> queue 2 (place_Accept)
 20:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:482 Send 4        -> queue 2 (place_Accept)
 24:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:484 Send 5        -> queue 2 (place_Accept)
 28:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:488 Send 1,999    -> queue 3 (place_DataBuf)
 32:              proc  0 (:init:) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:492 Send 1        -> queue 4 (place_AckIn)
 36:              proc  1 (Main) /Users/liusu1011/FIU/FIU11Fall/EclipseWorkspace/SAM/proModel.pml:170 [Recv] 1        <- queue 2 (place_Accept)
```

Figure 5.5: Model Checking the ABP in SAM Tool

However, since the model is a small example, and no errors can be found in this workflows with two robots.

5.2.9 Checking Alternate Bit Protocol in SAMAT

In Figure 5.5, the top layer of ABP model in SAMAT consists of three components and four connectors. The first component “Sender” has a behavior model shown in Petri net. On the right, it shows the FOLTL editor to editing formula <>(Deliver(m) = 5). After the modeling process, SAMAT automatically generates PROMELA code as an input for SPIN and displays the model checking result after SPIN finished model checking. In this case an error is found, the replayed simulation on the error path is shown below the model checking result. The error indicates the ABP specification model in [56] is incorrect. A deadlock state (a none final state such that none of the transitions are enabled) can be reached when an acknowledgement message was corrupted in the channel and a resend message successfully reached the receiver's DataIn place. This discovery highlights the great benefits and usefulness of SAMAT.

5.3 Bounded Model Checking

5.3.1 Overview

Bounded Model Checking (BMC) with satisfiability solving [18, 26] was proposed as an alternative approach to address the state explosion problem in the traditional model checking approach. BMC does not explore the whole state space compared to the explicit state model checking, so that it does not face the state explosion problem.

According to [18], BMC is best for checking safety (reachability) properties. In BMC, a feasible symbolic execution of a transition system and the negation of some safety properties are translated into a propositional formula ϕ, which is checked by a satisfiability solver. If the formula ϕ is satisfiable, a counter example is found and thus the safety property does not hold. On the other hand, if the formula is not satisfiable up to a pre-defined upper bound k, the safety property holds up to k. Thus this approach is not a complete technique for safety property analysis. The threshold k is hard to determine according to [26], which is as hard as explicit state model checking. But in real world applications, sometimes we know k beforehand. Although this approach is not a complete technique for safety property analysis, it has been shown to be very effective in detecting the violation of safety properties in many real-world applications.

In this section, BMC PrT Nets is achieved by an automatic approach that encoding PrT Nets model into a formula ϕ and then checked by SMT solver. Since encoding a low level Petri net model into a propositional logic formula and then checked by SAT solver is straightforward, but encoding a HLPN model is not since PrT Nets using structured data and algebraic expressions to define functionality. In recent years, great progress has been made on satisfiability modulo theories (SMT) [35, 90] solvers that can check the satisfiability of a subset of first-order logic formulas with a variety of underlying theories including linear arithmetic, difference arithmetic and arrays. These SMT solvers are expressive enough to represent the data and algebraic expressions in PrT Nets naturally. Furthermore, SMT solvers are becoming more efficient according to the annual competitions results from SMT [16], and have been successfully integrated into verification tools such as CBMC [11], SLAM2 [14], and VS3 [103]. Therefore, BMC PrT Nets is promising and an automatic approach is needed.

In this framework, this automatic method for BMC is defined and implemented. PrT Nets models are encoded through the theory of set [77] that has been integrated to some SMT solvers, where a place can have zero or more tokens. Similar to BMC, this method specifies a k value before checking, which defines the upper bound of transition firing actions (state changes). For each negated safety property reached within k steps, a transition firing sequence leading to an error state is generated. However, this method is incomplete because the upper bound k is often not given in real applications. Reference [28] discussed the complexity of finding a complete threshold. [18] shows BMC can check formulas in ACTL* [44].

A tool called PIPE+Verifier is implemented to support this method and automate the BMC process. After a HLPN model and its specified properties are defined, PIPE+Verifier automatically convert the HLPN model and the negated properties into a first-order logic formula. The formula is then checked by an SMT solver Z3 [36]. Figure shows the BMC process.

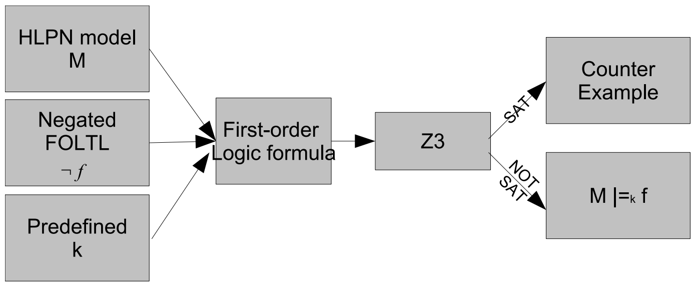


Figure 5.6: BMC HLPN Process

5.3.2 Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) [35] support a combination of theories such as bit-vectors, rational and integer linear arithmetic, arrays, and uninterpreted functions. SMT solvers are the extensions of satisfiability (SAT) solvers and directly applicable to the decision problems expressed in first-order logic formulas with respect to the multiple background theories.

For example, an SMT solver can decide whether a formula in the theory of linear arithmetic is satisfiable:

$$(x + y \leq 0) \wedge (\neg b \vee a \wedge (y = 0)) \wedge (x \leq 0)$$

where x, y are integer variables and a, b are Boolean variables. If the formula is satisfiable, the SMT solver returns a variable assignment satisfying the formula. Some important high level theories supported by SMT solvers are listed below as the foundation of our method.

- **Arrays:** The theory of arrays [105, 12] in SMT solvers are different from the ones in standard programming languages. In SMT, an array's size can be infinite. There are two built in functions: $\mathit{select} : \mathit{ARRAY} \times \mathit{INDEX} \rightarrow \mathit{ELEM}$ and $\mathit{store} : \mathit{ARRAY} \times \mathit{INDEX} \times \mathit{ELEM} \rightarrow \mathit{ARRAY}$ where

$ARRAY$, $INDEX$, $ELEM$ are the sorts of the array, the index of the array and the elements in the array.

- **Tuples:** The theory of tuples [77] supports a data structure with a list of components and access to individual components by projection.
- **Sets:** A set is a collection of objects. Reference [77] has defined a set theory, which has been implemented in several SMT solvers [17]. The theory of sets in SMT solvers supports a list of set operations including set member $\in$, set subset $\subseteq$, set union $\cup$, set intersect $\cap$ and set difference $\setminus$.

5.3.2.1 Z3

In recent years, the efficiency of SMT solvers has been greatly improved. An annual SMT competition is held every year [17] and the participants include CVC4 [15], Z3 [36], MathSAT [25], Opensmt [22], and Yices [41]. Among them, Z3 [36], developed by Microsoft Research Institution, is reported to have the largest number of users and supports almost all the popular SMT background theories such as rational and integer arithmetic, bit-vectors, array theory, and set theory. In addition, Z3 has been adopted as the backend verification engine for a variety of tools, such as VS3 [103], SLAM2 [14] and CBMC [11]. Z3's developing team provides api and documentation for different programming languages (C, C++, .NET, Python). Therefore, we have selected and integrated Z3 into our tool as the backend satisfiability solving engine.

5.3.3 General Idea of BMC using SMT Solver

In BMC, a logic formula ϕ_k is constructed from a given M_k, including the initial state I and unrolled transition relations T, and some negated safety properties f. Since transition T in ϕ_k is unrolled k times, the length of ϕ_k is dependent on k. The

DEF

s : STATETUPLE

ASSERT

$$
\begin{aligned}
&Initial\_marking(s_0)\\
&\wedge \bigwedge_{i=0}^{k-1} Transition(s_i, s_{i+1})\\
&\wedge \bigvee_{i=0}^{k} Negated\_property(s_i)
\end{aligned}
$$

CHECK

Figure 5.7: SMT context for bounded model checking

logic formula ϕ_k is represented in equation 5.1:

$$\phi_k \doteq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} f(s_i) \tag{5.1}$$

where $I\,(s_0)$ is the characteristic function of the initial state, $T\,(s_i,\, s_{i+1})$ is the characteristic function of the transition relation, and $f(s_i)$ represents the negated safety property in unrolled state s_i $(0 \leq i \leq k)$. If ϕ_k is satisfiable, there is a firing sequence or a state transition path from the initial state $I(s_0)$ to a state s_i $(0 \leq i \leq k)$ that satisfies f, thus violates the safety property. Otherwise, the safety property holds in M within k transition firings.

The general SMT logic context for BMC is shown in Figure 5.7.

5.3.4 Represent PrT Nets in SMT Context

Our goal is to translate a given HLPN model to a logic formula shown in Figure 5.7, and then use an SMT solver to check its satisfiability.

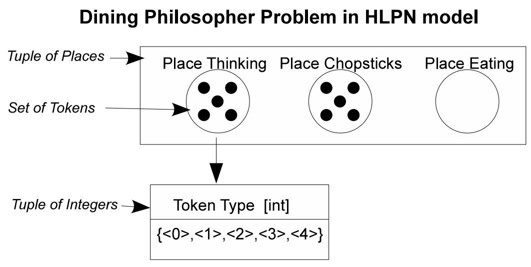


Figure 5.8: An inner view of dining philosophers problem in HLPN model

Table 5.2: High level Petri net elements mapped to SMT theory

HLPN Elements	**SMT Theory**	**In PIPE+Verifier**
HLPN Model	Tuple (Places)	STATETUPLE
Place Type	Set (Tokens)	SETiSORT
Token Type	Tuple (Integer or String Values)	DTiSORT
Primitive Data	Integer or String	INTSORT

5.3.4.1 Define States in SMT Context:

In HLPN, a state s_i is defined by a marking that is a distribution of tokens in places. Each place can contain 0 or more tokens (the number may be bound or unbound) and tokens can be structured data. To define a state in SMT context, a hierarchical layered data structure is constructed. A state s_i is defined by a tuple whose components are places: $s_i \doteq< p_0, p_1, \ldots, p_n >$. Each place p_j $(0 \leq j \leq n)$ is defined by a set containing $m \geq 0$ tokens: $p_j \doteq \{tok_0, tok_1, \ldots, tok_m\}$. Each token tok_k $(0 \leq k \leq m)$ is defined by a tuple of primitive data elements: $tok_k \doteq< e_0, e_1, \ldots, e_l >$. Figure 5.8 shows an inner view of a HLPN model. In Figure 5.8, the tuple of places is $<P_{Phil\_Thinking}$, $P_{Chopsticks}$, $P_{Phil\_Eating}>$, place $P_{Phil\_Thinking}$ has 5 tokens $\{\langle 0\rangle,\langle 1\rangle,\langle 2\rangle,\langle 3\rangle,\langle 4\rangle\}$ and each token tok_k has only one field $\langle ID\rangle$ whose type is Integer. In the SMT context, a state is defined by type STATETUPLE. The hierarchical data structure that constitutes STATETUPLE is shown in Table 5.2.

5.3.4.2 Define the Initial State

The $Inital\_marking\,(s_0)$ in Figure 5.7 is defined from the initial marking M_0 of a HLPN model. The state s_0 contains tokens of all the places marked in M_0.

5.3.4.3 Define Transitions in SMT Context

$Transition(s_i, s_{i+1})$ in Figure 5.7 is a binary relation between the current state s_i and the next state s_{i+1}. In BMC, the upper bound of the transition firing sequence is k, thus the state transition of ϕ_k is unrolled k times, denoted as $\bigwedge_{i=0}^{k-1} Transition(s_i, s_{i+1})$. A HLPN model consists with $n \geq 0$ transitions t_0, $t_1, \ldots, t_n$, and any one of them may fire if enabled, thus $Transition(s_i, s_{i+1})$ is represented by a disjunction of the transitions in the HLPN model $\bigvee_{j=0}^{n} t_j(s_i, s_{i+1})$. Transitions in ϕ_k is defined as an formula shown in Equation 5.2:

$$\bigwedge_{i=0}^{k-1}(Transition(s_i, s_{i+1})) = \bigwedge_{i=0}^{k-1}(\bigvee_{j=0}^{n} t_j(s_i, s_{i+1})) \tag{5.2}$$

Each transition in the HLPN model $t_j(s_i, s_{i+1})$ with a precondition (captured by c_0) and a post-condition (captured by c_1) are defined in an if-then-else structure $if\,c_0\;then\,c_1\;else\,c_2$, representing $(c_0 \implies c_1) \wedge (\neg c_0 \implies c_2)$. The translation schema is described below:

- If condition c_0:
  - Using set membership operation to check if each input place in s_i has at least one token;
    - In state s_i, each transition condition clause corresponds to a constraint;
  - Case True c_1:

    * Tokens are removed from t_j's input places of state s_i using set difference operation;
    * New tokens are added to t_j's output places of state s_{i+1};
    * Tokens in unrelated places in state s_i remain the same in those places in s_{i+1};
  - Case False c_2: tokens in all places in the next state s_{i+1} are the same as in the current state s_i.

5.3.4.4 Define Properties in SMT Context:

To check a safety property, we define $Negated\_property(s_i)$ as the negation of the safety property. If there exists a state s_i satisfies $Negated\_property(s_i)$, the safety property is violated at s_i. Thus, a disjunction of $Negated\_property(s_i)$ $0 \leq i \leq k$ is asserted in ϕ_k.

5.3.5 A Translation Example - Dining Philosophers Problem

Figure 2.3 illustrates a dining philosopher problem modeled in HLPN. The net consists of three places $P_{Phil\_Thinking}$, $P_{Chopsticks}$, P_{Phil_Eating} and two transitions T_{Pickup} and $T_{Release}$. All the places' token type is $\langle int \rangle$. $P_{Phil\_Thinking}$ and $P_{Chopsticks}$ are both initiated with markings that have five tokens $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$. T_{Pickup}'s transition condition is $p = c_1 \wedge (p+1)\%5 = c_2 \wedge e = p$. $T_{Release}$'s transition condition is $p = r \wedge c_1 = r \wedge c_2 = (r+1)\%5$.

From the dining philosophers HLPN model given in Figure 2.3, we obtain the following translation:

1. State Definition: As shown in Figure 5.9, a state consists of three places, $P_{Phil\_Thinking}$, $P_{Chopsticks}$ and P_{Phil_Eating}, which are defined as three sets in

DEF.

$$\begin{aligned}
STATETUPLE \equiv \langle & P_{Phil\_Thinking} : SETSORT, \\
& P_{Chopsticks} : SETSORT, \\
& P_{Phil\_Eating} : SETSORT \ \rangle \\
SETSORT \equiv \{ & set : DTSORT \} \\
DTSORT \equiv \{ & int : INTSORT \} \\
State \equiv \{ & s_0 : STATETUPLE \\
& s_1 : STATETUPLE \\
& \quad ... \\
& s_k : STATETUPLE \ \}
\end{aligned}$$

Figure 5.9: State definitions of 5-dining philosophers in SMT logic

$$\begin{aligned}
Initial\_marking(s_0) \ \equiv & P_{Phil\_Thinking}(s_0) = \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\} \\
& \wedge P_{Chopsticks}(s_0) = \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\} \\
& \wedge P_{Phil\_Eating}(s_0) = \emptyset
\end{aligned}$$

Figure 5.10: Initial State of 5-Dining Philosopher in SMT Logic

$$\begin{aligned}\bigwedge_{i=0}^{k-1} Transition(s_i, s_{i+1}) \equiv\ & (T_{Pickup}(s_0, s_1) \vee T_{Release}(s_0, s_1)) \\ & \wedge(T_{Pickup}(s_1, s_2) \vee T_{Release}(s_1, s_2)) \\ & \ \ ... \\ & \wedge(T_{Pickup}(s_{k-1}, s_k) \vee T_{Release}(s_{k-1}, s_k))\end{aligned}$$

```
$T_{Pickup}(s, s') \equiv$
  **IF** $p \in P_{Phil\_Thinking}$
  $\wedge l \in P_{Chopsticks}$
  $\wedge r \in P_{Chopsticks}$
  $\wedge p = l \wedge (p+1)\%5 = r$
  **THEN**
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking} - \{p\}$
  $\wedge P'_{Chopsticks} = P_{Chopsticks} - \{l\} - \{r\}$
  $\wedge P'_{Phil\_Eating} = P_{Phil\_Eating} \cup \{p\}$
  **ELSE**
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking}$
  $\wedge P'_{Chopsticks} = P_{Chopsticks}$
  $\wedge P'_{Ehil\_Eating} = P_{Phil\_Eating}$
```

```
$T_{Release}(s, s') \equiv$
  **IF** $p \in P_{Phil\_Eating}$
  **THEN**
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking} + \{p\}$
  $\wedge P'_{Chopsticks} = P_{Chopsticks} \cup \{p\}$
                                $\cup \{(p+1)\%5\}$
  **ELSE**
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking}$
  $\wedge P'_{Chopsticks} = P_{Chopsticks}$
  $\wedge P'_{Phil\_Eating} = P_{Phil\_Eating}$
```

Figure 5.11: State Transition of 5-Dining Philosophers in SMT Logic

$$\bigvee_{i=0}^{k} Negated\_property(s_i) \equiv (f(s_0) \vee f(s_1) \vee ... \vee f(s_k))$$
$$f \equiv P_{Phil\_Eating} = \{\langle 0\rangle, \langle 1\rangle\}$$

Figure 5.12: Property Definition of 5-Dining Philosophers in SMT Logic

STATETUPLE. All of the sets have the same set type $DTSORT$, and their element types are $INSORT$.

2. Initial state: place $P_{Phil\_Thinking}$ set contains five philosophers whose IDs are $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$ and place $P_{Chopsticks}$ has five chopsticks whose IDs are $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$. Therefore, as shown in Figure 5.10, both places at state s_0 contain five tokens.

3. State transition: $Transition$ is defined as $k-1$ transition steps that constrain pairs of consecutive states. Each transition step is an if-then-else structure that captures the pre-condition and post-condition of every local transition in HLPN. In Figure 5.11, s indicates the current state and s' indicates the next state.

4. Property definition: negated property $f(s_i)$ is state based, we need to define k disjunctions of error states. If one of $f(s_i)$ evaluates true, the whole formula is satisfiable and an error state s_i is reached. Figure 5.12 defines a simple negated safety property that the neighboring philosophers with ID $\{\langle 0\rangle, \langle 1\rangle\}$ can eat at the same time.

5.3.6 Building the Error Path

After checking a first-order logic formula ϕ_k is built, it is checked by an SMT solver. If the result is ϕ_k satisfiable, an instantiation of all the variables in ϕ_k is found, which means the error state (negated property satisfied state) is reachable. As the goal of our checking is to found the error as well as the counterexample to the error state. We need to find an error path that starts from the initial state and leads to the error state.

To rebuild the error path according to the variable instantiations provided by the SMT solver, we collect all the state tuples from s_0 to s_k. Since according to our translation schema, ϕ_k describes the transitions of a HLPN model, a state change is caused by a transition firing. The only differences between s_i to s_{i+1} are the markings of a set of places that is caused by a transition firing, thus the transition must connect to all modified places in the place set. Besides, in this modified places set, some of the place consumed tokens while others produced tokens, thus they are partitioned into two sets: an input place set and an output place set. After mapping back to the original net graph, the only transition t can be found through the two sets of places where the input place set are the input places of t and the output place set are the output places of t. Thus, after finding the transition t, a state transition $s_i \xrightarrow{t} s_{i+1}$ can be rebuilt that constructs part of the error path. The rest of the path can be rebuilt in a similar way.

For example in the Dining Philosophers problem in Figure 2.3, if the modified places set is $\{P_{Phil\_Thinking}, P_{Chopsticks}, P_{Phil\_Eating}\}$ where the input set is $\{P_{Phil\_Thinking}, P_{Chopsticks}\}$ and the output place set is $\{P_{Phil\_Eating}\}$ the transition must be T_{Pickup}. If the input set is $\{P_{Phil\_Eating}\}$ and the output place set is $\{P_{Phil\_Thinking}, P_{Chopsticks}\}$ the transition must be $T_{Release}$.

5.3.7 Bounded Model Checking in PIPE+Verifier

We have implemented an automated prototype tool called PIPE+Verifier to support our method and applied it to check relevant safety (reachability) properties in several benchmark problems modeled in HLPN. All experiments were conducted on a 32-bit Intel Core Duo CPU @3.0GHz box, with 4GB of RAM, running 32-bit Ubuntu.

5.3.7.1 Selected Benchmark Problems from Model Checking Contest @ Petri Nets

Model Checking Contest @ Petri nets (MCC) [73, 74] is held annually to assess Petri nets based formal verification tools and techniques. Petri net verification tools are compared with regard to the scaling abilities, efficiency, and property checking capabilities on selected benchmark problems. The benchmark problems are modeled in low level Petri nets and Colored Petri nets. However, none of the participating tools produced any promising results on checking colored Petri net models. We have translated several Colored Petri net models into PIPE+Verifier and analyzed their safety (reachability) properties. We have examined the scalability of our tool by changing parameters in the model and varying bound k. The running results are presented below.

5.3.7.2 Dining Philosophers Model

In the previous section, we presented the 5-dining philosophers model. We have selected the following two negated safety properties to check in PIPE+Verifier.

Table 5.3: Verifying Dining Philosophers Model

No. of Phils	**Prop.**	**Step Bound**	**Verdict**	**Prop. Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	5.3	5	unsat	yes	0.41	1.72
5	5.3	10	unsat	yes	79.93	9.97
5	5.3	15	N/A	N/A	N/A	N/A
5	5.4	2	sat	no	0.25	1.25
10	5.4	2	sat	no	0.76	1.62
20	5.4	2	sat	no	3.23	2.63

$$\Box\neg\left(marking\left(Eating\right)=4\wedge marking\left(Eating\right)=3\right) \tag{5.3}$$

$$\Box\neg(marking\left(Eating\right)\neq 4\wedge marking(Eating)=1\wedge marking\left(Chopsticks\right)\neq 4) \tag{5.4}$$

The scaling parameter is the number (up to 20) of philosophers. The experiment results are shown in Table 5.3. For property 5.3, PIPE+Verifier did not return a result when bound k reached 15 due to the exponential growth of the search space of Z3.

5.3.7.3 Shared Memory Model

In [24], a shared memory model involving P processors was given. These processors can access their local memories as well as compete for shared global memory using a shared bus. We have built a HLPN model based on the above shared memory model and checked the following two negated safety properties:

$$\Box\neg(marking\left(Ext\_Mem\_Acc\right)=\langle 1,5\rangle\wedge marking\left(Ext\_Bus\right)=1) \tag{5.5}$$

$$\Box\neg(marking\left(Ext\_Mem\_Acc\right)=\langle 1,5\rangle\wedge marking\left(Memory\right)\neq 4) \tag{5.6}$$

Table 5.4: Verifying Shared Memory Model

No. of Processors	**Prop.**	**Step Bound**	**Verdict**	**Prop. Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	5.5	5	unsat	yes	0.07	0.86
5	5.5	10	unsat	yes	0.3	1.54
5	5.5	15	unsat	yes	1.49	2.53
5	5.6	3	sat	no	0.75	1.80
10	5.6	3	sat	no	1.3	2.09
20	5.6	3	sat	no	13.05	4.35

The scaling parameter is the number (up to 20) of processors P. The results are shown in Table 5.4.

5.3.7.4 Token Ring

A token ring [39] model shows a system with a set of M machines connected in a ring topology. Each machine can determine if it has the privilege (the right) to perform an operation based on its state and its left neighbor.

We have modeled a token ring using HLPN and selected the following two negated safety properties to check:

$$\Box\neg(marking\,(State) = \langle 3, 0\rangle \wedge marking\,(State) = \langle 2, 4\rangle) \tag{5.7}$$

$$\Box\neg(marking\,(State) = \langle 3, 0\rangle \vee marking\,(State) = \langle 2, 4\rangle) \tag{5.8}$$

The scaling parameter is the number of machines M, which is up to 20. The results are shown in Table 5.5.

5.3.7.5 Abstract State Machine Model

In [110], a method for checking symbolic bounded reachability of abstract state machines was presented. An abstract state machine written in AsmL was translated

Table 5.5: Verifying Token Ring Model

No. of Machines	**Prop.**	**Step Bound**	**Verdict**	**Prop. Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	5.7	5	unsat	yes	0.32	1.34
5	5.7	10	unsat	yes	24.12	5.56
5	5.7	15	N/A	N/A	N/A	N/A
5	5.8	3	sat	no	0.09	1.01
10	5.8	3	sat	no	0.21	1.34
20	5.8	3	sat	no	0.86	2.03

Table 5.6: Running time of checking Count model

Model program	**Step bound**	**Verdict**	**Time of M.Veanes's Tool**	**Time of PIPE+Verifier**
Count(5)	10	Sat	0.14s	1.43s
Count(5)	9	Unsat	1.5s	0.24s
Count(8)	16	Sat	2.2s	86.1s
Count(8)	15	Unsat	152s	15.26s

into a logic formula checked by an SMT solver with rich background theories including set comprehensions. The running times of the prototype tool in [110] and our tool PIPE+ Verifier on property $Count(n)$ are shown in Table 5.6.

5.4 A Refinement of Bounded Model Checking

5.4.1 Motivation

Recall that bounded model check a Petri Net model, a formula ϕ_k is generated from the net model. ϕ_k can be represented in equation 5.9:

$$\phi_k \doteq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} f(s_i) \tag{5.9}$$

A state s_i is represented by all the places' marking in the net model. $I(s_0)$ denotes the initial marking and $f(s_i)$ represents negated properties. Equation 5.10 represents a transition between symbolic states s_i and s_{i+1}, each transition T is a disjunction of all the transitions t in the Petri net model as each of them may be enabled.

$$T(s_i, s_{i+1}) \;=\; \bigvee_{j=0}^{n} t_j(s_i, s_{i+1}) \tag{5.10}$$

This naive method in equation 5.9 is not efficient as it exploring exhaustively all the interleavings of a net's transitions in depth k but without considering the dependencies among them. The computation complexity of the naive method thus becomes very high and sometimes difficult to compute a result if k is getting larger.

Based on some observations and properties of PrT Nets, the firing of a transition depends on the existence of tokens from its input places P, thus depends on the other transitions that producing tokens for P. If in a state s, a transition t's input places are empty or not enough token to enable t, the checkings on t at state s obviously cannot able to fire. If in a state s, a transition t's output places are not visible to properties, firing t can only update the markings of its output places, thus if a transition firing sequence σ_k only has a t but does not have another transition t' consume tokens from t's output places and firing after t, then firing t is redundant in σ_k. With these observation, it is possible to avoid those redundant checkings by SMT solvers and improve the efficiency of BMC.

For example, in Figure 5.13, the initial marking is $P_0\{tok_0\}$, $P_1\{\}$, $P_2\{\}$, if we want to check whether it can reach a marking where $P_2\{tok_0\}$. The model formula

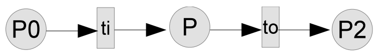


Figure 5.13: A simple model

produced by equation 5.9 within $k = 2$ is:

$$\phi_k = I(s_0) \wedge (t_i\,(s_0, s_1) \vee t_o\,(s_0, s_1)) \wedge (t_i\,(s_1, s_2) \vee t_o\,(s_1, s_2)) \wedge (f(s_0) \vee f(s_1) \vee f(s_2)) \tag{5.11}$$

By naive checking , the generated formula ϕ_k covers all possible firing orders including $t_i \rightarrow t_i$, $t_i \rightarrow t_o$, $t_o \rightarrow t_i$ and $t_o \rightarrow t_o$, which is infeasible in the original PrT Nets model. Firing t_i twice cannot reach a marking in P_2 because P_2 is not directly updated by t_i. Firing t_o before t_i is impossible because P_1 is empty initially that cannot enable t_o if t_i have not yet fired. The only firing sequence feasible for this model is $t_i \rightarrow t_o$. If we have this information before checking, we can check only feasible firing sequences directly by avoiding infeasible ones.

In this example, we get a reduced formula ϕ':

$$\phi^{'} = I(s_0) \wedge (t_i\,(s_0, s_{temp}) \wedge t_o\,(s_{temp}, s_2)) \wedge (f(s_0) \vee f(s_1))$$

where s_{temp} is an intermediate state for a consecutive firing of t_i and t_o, and does not need to be checked.

In this section, we are presenting a method to generate the reduced $\phi^{'}$ so that avoid exploring redundant transition firing sequences. Besides, we prove the new formula ϕ' preserves the reachability property in the original formula ϕ.

5.4.2 Generate a Reduced Formula

While translating a HLPN model to ϕ_k through naive method, ϕ_k represents all the permutations of transition firing sequences in the HLPN model within length k. However, some of subformulas represent infeasible sequences in the HLPN model and some of them is equivalent to another subformula, but they are still checked. In this section, we are providing a new translation method to generate a reduced formula ϕ' thus avoid checking these unnecessary subformulas.

5.4.2.1 Preliminary Definitions

Suppose a HLPN model N has a set of of n transitions T, a transition in N denotes $t_j \in T$, where $0 \leq j < n$.

Definition 1. In HLPN, a transition firing sequence σ of length k denoted as $\sigma_k = \underbrace{t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_{k-1}}_{k}$, where t_j can be any transition in T. A σ_k is feasible if there is a marking M_0 can enable all the transitions in σ_k with σ_k 's order, thus can produce a state transition sequence $M_0 t_0 M_1 t_1 ... t_{k-1} M_k$. If none of M_0 can produce such sequence, σ_k is infeasible.

In equation 5.9, formula $\phi_{ph} = \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ represents all the possible σ_k in a HLPN model.

Definition 2. A subformula ω_k is a formula represents a fire sequence σ_k in N, $\omega_k = \bigwedge_{i=0}^{k-1} t_j\,(s_i, s_{i+1})$, where $0 \leq j < n$.

Combining equation 5.10, expanding ϕ_{ph} results in a disjunction of subformulas:

$$\phi_{ph} = \omega_0 \vee \omega_1 \vee \cdots$$

Lemma 3. *If σ_k is infeasible in HLPN model, ω_k evaluates to be false.*

Proof. Since ω_k is a formula representation of a firing sequence σ_k in the HLPN model, if σ_k is infeasible, one of transition in σ_k cannot be enabled. If the formula represents t_j in ω_k evaluate to be false, ω_k evaluates to be false. $\square$

Theorem 4. *Suppose ω_k represents σ_k. If σ_k is infeasible in HLPN model, remove ω_k from ϕ_{ph}* does not affect the truth of ϕ_{ph}.

Proof. Denotes ϕ_{ph} is a reduced formula by removing ω_k from ϕ_{ph}. According to lemma 3, ω_k evaluates to be false if σ_k is infeasible. Thus, $\phi_{ph} = \phi'_{ph} \vee \omega_k = \phi'_{ph} \vee false = \phi'_{ph}$. □

Definition 5. According to the definition in [30], an independence relation $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:

Enabledness: If $\alpha, \beta \in enabled\,(s)$ then $\alpha \in enabled\,(\beta\,(s))$.

Commutativity: $\alpha, \beta \in enabled\,(s)$ then $\alpha\,(\beta\,(s)) = \beta\,(\alpha\,(s))$

The dependency relation $D = (T \times T)\setminus I$.

Lemma 6. *If transition t and t' has independence relation, switching the firing order of them results in an equivalent ω'_k of ω_k.*

Proof. According to $Def.$ 5's commutativity property, firing t and t' in either order results in the same state. Thus switching their firing order in ω_k results in equivalent ω'_k. □

Lemma 7. *If two subformula ω_k and ω'_k is equivalent, remove ω'_k from ϕ_{ph} does not affect the truth of ϕ_{ph}. $\phi_{ph} = \phi'_{ph} \vee \omega'_k = \phi'_{ph}$.*

Proof. If ω_k and ω'_k is equivalent, we can substitute ω'_k with ω_k. $\phi_{ph} = \phi'_{ph} \vee \omega_k \vee \omega'_k = \phi'_{ph} \vee \omega_k \vee \omega_k = \phi'_{ph}$. □

Theorem 8. *If ω'_k can be obtained by switching independent transitions t and t' from ω_k, removing ω'_k from ϕ_{ph} does not affect the truth of ϕ_{ph}.*

Proof. From Lemma 7 and 6, this theorem is proved. □

Definition 9. According to [30], a transition $t\left(s_i, s_{i+1}\right)$ is invisible with respect to property f if $f\left(s_i\right) = t\left(s_i, s_{i+1}\right) \wedge f\left(s_{i+1}\right)$.

Therefore, from theorem 4 and 8, if we can generate a reduced formula $\phi^{'}$ that avoid checking infeasible path and redundant equivalent path, we can improve our checking efficiency and still get correct result.

To lock some transitions firing order, we need to analyze the dependencies in a PrT Nets model.

Since a transition t's input arc label l_i indicates the number of tokens from t's input place to be consumed and t's output arc label l_j indicates the number of tokens to be produced and added to the connected output places.

5.4.2.2 Preprocessing By Locking Transitions

Because ϕ_k includes subformula ω_k that represents infeasible path in HLPN model as well as redundant equivalent subformula, we can preprocess the formula to remove these subformulas from ϕ_k. Denotes T_{all} as all the transitions in a HLPN model N.

Definition 10. A pattern Pat with a set of transitions $T_p \subseteq T_{all}$ is a subformula that represents a segment of transition firing sequence of length n from state i to state j in N. $Pat = t_0\left(s_i, s_{i+1}\right) \wedge t_1\left(s_{i+1}, s_{i+2}\right) \wedge \ldots \wedge t_n\left(s_{j-1}, s_j\right)$, where $t_1, ..., t_n \in T_p$.

Definition 11. Locking a set of transitions $T_p \subseteq T_{all}$ with a pattern Pat means each $T\left(s_i, s_{i+1}\right)$ in the original model formula ϕ_k is substituted with $T^{'}\left(s_i, s_{i+1}\right) = Pat \vee T_{rest}$, where $Pat\left(s_0, s_1\right) = t_0\left(s_i, s_{t0}\right) \wedge t_1\left(s_{t0}, s_{t1}\right) \wedge \ldots \wedge t_n\left(s_{tn}, s_{i+1}\right)$ and s_t represents a temporary state, $T_{rest} = T_{all} \backslash T_p$.

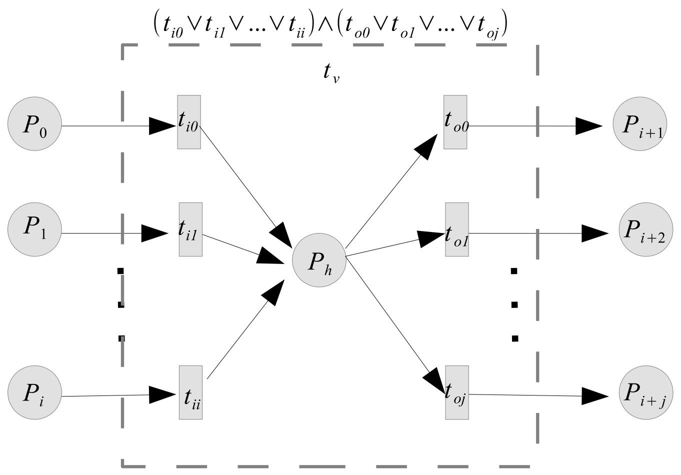


Figure 5.14: A Preprocessing Pattern

After locking T_p with pattern Pat on ϕ_k, the resulted reduced formula ϕ'_j:

$$\phi_j^{'} = I(s_0) \wedge \bigwedge_{i=0}^{j-1} T^{'}(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{j} f(s_i) \tag{5.12}$$

where $T^{'}(s_i, s_{i+1}) = Pat \vee T_{rest}$ and $j \leq k$.

Compare to the original formula ϕ_k, we need to build the Pat carefully for ϕ'_j has to be equivalent with ϕ_k under reachability properties:

1. only the false ω_k and redundant ω_k in ϕ_k are removed;
2. a transition t generating temporary state s_t is invisible to reachability properties.

5.4.2.3 A Pattern

Definition 12. An initial marking place p_{ini} is that p_{ini} has at least one token in state s_0; a property identified place p_r is part of property f that p_r needs to be checked for the satisfiability of f.

In a HLPN model, a transition is connected with input arcs and output arcs, each arc has a label indicating the type of tokens to instantiate the transition.

Definition 13. A simple variable input/outputs label denotes the transition consumes/produces one token to the arc connected place. A set variable input/output label denotes the transition consumes/produces a set token to the arc connected place.

Definition 14. In PrT Nets, a transition t is enabled when:

1) If t's input arc label is simple variable, the label connected place has at least one token;

2) If t's input arc label is a set variable, the label connected place has at least one set token;

3) The token can instantiate t's transition formula to be true.

Figure 5.14 shows a place P_p that is connected by a set of transitions $T_p = \{t_{i0}, t_{i1}, \dots, t_{iu}, t_{o0}, t_{o1}, \dots, t_{ov}\}$. P_p's input transition set is $T_{pi} = \{t_{i0}, t_{i1}, \dots, t_{iu}\}$and output transition set is $T_{po} = \{t_{o0}, t_{o1}, \dots, t_{ov}\}$.

Under the following conditions:

1. All the arc label connected to P_p are simple variables;
2. P_p is neither an initial marking place nor a property identified place;
3. P_p is the only output place of all T_{pi} and the only input place of all T_{po}.

Let $s^{'}$ be a next state of s and $s^{''}$ be a next state of $s^{'}$. We can apply a locking pattern:

$$Pat = \left(t_{i0}\left(s, s^{'}\right) \vee t_{i1}\left(s, s^{'}\right) \vee ... \vee t_{iu}\left(s, s^{'}\right)\right) \wedge \\ \left(t_{o0}\left(s^{'}, s^{''}\right) \vee t_{o1}\left(s^{'}, s^{''}\right) \vee ... \vee t_{ov}\left(s^{'}, s^{''}\right)\right) \quad (5.13)$$

5.4.3 The Correctness of the Pattern

Lemma 15. *In HLPN, if a transition t's output places does not include a property identified place, t is invisible to reachability property f_r.*

Proof. Reachability property f_r in a HLPN refers to the property identified place P_r's marking. Since only add a token to P_r can update P_r's marking with new value, if fire t does not update P_r, t is invisible to f_r. □

Lemma 16. *In a shortest subformula ω_k that evaluates ϕ_k to be true, if ω_k has a $t_i \in T_{pi}$, it has a $t_o \in T_{po}$ after t_i, represents as $t_i(s_j, s_{j+1}) \wedge t_o(s_k, s_{k+1})$ and $j < k$.*

Proof. Because t_i's only output place is P_p, but P_p is assumed as not an property identified place, according to lemma 15, t_i is invisible to reachability property f_r. Since P_p's output transition $t_o \in T_{po}$, if a subformula ω_k does not contain a t_o, according to def. 9, $f(s) = t_i(s, s') \wedge f(s')$, so, t_i becomes redundant thus ω_k is not the shortest ω_k. □

Lemma 17. *In a shortest subformula ω_k that evaluates ϕ_k to be true, if ω_k has $t_o \in T_{po}$, it must have t_i before t_o.*

Proof. Since there is no other than t_i can enable t_o and t_o's only input place P_p is not an initial marking place still cannot enable t_o. If ω_k does not have a t_i before

t_o, t_o cannot be enabled, thus a σ_k with only t_o is infeasible in N. Therefore, ω_k cannot evaluate ϕ_k to be true. $\square$

Theorem 18. *In a shortest subformula ω_k that evaluates ϕ_k to be true, if it has a transition $t \in T_p$, it must have both and with an order t_i before t_o. Otherwise, ω_k is false thus can be removed from ϕ_k.*

Proof. From lemma 16 and 17, if ω_k has either t_i or t_o, it must have both and in an order that t_i before t_o. $\square$

Lemma 19. *A transition t does not depend on another transition t' firing after t.*

Lemma 20. *A transition t_m fires between $t_i \in T_{pi}$ and $t_o \in T_{po}$ in a Pat is either independent of t_i or independent of t_o.*

Proof. Because t_m fires between t_i and t_o:

If t_m is either $t'_i \in T_{pi}$ or $t'_o \in T_{po}$, we can let t'_i or t'_o be t_i or t_o.

If t_m includes a Pat, it only updates t_i's input places and t_o's output places, thus it does not update P_p's marking, as t_o depends on P_p, t_o does not depend on t_m. Besides, according to lemma 19, t_m does not depend on t_o as it happens before t_o. Thus, t_m is independent of t_o.

If $t_m \in T_{rest}$, since t_m is firing after t_i, according to lemma 19, t_i does not depend on t_m. Similarly, t_o firing after t_m, t_m does not depend on t_o. On the other hand, since t_o only depend on t_i but t_i does not depend on t_m, t_o does not depend on t_m. Thus, in this case, t_m is independent of t_o. $\square$

Theorem 21. *All the subformula ω_k that have a set of transitions in T_p and in $Pat's$ order are equivalent to ω'_k where $t \in T_p$ are in a consecutive manner like Pat.*

Proof. If the T_p is in ω_k but not represent consecutive transition firings in N like the Pat, at least one transition t_m is firing between t_i and t_o. According to lemma 6, if t_m is independent of either t_i or t_o, t_m's firing order in N can be switched with t_i or t_o and according to lemma 20, the switched $\omega_k^{''}$ is equivalent with ω_k, we got $\omega_k = \omega_k^{''}$. Obviously since $\omega_k^{''} = \omega_k^{'}$, then $\omega_k = \omega_k^{'}$. $\square$

Therefore, according to the theorem 18 and 21, the correctness of applying the patten in ϕ_k and get a reduced formula ϕ_j' is proved.

5.4.4 Error Path

After applying pattern to HLPN model and produced a reduced formula ϕ_k', the error path can still be regenerated along with the pattern. Compared to the naive method, there are two cases to consider:

1. If the error path σ_k does not contain a transition in the pattern, the generation of error path is the same as the naive method;

2. If the error path σ_k contain a pattern, since the new path generated by SMT solver is also a reduced state sequence because the S_{temp} is ignored in the new path, to build a complete path, each state transition that are not reflected by firing one transition need to refer to the pattern to see if it is result in firing a sequence of transitions. If yes, the intermediate transitions in the pattern are added to the path; otherwise, search and refer for the next pattern.

Table 5.7: Verifying Shared Memory Model with Refined Method

Processors	**Step Bound**	**Time (seconds)** Naive Method	**Time (seconds)** Refined Method	**Heap Size (Mb)** Naive Method	**Heap Size (Mb)** Refined Method
5	5	0.07	0.05	0.86	0.78
5	10	0.30	0.23	1.54	1.34
5	15	1.49	1.20	2.53	2.42
10	5	0.12	0.10	1.02	1.00
10	10	0.98	0.84	2.08	1.97
10	15	15.50	8.37	4.73	4.60

5.4.5 Experiment on Reduced Model: Shared Memory Model

Figure 4.11 shows a share memory model in PrT Nets. In this model, the pattern can be applied to place $P_{OwnMemAcc}$'s input transition $T_{Begin\_Own\_Acc}$ and output transition $T_{End\_Own\_Acc}$, thus the pattern is defined as $T_{Begin\_Own\_Acc} \wedge T_{End\_Own\_Acc}$. Table 5.7 presents a comparison of time and memory consumption of the naive method and the refined method. Since we only applied one pattern in this model, it still reflects some improvements. The formula we checked is shown in Equation 5.14:

$$\Box\neg(marking\,(Ext\_Mem\_Acc) = \langle 3,0\rangle \wedge marking\,(Ext\_Mem\_Acc) = \langle 2,4\rangle) \tag{5.14}$$

Table 5.8: Verifying Seabed Rig Robotic Workflow Model with Refined Method

Step Bound	**Time (seconds)** Naive Method	**Time (seconds)** Refined Method	**Heap Size (Mb)** Naive Method	**Heap Size (Mb)** Refined Method
5	0.20	0.14	1.09	0.94
10	0.40	0.25	1.80	1.48
15	0.60	0.38	2.60	2.19
30	1.41	1.04	4.98	4.17
50	3.55	2.03	8.42	6.44

5.4.6 Experiment on Reduced Model: Seabed Rig Robotic Workflow

Figure 4.15 shows a Seabed Rig robotic workflow model in PrT Nets. In this model, our pattern can be applied to place $P_{RO\_atGripperPos}$, $P_{RO\_atGripperAttach}$ and $P_{RO\_hasGripper}$. Thus, the pattern is contructed from the related transitions and defined as in Equation 5.15:

$$T_p = T_{RO\_MoveToGripper} \wedge T_{RO\_MoveToGripperAttach} \wedge T_{RO\_RetriveGripper} \wedge T_{RO\_MoveToHomePos} \qquad (5.15)$$

Table 5.8 presents a comparison of time and memory consumption of the naive method and the refined method. The formula we checked is shown in Equation 5.16:

$$\Box\neg(marking\,(RG\_CenterDown) = \langle 1, 1\rangle \qquad (5.16)$$

5.5 Analyzing Hierarchical Models

As we have defined a hierarchical model SAM in Section 4.2, a supporting analysis method should be defined. Compositional analysis is a possible solution but is only effective to loosely coupled modules in a system [31]. An efficient way is to find an equivalent PrT Nets model thus can leverage the three analysis methods mentioned above.

5.5.1 Generating An Integrated Flat Petri Net Model

Because a SAM model is hierarchically and modulely specified and each component in a different layer has its own behavior model, it is hard to leverage the three existing analysis methods. However, by following the definition of SAM framework, it is possible preprocesses the model by flattening the hierarchical structure and integrating modulely distributed net models.

In this phase, all the individual behavior models for different components of a SAM model need to be connected by directed arcs both horizontally and vertically . Therefore, selecting interfaces among all the behavior models are important. Because each behavior model has input places (places without any input arc, e.g. Sender in Figure 4.16) and output places (places without any output arc, e.g. Receiver in Figure 4.16), these input and output places are chosen as candidate interface places heuristically. Similarly, each SAM component has its input ports (P1 in Figure 4.16) and output ports (P4 in Figure 4.16) for the communication with other components, these input and output ports form the interface of the component.

The connection strategies are :

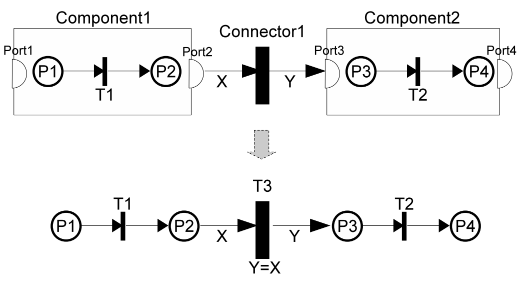


Figure 5.15: Generating Analysis Model by Horizontal Connection

- **Horizontally**: each SAM component has its input ports and output ports specified by one of the interface places of the underlying Petri net model (e.g. in Figure 5.15, Port 1 specified by P1 and Port 2 specified by P2). Integrating Petri net models from different components in the same hierarchical layer is by connecting the interface places. Moreover, the components in the same layer are connected by SAM connectors and arcs, so that SAMTools transforms them into Petri net transitions and arcs respectively. A new transition is created for each connector during the transformation (e.g. in Figure 5.15 is T3). The pre-condition of such transition is true by default; however a post-condition may be added. In the example, a post-condition "Y=X" is added. The variables in the new transition formula match the connector's input and output arc variables. The sort of the variables is exactly the sort of the interface places, specified in ports, through connected arcs. Corresponding new arcs are added to preserve the flow relationships, which are connected with the interface places in ports and related transitions. For example, a new arc between place P2 in Port2 and T3 in Connector1.

- **Vertically**: The input or output ports not connected with any arcs in a component are mapped to the corresponding input and output ports in the parent component. For example in Figure 4.16, ports P1 and P2 in top layer component A1 are mapped to the second layer's input port P5 and output port P8.

Thus, the behavior models are connected and flattened into an integrated flat HLPN model that is ready to be checked via different analysis methods.

CHAPTER 6 MODELING AND ANALYSIS IN SAMTOOLS

6.1 SAMTools

Since 1990s, software architecture has become an active research area within software engineering for studying the structure and behavior of large software systems [100]. A rigorous approach towards architecture system design can help to detect and eliminate design errors early in the development cycle, to avoid costly fixes at the implementation stage and thus to reduce overall development cost and increase the quality of the systems. SAM [113, 54, 55, 56] is a general framework for systematically modeling and analyzing software architecture specifications, whose foundation is a dual formalism combining a Petri net model for behavioral modeling and a temporal logic [55] for property specification. In this Chapter, we present a tool set, called SAMTools, for modeling and analyzing SAM specifications. SAMTools supports:

1. a software architecture description in a hierarchical manner through decomposition and refinement;
2. dynamic behavioral simulation;
3. property analysis through explicit state model checking [30] using SPIN tools [60];
4. property analysis through bounded model checking (BMC) [26] using SMT solver Z3 [36].

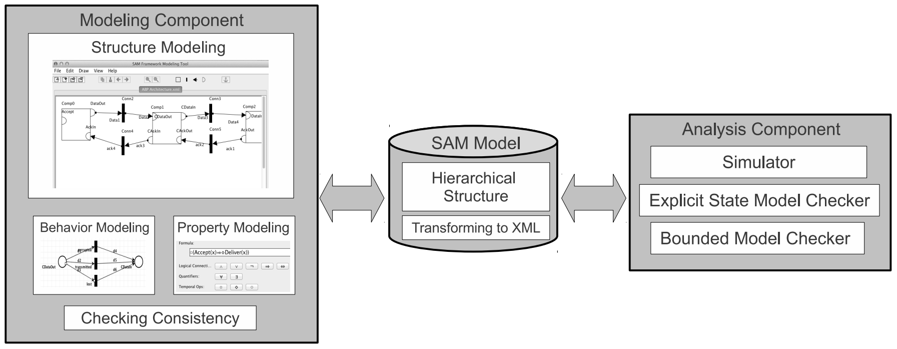


Figure 6.1: The Functional View of SAMTools

6.1.1 An Overview of SAMTools

SAMTools is comprised of a modeling component, a SAM model, and an analysis component (Figure 6.1). The modeling component has three functions: structure modeling creates hierarchical compositions, behavior modeling specifies behaviors of software components/connectors using Petri nets, and property modeling defines property specifications using temporal logic. The SAM specification is a hierarchical structure integrating the results of structure, behavior, and property modeling, which can be transformed into XML format. The analysis component includes three complementary analysis methods: simulation, explicit state model checking and bounded model checking (BMC). The simulator in SAMTools executes the behavior model in PrT Nets. For explicit state model checking in SAMTools, a SAM model is translated into an equivalent model in PROMELA [10] suitable for checking by a widely used model checker called SPIN [60]. For BMC in SAMTools, a first-order logic formula representing the model is generated and checked by an SMT solver called Z3 [36].

6.2 An Example of Using SAMTools: Mondex

We use a smart card system, Mondex, as an example to demonstrate our modeling and analysis methodology in SAMTools.

6.2.1 Mondex

Mondex [119] smart card system is an electronic purse payment system, which involves a number of electronic purses with values and can exchange the values through a communication device. Mondex was the first pilot project of the International Grand Challenge on Verified Software [115], and was awarded the highest assurance level of secure systems, ITSEC Level E6 [116]. Mondex was first formally specified and proved using Z language [104]. Since 2006, several research groups around the world have applied different formalism to specify and analyze the Mondex. For example, Massachusetts Institute of Technology uses Alloy [63], University of Southampton uses Event-B [7] and University of Bremen uses OCL (object constraint language) [99].

The refinement relation of two models in the Mondex are:

1. Abstract model is a very simple model with an atomic operation, payment, which is transfer balance from one purse to another.
2. Concrete model is the actual implementation of the transaction protocol which involves a sequence of non-atomic operations. Security issues needs to be concern at this level. For example, a purse could disconnect at any time due to power failure, a messages could be lost by the communication channel and messages in channel are public readable by other purses.

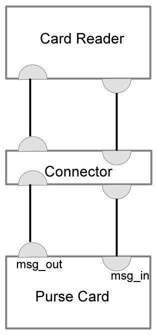


Figure 6.2: Mondex Composition Model in SAM

6.2.2 Modeling Mondex in SAMTools

Our previous work [119, 118] formalized Mondex abstract and concrete purse models using SAM. We give a brief description in this section.

6.2.2.1 Structure Modeling

A Mondex smart card system is composed of three components: Card Reader, Connector and Purse Card. This architecture is modeled in SAM in Figure 6.2. Purse communicates with card reader via connector component. To the Purse Card component, it takes input massages and produce output messages via msg_in and msg_out connectors.

6.2.2.2 Behavior Modeling: Concrete Purse

The Net Graph In a refined level, the purse's behavior is specified in high level Petri nets (Figure 6.3). The net model contains 3 places, ConWorld, msg_in and msg_out (3 msg_out circles in 6.3 represents the same place). ConWorld place is where purses are located. Each purse is a data structure of 15 data fields, such as purse name, balance, sequence number, log, pay details and so on. Each msg represents a data structure of 10 data fields, such as operation type, name, value, sequence

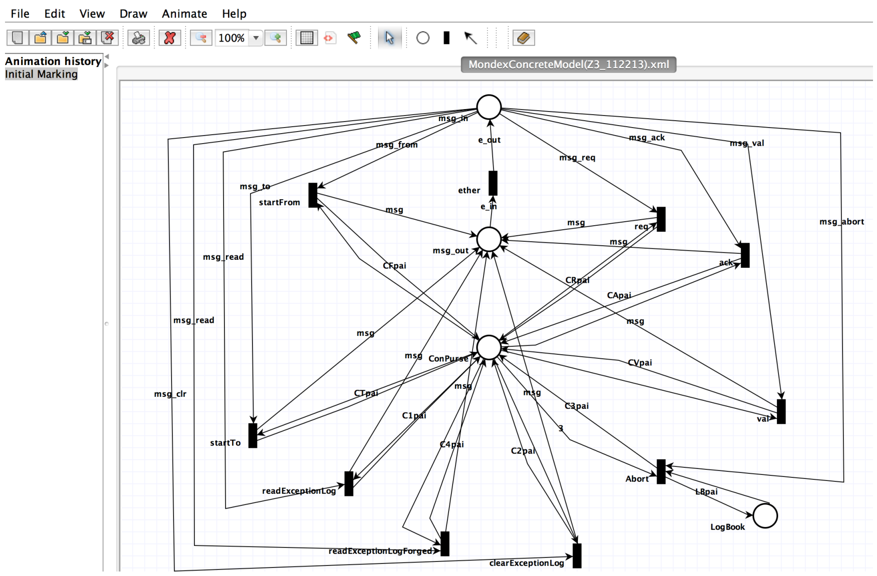


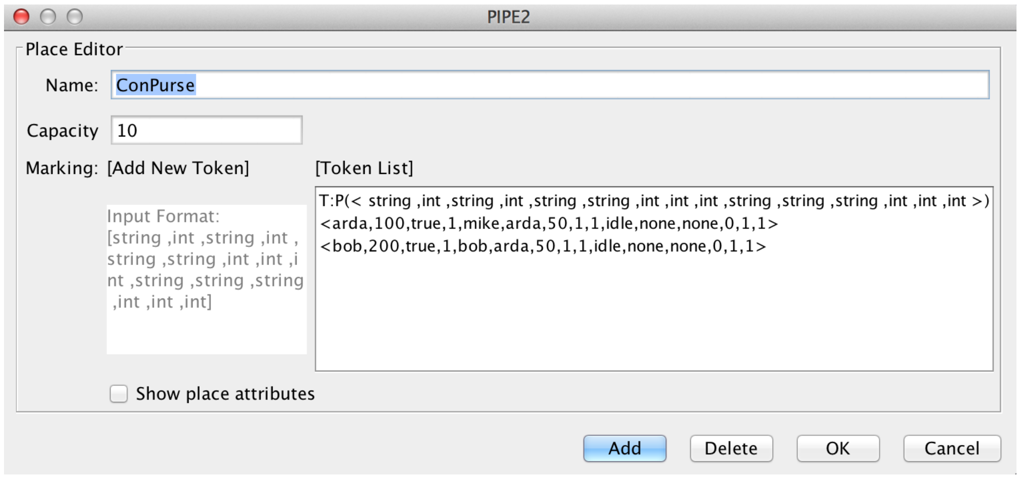
Figure 6.3: Concrete Purse Model in PrT Nets

number and so on. Besides, there are 7 transitions in the model indicates the steps of a payment process, including startFrom,startTo,readExceptionLog,req,val,ack, exceptionLogResult, exceptionLogClear and forged.

Place Type The semantics of place types have been defined in [120], we only illustrate how they are modeled in SAMTools. As the modeling in SAMTools support primitive data types: string and integer, we specify all the place types using this two primitive data types. Figure 6.4 illustrates the place type of $P_{ConPurse}$. Figure 6.5 shows the place type of P_{msg_in} and P_{msg_out}.

Transition Formulas There are 7 transitions in Figure 6.3, but we only show the transition T_{start_From} in Figure 6.6.

Arc Labels The arc labels are specified on the net graph in Figure 6.3. For example, the arc label between $T_{startFrom}$ and P_{msg_in} is msg_from, indicating a



Name: ConPurse

Capacity 10

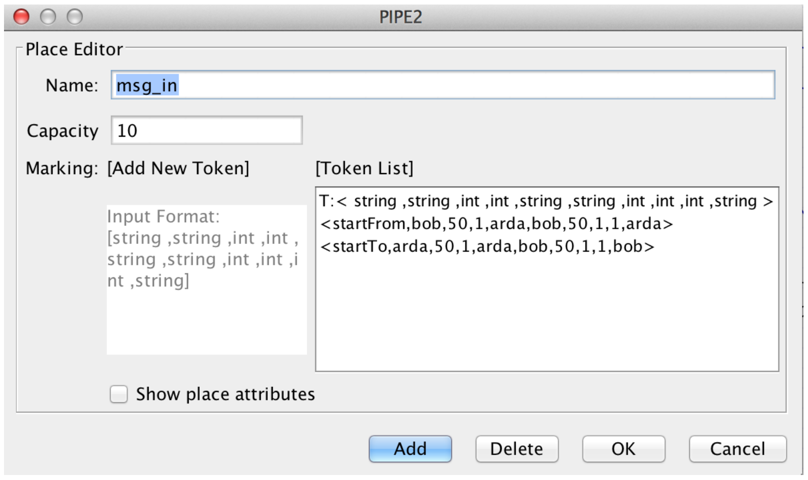
Marking: [Add New Token] [Token List]

Input Format:
[string ,int ,string ,int ,string ,string ,int ,int ,int ,string ,string ,string ,int ,int ,int]

```
T:P(< string ,int ,string ,int ,string ,string ,int ,int ,int ,string ,string ,string ,int ,int ,int >)
<arda,100,true,1,mike,arda,50,1,1,idle,none,none,0,1,1>
<bob,200,true,1,bob,arda,50,1,1,idle,none,none,0,1,1>
```

Show place attributes

Figure 6.4: Place Type and Initial marking of ConPurse



Name: msg_in

Capacity 10

Marking: [Add New Token] [Token List]

Input Format:
[string ,string ,int ,int ,string ,string ,int ,int ,int ,string]

```
T:< string ,string ,int ,int ,string ,string ,int ,int ,int ,string >
<startFrom,bob,50,1,arda,bob,50,1,1,arda>
<startTo,arda,50,1,arda,bob,50,1,1,bob>
```

Show place attributes

Figure 6.5: Place Type and Initial Marking of Msg_in and Msg_out

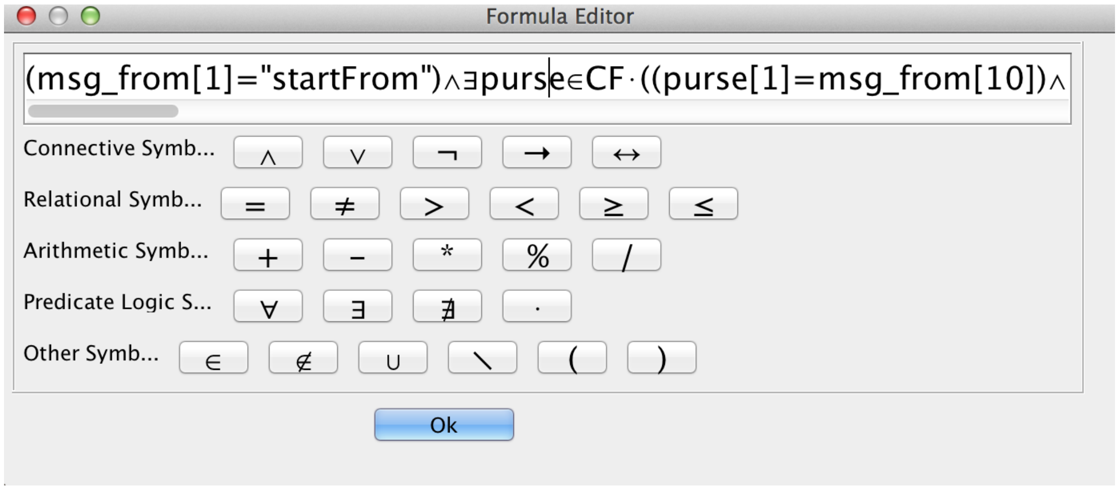


Figure 6.6: Transition startFrom's Formula

simple variable. The arc label between $T_{startFrom}$ and $P_{ConPurse}$ is CF', indicating a set variable.

Tokens and Abstract Tokens As specified in the Figure 6.4, $P_{ConPurse}$ is a power set place thus tokens is stored as a whole set and the token type is abstract token. P_{msg_in} and P_{msg_out} are regular places that contain regular tokens shown in Figure 6.5.

Place Bound The place bound is specified for the following analysis process that requires to define the maximum token number in a place. We define them as 10.

6.2.2.3 Property Specification

As described in [118], two properties of Mondex are to be verified:

1. All Value Accounted: all value must be accounted, which is the sum of all purses' balances and lost components does not change;
2. No Value Created: no value may be created in the system, which states that the sum of all the purses' balances does not increase.

Suppose the system initiates with two purses, then the properties are specified in FOLTL as:

1. $\Box\ (purse1.balance + purse2.balance + lost\_sum) = balance\_sum$
2. $\Box\ (purse1.balance + purse2.balance) \leq balance\_sum$

6.2.3 Analyzing Mondex in SAMTools

As mentioned above, we illustrate SAMTools analysis functionalities by running three analysis methods on Mondex HLPN model, simulation, explicit state model checking and bounded model checking. All experiments were conducted on a 32-bit Intel Core Duo CPU @3.0GHz box, with 4GB of RAM, running 32-bit Ubuntu.

6.2.3.1 Simulation

In SAMTools, simulation is the execution of PrT Nets. A global simulation runs on the flattened PrT Nets model explained in Section 5.2. SAMTools imports PIPE+'s simulator that can enable and fire transition, which results in a transition firing sequence and markings. Users can either click "fire high level Petri nets" button to generate the firing sequence step by step or automatically generate a firing sequence up to a specified bound of firing actions. A simulation run is shown in Figure 6.7, the firing sequence is stored in the animation history in the editor. In this example is startTo - ether - startFrom - req - ether - val - ether - ack. In addition, SAMTools also provides a simulation summary report (Figure 6.8) including the snapshots of each state showing the distribution of high level tokens, transition firing sequences and total time consumed during the simulation run.

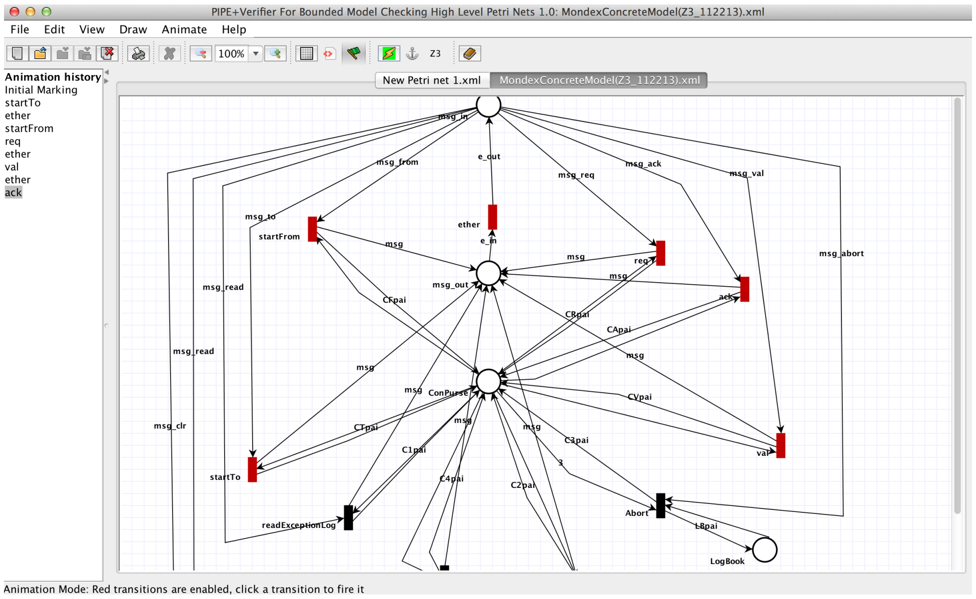
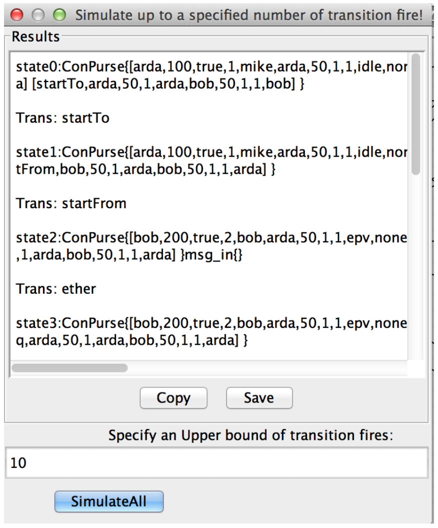
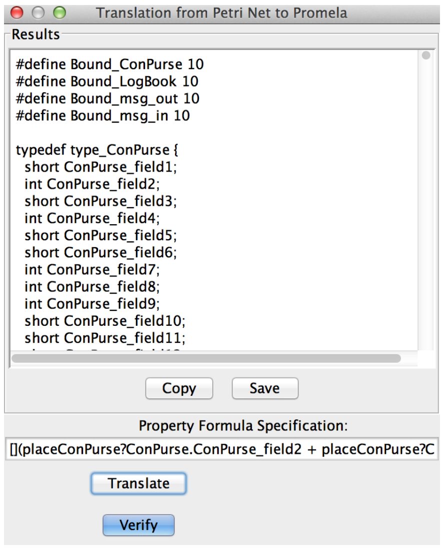


Figure 6.7: Simulate Mondex Model


Simulate up to a specified number of transition fire!
Results
Copy
Save
Specify an Upper bound of transition fires:
10
SimulateAll

```
state0:ConPurse{[arda,100,true,1,mike,arda,50,1,1,idle,nor
a] [startTo,arda,50,1,arda,bob,50,1,1,bob] }

Trans: startTo

state1:ConPurse{[arda,100,true,1,mike,arda,50,1,1,idle,nor
tFrom,bob,50,1,arda,bob,50,1,1,arda] }

Trans: startFrom

state2:ConPurse{[bob,200,true,2,bob,arda,50,1,1,epv,none
,1,arda,bob,50,1,1,arda] }msg_in{}

Trans: ether

state3:ConPurse{[bob,200,true,2,bob,arda,50,1,1,epv,none
q,arda,50,1,arda,bob,50,1,1,arda] }
```

Figure 6.8: A Simulation Run of Mondex in SAMTools


Translation from Petri Net to Promela
Results
Copy
Save
Property Formula Specification:
Translate
Verify

```
#define Bound_ConPurse 10
#define Bound_LogBook 10
#define Bound_msg_out 10
#define Bound_msg_in 10

typedef type_ConPurse {
  short ConPurse_field1;
  int ConPurse_field2;
  short ConPurse_field3;
  int ConPurse_field4;
  short ConPurse_field5;
  short ConPurse_field6;
  int ConPurse_field7;
  int ConPurse_field8;
  int ConPurse_field9;
  short ConPurse_field10;
  short ConPurse_field11;
```

```
[](placeConPurse?ConPurse.ConPurse_field2 + placeConPurse?C
```

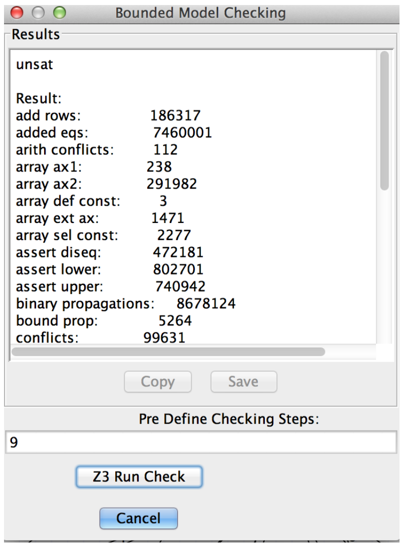
Figure 6.9: Explicit State Model Checking Mondex in SAMTools

6.2.3.2 Explicit State Model Checking

Explicit state model checking in SAMTools is a push button process. After the Mondex model and its properties are completely specified in PrT Nets, it is then automatically translated into a PROMELA model, as shown in Figure 6.9. The PROMELA model is sent to SPIN model checker, and SAMTools receives the checking results of the model checking run in SPIN. If the property is not satisfied, an error trace will be produced and trimmed to allow a user to find the cause of the design error from the beginning.

6.2.3.3 Bounded Model Checking

Bounded model checking in SAMTools requires a user to define the length of the checking paths k. The generated formula is sent to a SMT solver Z3 to check its satisfiability. If the result is satisfiable, a path leads to an error state is produced and trimmed to allow a user to find the cause of the design error from the beginning. Otherwise, the system is safe within the predefined length of paths k. Since the



```
unsat

Result:
add rows:            186317
added eqs:            7460001
arith conflicts:      112
array ax1:           238
array ax2:           291982
array def const:      3
array ext ax:         1471
array sel const:      2277
assert diseq:         472181
assert lower:         802701
assert upper:         740942
binary propagations:    8678124
bound prop:           5264
conflicts:           99631
```

Figure 6.10: Bounded Model Checking Mondex in SAMTools

checking result is incomplete, a user can increase the k value and repeat the checking process to increase confidence.

In the Mondex concrete purse model, up to nine transitions may be involved in a transaction process, we set $k = 9$. The SAMTools encodes the generated formula into a 5000 lines C file by using Z3 provided C language API. Then the c file is sent to the Z3 solver. The checking report shown in the first line in Figure6.10 is unsat, which means this transaction process is preserved since $\neg f$ is not reachable within $k = 9$ transition firing steps. The time and memory consumed for this checking process are 27.85s and 11.42 Mbytes respectively.

CHAPTER 7 CONCLUSION AND FURTHER RESEARCH

7.1 Conclusion

This dissertation describes a framework that can formal modeling and analysis of concurrent and distributed systems based on PrT Nets. The framework has the following functionalities:

1. Modeling:

   (a) Formal modeling a system's behavior based on PrT Nets;

   (b) Formal modeling a system's property using FOLTL;

   (c) Formal modeling a system modulely via components and connectors.

   (d) Formal modeling a system hierarchically via hierarchical layers of components and connectors.

2. Analysis:

   (a) Simulating the PrT Nets model;

   (b) Model checking the PrT Nets model with properties described in FOLTL;

   (c) Bounded model checking the PrT Nets model with properties described in FOLTL;

   (d) Bounded model checking the PrT Nets model with an improved method;

   (e) Analyzing a HLPN model specified modulely and hierarchically.

In addition, we implemented a tool set called SAMTools to automate the formal modeling and analysis process, SAMTools has the following functionalities:

1. Modeling:

    (a) Draw a Petri net graph through a graphical editor via drag and drop actions;

    (b) Specify high level concepts such as place type (token type), transition formula, arc label, markings on the net graph;

    (c) Specify FOLTL properties through the FOLTL input panel;

    (d) Draw components and connectors graphically via drag and drop actions;

    (e) Define a hierarchical model in a top down manner;

    (f) Specify behavior model inside each components.

2. Analysis:

    (a) Flattern hierarchical model;

    (b) Connect modules of components;

    (c) Simulate PrT Nets behavior model step by step or perform a simulation run till no transition can be enabled;

    (d) Model checking a HLPN model with properties in FOLTL with a bottom click. Returns a counterexample if an error is found;

    (e) Bounded model checking a HLPN model with safety (reachability) properties. Returns a state transition path if an error if found.

This framework and supporting tools can be very perspective in modeling critical systems and gain reasonable results by choosing proper analysis method. Modeling in PrT Nets is more closely to real world high level systems than low level Petri nets, thus simplify the modeling process. Analysis by simulation is straightforward, by model checking is complete and by bounded model checking is incomplete but sometimes more efficient in practical problems and can get reasonable result when the problem cannot be tackled by model checking. Besides, we refined the method in automatic converting PrT Nets model into first-order logic formula by removing subformulas describing infeasible interleavings and redundant transition sequences, thus avoid unnecessary checkings and computations by SMT solvers.

Besides, SAMTools is implemented to automate the whole analysis process, thus if a model is completely and properly specified, an analysis result can be gained by a button click action. SAMTools allow us to experiment our ideas and to develop new ideas on HLPNs conveniently.

7.2 Future Work

7.2.1 Extension on Timed Petri Nets

The framework can be extended with integrating timed Petri nets [108], since it has been widely used in modeling and analysis of real time systems. A timed Petri nets (Timed PN) models a discrete event dynamic system and for generating the underlying stochastic processes. Timed PN have well defined semantics which unambiguously defines the behavior of the net and these semantics make it possible to implement simulators for timed PN as well as forming the basis for formal analysis methods.

Besides, van der Aalst introduced time coloured Petri nets (Timed CPN) in [109], that add time to PrT Nets. This enhanced the potential of usage of Timed PN to high level systems by simplying the modeling process. Therefore, adding time can diversify our framework to model a wider range of systems.

7.2.2 Extention with an Adaptor

In SAMTools, the model need to be built with drag and drop action. However, there are a large number of colored Petri net models stored in colored PNML [19], thus it is necessary to load them automatically. In this way, we can extend SAMTools with an adapter that can accommodate various types of Petri nets format, so that SAMTools can test more models easily.

BIBLIOGRAPHY

[1] Cpn tools. http://cpntools.org.

[2] Jcup parser generator. http://www2.cs.tum.edu/projects/cup.

[3] Jflex lexical analyzer generator. http://jflex.de.

[4] Petri net tool database. http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html.

[5] Toyota vehicle recalls. http://www.toyota.com/recall/.

[6] *High-level Petri Nets - Concepts, Definitions and Graphical Notation*, 2000.

[7] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[8] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.

[9] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997.

[10] Gonzalo Argote-Garcia, Peter J. Clarke, Xudong He, Yujian Fu, and Leyuan Shi. A formal approach for translating a sam architecture to promela. In *SEKE*, pages 440–447, 2008.

[11] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, January 2009.

[12] Alessandro Armando, Silvio Ranise, and MichaÃ«l Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140 – 164, 2003. <ce:title>12th International Conference on Rewriting Techniques and Applications (RTA 2001)</ce:title>.

[13] Jean-Loup Baer and C.S. Ellis. Model, design, and evaluation of a compiler for a parallel processing environment. *Software Engineering, IEEE Transactions on*, SE-3(6):394–405, Nov 1977.

[14] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2: static driver verification with under 4 In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 35–42, Austin, TX, 2010. FMCAD Inc.

[15] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Clark Barrett, Leonardo De Moura, and Aaron Stump. Design and results of the 1st satisfiability modulo theories competition (smt-comp. *Journal of Automated Reasoning*, 35:2005, 2005.

[17] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[18] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[19] Jonathan Billington, Søren Christensen, Kees Van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The petri net markup language: Concepts, technology, and tools. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 483–505, Berlin, Heidelberg, 2003. Springer-Verlag.

[20] Pere Bonet, Catalina Llado, Ramon Puijaner, and William Knottenbelt. Pipe v2.5.: a petri net tool for performance modelling. In *23rd Latin American Conference on Informatics*, October 2007.

[21] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189 –209, jul 1993.

[22] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In Javier Esparza and Rupak Majumdar, editors, *Tools*

and Algorithms for the Construction and Analysis of Systems, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer Berlin Heidelberg, 2010.

[23] C.A.Petri. Kommunikation mit automaten. bonn: Institut fÃŒr instrumentelle mathematik, schriften des iim nr. 2. 1962.

[24] Giovanni Chiola and Giuliana Franceschinis. Colored gspn models and automatic symmetry detection. In *PNPM*, pages 50–60, 1989.

[25] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[26] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, page 2001. Kluwer Academic Publishers, 2001.

[27] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.

[28] Edmund Clarke, Daniel Kroening, JoÃ«l Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin Heidelberg, 2004.

[29] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[30] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2000.

[31] E.M. Clarke, D.E. Long, and K. L. McMillan. Compositional model checking. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 353–362, Jun 1989.

[32] M. Courvoisier, R. Valette, J. M. Bigou, and P. Esteban. A programmable logic controller based on a high level specification tool. In *Proc. of the 1983 Conf. on Industrial Electronics*, pages 174–179, New York, 1983. IEEE.

[33] S. Crespi-reghizzi and D. Mandrioli. Petri nets and szilard languages. *Information and Control*, 33(2):177 – 192, 1977.

[34] D. Crockett, A. Desrochers, F. DiCesare, and T. Ward. Implementation of a petri net controller for a machining workstation. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 1861–1867, Mar 1987.

[35] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[36] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[37] Michel Diaz. Modeling and analysis of communication and cooperation protocols using petri net based models. *Computer Networks (1976)*, 6(6):419 – 441, 1982. Protocol Specification, Testing and Verification.

[38] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[39] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[40] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *Software Engineering, IEEE Transactions on*, 28(7):638 – 653, jul 2002.

[41] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[42] Steven Eker, JosÃ© Meseguer, and Ambarish Sridharanarayanan. The maude ltl model checker and its implementation. In Thomas Ball and SriramK. Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 230–234. Springer Berlin Heidelberg, 2003.

[43] Clarence A. Ellis and Gary J. Nutt. Office information systems and computer science. *ACM Comput. Surv.*, 12(1):27–60, March 1980.

[44] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, June 1987.

[45] G. Estrin, Robert S. Fenchel, R.R. Razouk, and M.K. Vernon. Sara (system architects apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *Software Engineering, IEEE Transactions on*, SE-12(2):293–311, Feb 1986.

[46] Lukasz Fronc and Alexandre Duret-Lutz. Ltl model checking with neco. In *ATVA*, pages 451–454, 2013.

[47] HartmannJ. Genrich and Gerda Thieler-Mevissen. The calculus of facts. In Antoni Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976*, volume 45 of *Lecture Notes in Computer Science*, pages 588–595. Springer Berlin Heidelberg, 1976.

[48] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13(1):109 – 135, 1981.

[49] H.J. Genrich and P.S. Thiagarajan. A theory of bipolar synchronization schemes. *Theoretical Computer Science*, 30(3):241 – 318, 1984.

[50] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[51] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. *SIGSOFT Softw. Eng. Notes*, 28(5):257–266, September 2003.

[52] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31 –42, jul 1997.

[53] Xudong He. A formal definition of hierarchical predicate transition nets. In *Application and Theory of Petri Nets*, pages 212–229, 1996.

[54] Xudong He and Yi Deng. Specifying software architectural connectors in sam. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):411–431, 2000.

[55] Xudong He and Yi Deng. A framework for developing and analyzing software architecture specifications in sam. *Comput. J.*, 45(1):111–128, 2002.

[56] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using sam. *Journal of Systems and Software*, 71:1–2, 2004.

[57] Carlos A. Heuser and Gernot Richter. Constructs for modeling information systems with petri nets. In *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, pages 224–243, London, UK, UK, 1992. Springer-Verlag.

[58] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[59] AnatolW. Holt. Coordination technology and petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 278–296. Springer Berlin Heidelberg, 1986.

[60] Gerard Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[61] Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-level petri net model checking with alpina. *Fundam. Inf.*, 113(3-4):229–264, August 2011.

[62] ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.

[63] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[64] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[65] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, May 2007.

[66] Cliff B. Jones. *Systematic software development using VDM (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[67] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 1999.

[68] G. Juanole, B. Algayres, and J. Dufau. On communication protocol modelling and design. In G. Rozenberg, editor, *Advances in Petri Nets 1984*, volume

188 of *Lecture Notes in Computer Science*, pages 267–287. Springer Berlin Heidelberg, 1985.

[69] J. Robert Jump. Asynchronous control arrays. *Computers, IEEE Transactions on*, C-23(10):1020–1029, Oct 1974.

[70] J. Robert Jump and P. S. Thiagarajan. On the interconnection of asynchronous control structures. *J. ACM*, 22(4):596–612, October 1975.

[71] Richard M. Karp and Raymond E. Miller. Parallel program schemata: A mathematical model for parallel computation. In *Switching and Automata Theory, 1967. SWAT 1967. IEEE Conference Record of the Eighth Annual Symposium on*, pages 55–61, Oct 1967.

[72] W. E. Kluge and K. Lautenbach. The orderly resolution of memory access conflicts among competing channel processes. *IEEE Trans. Comput.*, 31(3):194–207, March 1982.

[73] F. Kordon, A. Linard, M. Becutti, D. Buchs, L. Fronc, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, E. Paviot-Adet, F. Pommereau, C. Rodrígues, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf. Web report on the model checking contest @ petri net 2013, available at http://mcc.lip6.fr, June 2013.

[74] Fabrice Kordon, Alban Linard, Marco Beccuti, Didier Buchs, Lukasz Fronc, Lom-Messan Hillah, Francis Hulin-Hubard, Fabrice Legond-Aubry, Niels Lohmann, Alexis Marechal, Emmanuel Paviot-Adet, Franck Pommereau, César Rodríguez, Christian Rohr, Yann Thierry-Mieg, Harro Wimmel, and Karsten Wolf. Model checking contest @ petri nets, report on the 2013 edition. *CoRR*, abs/1309.2485, 2013.

[75] Bernd Kramer. Stepwise construction of non-sequential software systems using a net-based specification language. In G. Rozenberg, editor, *Advances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*, pages 307–330. Springer Berlin Heidelberg, 1985.

[76] Lars M. Kristensen, SÃžren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[77] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal*

proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22, 2009.

[78] Olaf Kummer and Frank Wienberg. Renew - the reference net workshop. In *Petri Net Newsletter*, pages 12–16, 2000.

[79] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[80] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[81] N.G. Leveson and Janice L. Stolzy. Safety analysis using petri nets. *Software Engineering, IEEE Transactions on*, SE-13(3):386–397, March 1987.

[82] Su Liu, Reng Zeng, and Xudong He. Bounded model checking high level petri nets in pipe+verifier. Submitted.

[83] Su Liu, Reng Zeng, and Xudong He. Pipe+ - a modeling tool for high level petri nets. *International Conference on Software Engineering and Knowledge Engineering (SEKE11)*, pages 115–121, 2011.

[84] Su Liu, Reng Zeng, Zhuo Sun, and Xudong He. Samat - a tool for software architecture modeling and analysis. In *SEKE*, pages 352–358, 2012.

[85] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proc. IEEE/IFIP Working Conf. Software Architecture (WICSA)*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer Academic Publishers.

[86] Marko Makela. Maria: Modular reachability analyser for algebraic system nets, 2002.

[87] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[88] M.Ajmone Marsan and G. Chiola. On petri nets with deterministic and exponentially distributed firing times. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1987*, volume 266 of *Lecture Notes in Computer Science*, pages 132–145. Springer Berlin Heidelberg, 1987.

[89] Kjeld H. Mortensen. Efficient data-structures and algorithms for a coloured petri nets simulator. In *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01) Kurt Jensen (Ed.)*, pages 57–74. DAIMI PB-554, Aarhus University, August 2001.

[90] Leonardo Moura and Nikolaj Bjørner. Formal methods: Foundations and applications. chapter Satisfiability Modulo Theories: An Appetizer, pages 23–36. Springer-Verlag, Berlin, Heidelberg, 2009.

[91] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[92] Tomohiro Murata, N. Komoda, Kuniaki Matsumoto, and Koichi Haruna. A petri net-based controller for flexible and maintainable sequence control and its applications in factory automation. *Industrial Electronics, IEEE Transactions on*, IE-33(1):1–8, Feb 1986.

[93] J. D. Noe. A petri net model of the CDC 6400. In *Proc. of the ACM/SIGOPS Workshop on Systems Performance Evaluation*, pages 362–378, 1971.

[94] M.T. Ozsu. Modeling and analysis of distributed database concurrency control algorithms using an extended petri net formalism. *Software Engineering, IEEE Transactions on*, SE-11(10):1225–1240, Oct 1985.

[95] S. K. Paranjpe, A. B. Ektare, and D. P. Mital. Fault diagnosis of alignment networks using petri nets. *Int. J. Electron. (GB)*, 56(3):365–370, Mar, 1984.

[96] C.V. Ramamoorthy and G.S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *Software Engineering, IEEE Transactions on*, SE-6(5):440–449, Sept 1980.

[97] F. J. Rammig. Hierarchical modulator description of VLSI systems. In *Workshop Report. VLSI and Software Engineering Workshop*, pages 112–116, Silver Spring, MD, USA, 1983. IEEE Comput. Soc. Press.

[98] W. Reisig. *Petri nets: an introduction.* EATCS monographs on theoretical computer science. Springer-Verlag, 1985.

[99] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Proc. 17th Int. Conf. Conceptual Modeling (ER'98*, pages 449–464. Springer, 1998.

[100] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23:31–39, March 2006.

[101] T. Smigelski, Tadao Murata, and Masahiro Sowa. A timed petri net model and simulation of a dataflow computer. In *International Workshop on Timed Petri Nets*, pages 56–63, Washington, DC, USA, 1985. IEEE Computer Society.

[102] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[103] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Vs3: Smt solvers for program verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 702–708, Berlin, Heidelberg, 2009. Springer-Verlag.

[104] S. Stepney. *An Electronic Purse: Specification, Refinement, and Proof.* Technical monograph. Oxford University Computing Laboratory, Programming Research Group, 2000.

[105] Aaron Stump, Clark W. Barrett, and David L. Dill. A decision procedure for an extensional theory of arrays. In *In 16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society, 2001.

[106] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, New York, NY, USA, 1994.

[107] Jeffrey D. Ullman. *Elements of ML programming (ML97 ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[108] J. van Benthem. *The Logic of Time: A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse.* Synthese Library. Springer, 1991.

[109] W.M.P. van der Aalst. *Time Coloured Petri Nets and Their Application to Logistics.* Springer-Verlag, 1992.

[110] Margus Veanes, Nikolaj Bjørner, and Alexander Raschke. An smt approach to bounded reachability analysis of model programs. In *FORTE*, pages 53–68, 2008.

[111] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

[112] K. Voss. Using predicate/transition-nets to model and analyze distributed database systems. *IEEE Trans. Softw. Eng.*, 6(6):539–544, November 1980.

[113] Jiacun Wang, Xudong He, and Yi Deng. Introducing software architecture specification and analysis in sam through an example. *Information & Software Technology*, 41(7):451–467, 1999.

[114] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, September 1990.

[115] Jim Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, 2006.

[116] Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The certification of the mondex electronic purse to itsec level e6. *Formal Asp. Comput.*, 20(1):5–19, 2008.

[117] Reng Zeng and Xudong He. Analyzing a formal specification of mondex using model checking. In *ICTAC*, pages 214–229, 2010.

[118] Reng Zeng and Xudong He. Analyzing a formal specification of mondex using model checking. In *ICTAC*, pages 214–229, 2010.

[119] Reng Zeng, Jianling Liu, and Xudong He. A formal specification of mondex using sam. In *Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on*, pages 97 –102, dec. 2008.

[120] Reng Zeng, Jianling Liu, and Xudong He. A formal specification of mondex using sam. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 97–102, Washington, DC, USA, 2008. IEEE Computer Society.

[121] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723 – 744, 2010.

Appendix

BNF For Restricted First-order Logic

- sentence ::= formula
- formula ::= atomicFormula | compoundFormula | complexFormula
- atomicFormula ::= NOT formula | term
- compoundFormula ::= formula AND formula | formula OR formula | formula IMP formula | formula EQUIV formula
- complexFormula ::= quantifier userVariable domain variable DOT LPAREN formula RPAREN
- term ::= constant | variable | empty | exp
- terms ::= term termRests
- termRest ::= COMMA term
- termRests ::= | termRests termRest
- exp ::= arith_exp | rel_exp | set_exp | LPAREN exp RPAREN
- rel_exp ::= term EQ term | term NEQ term | term GT term | term LT term | term GEQ term | term LEQ term | term IN term: | term NIN term
- arith_exp ::= term MINUS term | term PLUS term | term MUL term | term DIV term | term MOD term | MINUS term UMINUS
- set_exp ::= term UNION term | term DIFF term | LBRACE term RBRACE | LBRACE LBRACK terms RBRACK RBRACE
- variable ::= ID | ID LBRACK index RBRACK

- userVariable ::= ID
- quantifier ::= FORALL | EXISTS | NEXISTS
- domain ::= IN | NIN
- constant ::= TRUE | FALSE | NUM | STR
- empty ::= EMPTY
- index ::= NUM

VITA

SU LIU

1985	Born, Yueyang, Hunan, China
2008	B.E., Software Engineering Sun Yat-sen University Guangzhou, China
2008–2014	Doctoral Candidate, Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Su Liu, Reng Zeng, Zhuo Sun, Xudong He. *Bounded Model Checking High Level Petri Nets in PIPE+Verifier.* Accepted by International Conference on Formal Engineering Methods, 2014

Reng Zeng, Zhuo Sun, Su Liu, Xudong He. *A Method for Improving the Precision and Coverage of Atomicity Violation Predictions.* submitted to International Conference on Runtime Verification, 2014, under review

Su Liu, Reng Zeng, Xudong He. *SAMTools - A Tool for Software Architecture Modeling and Analysis in SAM.* submitted to International Journal of Software Engineering and Knowledge Engineering, under review

Francisco R.Ortega, Armando Barreto, Naphtali Rishe, Malek Adjouadi, Su Liu. *Exploring Modeling Language for Multi-Touch Systems using Petri Net.* The ACM Interactive Tabletops and Surfaces Conference, pages 361-364, 2013

Su Liu, Reng Zeng, Zhuo Sun, Xudong He. *SAMAT - A Tool for Software Architecture Modeling and Analysis.* International Conference on Software Engineering and Knowledge Engineering, pages 352-358, 2012.

Reng Zeng, Zhuo Sun, Su Liu, Xudong He. *McPatom: A Predictive Analysis Tool for Atomicity Violation Using Model Checking.* International SPIN Workshop on Model Checking of Software, pages 191-207, 2012.

Su Liu, Reng Zeng, Xudong He. *PIPE+ - A Modeling Tool for High Level Petri Nets*. International Conference on Software Engineering and Knowledge Engineering, pages 115-121, 2011.

Reng Zeng, Yu Huang, Su Liu, Peter J. Clarke, Xudong He, Gwendolyn W. van der Linden, Jon L.Ebert. *SC-xScript: An Embedded Script Language for Scientific Computation in Embedded Systems*. International Conference on Software Engineering and Knowledge Engineering, pages 308-314, 2011.