7-1-2020

Modeling and Analyzing Cyber-Physical Systems Using Hybrid Predicate Transition Nets

Dewan Mohammad Moksedul Alam
Florida International University, dalam004@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

MODELING AND ANALYZING CYBER-PHYSICAL SYSTEMS USING HYBRID PREDICATE TRANSITION NETS

A dissertation submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Dewan Mohammad Moksedul Alam

2020

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Dewan Mohammad Moksedul Alam, and entitled Modeling and Analyzing Cyber-Physical Systems Using Hybrid Predicate Transition 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.

Jason Liu

Peter J. Clarke

Armando Barreto

Leonardo Bobadilla

Xudong He, Major Professor

Date of Defense: July 1, 2020

The dissertation of Dewan Mohammad Moksedul Alam is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2020

DEDICATION

To my family members for their supports and sacrifices

ACKNOWLEDGMENTS

I would like to thank each and everyone I have shared my research ideas. Whatever small the discussions were, some of them were eye-opening. Many ideas, solutions to little problems stemmed from them.

I would like to reserve special thanks to my committee members Dr. Armando Barreto, Dr. Jason Liu, Dr. Leonardo Bobadilla, and Dr. Peter J. Clarke. Every communication to them gave me a feeling that I am being cared which inspired me to be positive. Also, their suggestions and guidelines helped me to be on track.

This work was partially supported by AFRL under FA8750-15-2-0106. I also received TAship from the SCIS department at FIU. Many thanks to both AFRL and the SCIS department for providing me financial support.

I am grateful to my family members, my son Aymaan, daughter Amaya, and wife Shamsia. They suffered and sacrificed a lot due to my pursuit of this degree. Without their consideration, it would not be possible for me to come to this end.

Last but not the least, I have no word to express my gratitude to my major professor Xudong He. He was always there for me, all the time. He endured my ignorance, negligence, and stubbornness. Above all, he was patient enough to discuss anything with enthusiasm and brought me in the right direction whenever I needed it. Without his support, it would be very difficult for me to complete this.

ABSTRACT OF THE DISSERTATION

MODELING AND ANALYZING CYBER-PHYSICAL SYSTEMS USING HYBRID PREDICATE TRANSITION NETS

by

Dewan Mohammad Moksedul Alam

Florida International University, 2020

Miami, Florida

Professor Xudong He, Major Professor

Cyber-Physical Systems (CPSs) are software controlled physical devices that are being used everywhere from utility features in household devices to safety-critical features in cars, trains, aircraft, robots, smart healthcare devices. CPSs have complex hybrid behaviors combining discrete states and continuous states capturing physical laws. Developing reliable CPSs are extremely difficult. Formal modeling methods are especially useful for abstracting and understanding complex systems and detecting and preventing early system design problems. To ensure the dependability of formal models, various analysis techniques, including simulation and reachability analysis, have been proposed in recent decades. This thesis aims to provide a unified formal modeling and analysis methodology for studying CPSs.

Firstly, this thesis contributes to the modeling and analysis of discrete, continuous, and hybrid systems. This work enhances modeling of discrete systems using predicate transition nets (PrTNs) by fully realizing the underlying specification through incorporating the first-order logic with set theory, improving the type system, and providing incremental model composition. This work enhances the technique of analyzing discrete systems using PrTN by improving the simulation algorithm and its efficient implementation. This work also improves the analysis

of discrete systems using SPIN model checker by providing a complete and more accurate translation method.

Secondly, this work contributes to the modeling and analysis of hybrid systems by proposing an extension of PrTNs, hybrid predicate transition nets (HPrTNs). The proposed method incorporates a novel concept of token evolution, which nicely addresses the continuous state evolution and the conflicts present in other related works. This work presents a powerful simulation capability that can handle linear, non-linear dynamics, transcendental functions through differential equations. This work also provides a complementary technique for reachability analysis through the translation of HPrTN models for analysis using SpaceEx.

Finally, several well-known CPSs are modeled and analyzed to demonstrate the effectiveness and applicability of the proposed methodology, which include a system with complex dynamics defined using the second-order differential equation, a system with multiple non-linear dynamics, and a system composed of hybrid components. All the improvements and proposed methods are fully realized in the PIPE+ tool environment.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A Cyber-Physical System (CPS) is an ecosystem that combines and integrates heterogeneous components providing cyber features (communication, computing, and control) with physical devices, which work together to accomplish specific goals. For instance, many safety features in modern cars like the anti-lock braking system (ABS), cruise control, vehicle stability assist (VSA), electronic brake distribution (EBD) are some examples of cyber-physical systems. With the technological breakthroughs in recent years, these computing devices have become more sophisticated, powerful, and embeddable. With this advancement, these computing devices have become the most acceptable choice to be used as controllers, and mechanical controllers are being replaced with digital controllers everywhere. From utility features in household devices, like smart refrigerator, smart air-conditioning to safety-critical features in cars, trains, aircraft, robotics, smart healthcare devices, smart grids, manufacturing process control, collision avoidance in avionics they are being used.

With the proliferation of CPS powered systems, the reliance on these systems for safe operations is ever-growing. So is the expectation of reliability and correctness. Especially for systems performing safety-critical tasks, it is imperative to ensure that these systems are working correctly. Fulfilling this expectation is extremely difficult due to the involvement of complex, multi-modal, multi-domain, and physical components. This difficulty is magnified when different components exhibit different fundamental behavior. A typical strategy to verify the safe behavior is to analyze and predict the components' behavior to see whether they always remain in the defined safe zones when started within a safe state. Finally, it is also analyzed how the behavior of one component affects the behavior of the other.

The first step to analyze a system is to model its behavior. However, it is impractical to carry out the analyses on the actual systems. Instead, we need a representation or model of the system that depicts the high-level description of the essential system behavior. These models are then analyzed to decide whether they are guaranteed to meet the requirements. Several different models have been proposed in the literature to capture the behavior of varying nature of different systems. Among other models, Hybrid Systems [2] have been the focus of intense research in the past few decades. Hybrid systems are a mathematical model that combines discrete dynamics with continuous dynamics. In essence, these are one particular case of CPSs, where only the nature of the dynamics is focused, and all other subtleties are abstracted away.

One early prominent work is the hybrid automata [3] that provides a concrete mathematical framework for the analysis and verification of hybrid systems. Hybrid automata integrate diverse models such as differential equations and state machines in a single formalism with a uniform mathematical semantics and novel algorithms for multi-modal control synthesis and safety and real-time performance analysis [2]. However, despite providing powerful methods to analyze hybrid systems, the major inconvenience of hybrid automata is the dramatic increase of model dimensions for complex systems due to the intrinsic global state configurations and sequential behaviors of hybrid automata.

Petri nets (also known as low-level Petri nets), a concurrent and distributed formal method, provide great flexibility to model complex discrete reactive systems. To model continuous dynamical systems, Petri nets have been evolved towards continuous Petri nets [4]. Continuous Petri nets have been extended to hybrid Petri nets [5] for modeling hybrid systems. Hybrid Petri nets inherit the advantages of the Petri net model, such as capturing distributed behaviors, concurrency, synchronization,

and conflicts. However, similar hybrid automata, modeling complex systems using low-level Petri nets, is extremely difficult. Also, low-level Petri nets suffer from the state explosion problem. To solve this problem, several extensions to low-level Petri nets are introduced. These are commonly known as high-level Petri nets.

High-level Petri nets (HLPNs) are powerful formal methods for modeling concurrent and distributed systems. HLPNs provides a graphical representation of systems to make them easier to understand. They offer strong expressive power through rich data abstraction, algebraic expressions, and logic formulas to define system functionality—furthermore, their dynamic semantics support model level simulation. As a result, they are being used widely in system modeling in many application domains. They are also being studied extensively. Numerous extensions to these formalisms are available to model different types of systems. Predicate transition nets [6], Colored Petri nets [7], algebraic Petri nets [8] are widely used classes of high-level Petri nets. HLPNs are further extended towards hybrid high-level Petri nets to model hybrid systems [9, 10, 11].

Despite providing great flexibility to model hybrid systems, hybrid high-level Petri nets, similar to all class of high-level Petri nets, are very hard to analyze. Also, there is no effective tool available to verify the correctness of hybrid high-level Petri nets models mathematically. However, many sophisticated tools are available to analyze and verify the safety properties of hybrid system models build using hybrid automata. In recent years, *SpaceEx*[12] has gained great success in providing reachability analysis for hybrid system models.

The Problem. In the previous paragraphs, two fundamentally different techniques are mentioned that are the focus of extensive research during the last few decades. One of them, hybrid automata, provides numerous efficient analysis techniques but modeling with it is challenging. On the other hand, high-level Petri nets

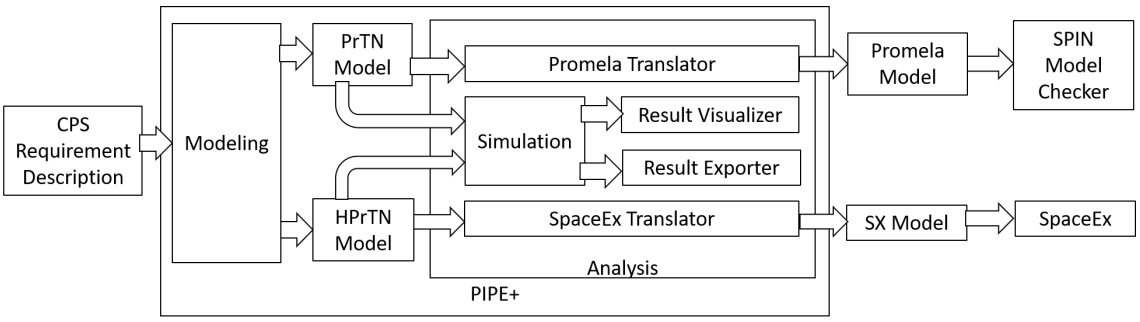

Figure 1.1: A unified modeling and analysis methodology supported in PIPE+

are well suited for modeling concurrent and distributed system control and supporting synchronous and asynchronous communication; however them lack of efficient analysis techniques except simulation. In this settings, under this thesis, we seek to find out - (1) how to develop a unified modeling methodology that allows us to model a wide range of CPSs, and (2) how to provide effective analysis techniques, within the same methodology, to ensure the dependability of these systems.

Preliminary Results. In previous works along this line [13, 14, 15] a class of predicate transition nets (PrTN) is realized in the PIPE+ [13] tool environment. The adopted class of PrTN is slightly different than what originally defined in [6]. Rather it adopts the concepts of hierarchical predicate transition net [16], algebraic Petri nets [8], and uses first-order logic formulas to model the behavior. These provide a powerful technique to model a wide range of discrete systems. For analysis, it provides simulation and translation-based model checking techniques where the PrTN models are translated into the input language of some well-known model checkers for discrete systems. However, the realization of PrTN in PIPE+ and some translation methods have many limitations and severe flaws. Under this study, those limitations are resolved. The modeling capability in PIPE+ is enhanced in many ways. Also, several new methods of translation of PrTN are introduced [17, 18]. Thus, a unified methodology is developed that brings powerful modeling methods

and sophisticated analysis techniques for discrete systems within the same methodology. Figure 1.1 shows an overview of the workflow of this methodology.

New Contributions. The methodology resulted from the preliminary works provides a sophisticated method to model and analyze discrete systems. Under this study, the concepts of this methodology are further extended for hybrid systems. Here, the concepts of PrTN, as realized in PIPE+, are extended to make it capable of modeling hybrid systems and introduced as Hybrid Predicate Transition Nets (HPrTN) [19, 20]. For analysis, along with simulation, a translation-based technique is developed. These works made the following major contributions:

1. An improved modeling method for discrete event systems using PrTN by fully realizing the underlying algebraic specification and providing new capabilities
2. An enhanced analysis techniques for discrete systems by improving the simulator and providing a more accurate translation method for model checking using SPIN model checker
3. A new definition of hybrid predicate transition nets (HPrTNs) to model cyber-physical systems
4. An effective analysis technique for cyber-physical systems model using HPrTNs leveraging the state of the art model checker SpaceEx [12] for hybrid systems through model translation
5. The redesign and enhancement of the tool PIPE+ to support the modeling and analysis of CPSs using HPrTNs.

The following subsections provide some overview of these contributions.

Modeling discrete systems. While in theory, PrTNs support all the data types and functionality provided by an underlying algebraic specification. In prac-

tice, only a limited algebraic specification was realized. This dissertation generalizes PrTN definition used in PIPE+ [21] with several new features, which include:

- Improved type system by introducing more generic type to model the numerical attributes.
- New mathematical expressions that increased the expressiveness of PrTN
- Full First-Order Logic formulas with quantifiers, and set-operations.
- Incremental modeling through model composition.
- A logical clock to model time-dependent behaviors.

Analyzing discrete systems. The method to analyze PrTN models is improved by providing an enhanced simulation environment and implementing more robust and complete translation techniques from PrTN to Promela, the underlying modeling language for model checker SPIN. The simulation environment for PrTN in the PIPE+ tool is enhanced to provide better performance. Some of the key enhancements are - a new syntax tree designed to reduce the depth of the syntax tree for any formula by half, run-time efficiency of the evaluation of the transition constraints is improved through memoization of the intermediate results during simulation, and so on. For model checking with SPIN, a complete translation scheme is implemented where each element in PrTN is translated to equivalent form in Promela. The transition constraint evaluation is emulated the same as the simulator in PIPE+. Apart from these, additional translation schemes are implemented to capture agent-oriented models. These improvements, along with other minor modifications, are discussed further in chapter 3.

Modeling Hybrid Systems. To model CPSs, hybrid predicate transition nets (HPrTNs), a new class of hybrid high-level Petri nets is introduced. HPrTN is

an extension to PrTN with additional new functionalities to incorporate models of components containing continuous dynamics with discrete components and new semantics to define the evolution of continuous components and resolving conflicts between the discrete and continuous components. Important features of HPrTNs include continuous places, token evolution through differential equations, logical time, and net composition. Some preliminary results in incorporating some continuous features into predicate transition nets (PrTNs) for modeling and analyzing hybrid systems are presented in [19]. In [20] these concepts are extended and refined further to (1) provide a complete formal definition of hybrid predicate transition nets (HPrTNs), (2) show the formal relationships between HPrTNs and hybrid automata, (3) eliminate the conflicts between continuous and discrete components by removing the continuous transitions and associating differential equations to continuous places to more accurately reflect the token evolution, and (4) model and analyze several new benchmark hybrid systems using HPrTNs. Chapter 4 discusses these concepts in detail.

Analyzing Hybrid Systems. The support for analyzing the hybrid systems are two-fold - simulation and reachability analysis using state of the art model checker. The simulation of HPrTN models needs simulation of both discrete and continuous components. Although the discrete and continuous components share much of the structural features of the HPrTN, they have completely different dynamic semantics requiring different simulation mechanisms. To facilitate the simulation of the continuous components, several new concepts are introduced. Some of the important aspects are summarized here.

- **New scheduling mechanisms**. The connected components are thought to be evolving at all times. To simulate this behavior, a new scheduler is in-

troduced, which periodically evaluates the connected components' constraints and computes the evolution of the connected components.

- **Differential equation solver**. Incorporation of differential equation solvers. The computation of the evolution of the continuous components depends on solving the differential equations specifying the behavior. To solve these equations, a well-known and community accepted Python library is used. To communicate between Java and Python process, a light-weight inter-process communicator is introduced. This communicator, takes the commands and inputs to Python process, pythons process execute the input, and returns the result.
- **Token evolution**. The evolution rules (differential equations) of the continuous components are associated either with the continuous places or with the tokens within those. The evolution of these components is computed by solving those equations and updating the corresponding tokens. This is termed a token evolution. It is similar to transition firing but fundamentally different.
- **Visualization of evolution**. To help understand the evolution, a result visualizer is introduced, which provides a chart-based visualization mechanism. The user can configure charts against different continuous attributes. During the simulation, the results are plotted according to the chart configurations. The results also are visualized using a third-party tool that can be performed—the result exporter help doing it. The result exporter exports the results to the configured medium.
- **Translation of HPrTN**. To support the analysis of HPrTN, a translation based method is developed. Here, the HPrTN models can be translated into the input language of SpaceEx. SpaceEx is one of the states of the art reacha-

bility analysis tools. The translated models can be directly used with the tool to perform analysis.

Redesigning of PIPE+. The tool PIPE+ provides full support of the concepts of both PrTN and HPrTN definitions. Furthermore, PIPE+ had undergone a significant redesign and rebuilt. The previous version of PIPE+ has several flaws. Some of these are as follows

- PIPE+ was built on top of another app PIPE [22], which was built using an older version of Java and related libraries based on that version. Many of the used functionalities are either obsolete or deprecated.
- PIPE focused on modeling and analysis of low-level Petri nets. In PIPE+, all the existing functionalities of PIPE were preserved alongside with the features modified for PrTN. This caused a lot of confusion about choosing the right functionalities for PrTN.
- The design of PIPE+ was not transparent. Different components were interconnected to one another. As a result, modification to one functionality requires a lot of changes.
- All the operations were performed on the main thread. As a result, the UI used to become frozen if the current task takes a long time to finish. One example of such a task is the multi-step simulation. Once a multi-step simulation is started, the UI becomes available again to interact with only when all the steps are performed.

The redesigned PIPE+ addresses and eliminates the identified flaws. This new design makes use of the latest java libraries adopts layered component-based architecture with independent modules for the data model, core functionalities, and

presentation. This new design offers an API layer intended to be used as the integration point of the core modules and the application layers. This new design also provides means for the developers to extend the tool's functionalities along with the inclusion of new features.

The rest of this thesis is organized as follows. Chapter 2 discusses some background on PrTN (the base modeling formalism used in this thesis), the improvements made to the modeling formalism regarding discrete event systems. Chapter 3 provides a comprehensive discussion on the analysis techniques for discrete event systems. Chapter 4 presents a new definitions of hybrid predicate transition nets for modeling hybrid systems in which, the continuous place is introduced to model continuous states and the evolution is defined using differential equations. Chapter 5 presents techniques to analyze hybrid systems, in which simulation technique based on the dynamic semantics of HPrTNs is used to analyze systems with non-linear hybrid behaviors and a complementary reachability analysis technique is used to analyze systems with piece-wise linear hybrid behaviors. Chapter 6 discusses motivations to redesign PIPE+ and the new proposed architecture. Chapter 7 concludes the thesis and guidelines for future development.

CHAPTER 2

MODELING DISCRETE EVENT SYSTEMS

This chapter focuses on modeling of discrete event systems (DES) using predicate transition nets (PrTNs). First, it provides an overview on DES and Petri nets in general. The later part is organized as - section 2.2 gives formal definition of PrTN and its dynamic semantics; section 2.3 discusses some methods to model DESs using PrTNs; some DESs are modeled in section 2.4; and in section 2.5, some significant improvements and new features incorporated in PIPE+ are discussed. This chapter is partially based on the publications [17], and [18].

2.1 Overview

In discrete event systems (DES), the state remains unchanged until an event occurs. A DES evolves from event to event. Events are chronological and autonomous. Some events trigger some others, but it is uncertain at what time. Furthermore, any event can block, freeze, delay, enable/disable future events. Therefore, discrete event systems are stochastic, dynamic, and asynchronous. Modeling of DES, thus, requires a formalism that can capture all the events of the system and the causal relationship among them to describe the whole system behavior [23].

Petri nets are very effective in describing distributed and concurrent systems. They also provide powerful tools in describing systems in terms of events and activities. Models of discrete event systems are mostly based on the concepts of events and activities. Therefore, Petri nets are the modeling formalism that enables us to represent a DES naturally. The following paragraph provides an informal description of Petri net.

Graphically, a Petri net is represented as a directed and weighted bipartite graph with two kinds of nodes, namely **places** and **transitions**. Places can connect to

only transitions or vice verse via edges called **arcs**. Places contain a positive integer number of **tokens**. The tokens in places are collectively called **marking** of the net and constitute the state of the system. On the other hand, transitions are associated with the events of the system. The behavior of the system is generated by **firing** of the enabled transitions, i.e., the occurrence of the actions when associated event happens. A transition can fire when it is enabled. Non-conflicting enabled transitions can fire at the same time.

The locality of determining the enabledness of transitions and the firing of concurrent transitions make Petri nets a well suited tool for studying distributed and concurrent systems. However, the simplicity of Petri nets makes it difficult for them to model systems with complex data structures and functional processing. To solve these problems, several extensions to Petri net formalism are introduced. These are commonly known as high-level Petri nets. Predicate Transition Nets (**PrTN**s) are a class of high-level Petri nets.

This chapter focuses on modeling discrete event systems using PrTNs. First it provides a formal definition of PrTNs to describe the structure and dynamic semantic of PrTN. Then a modeling methodology is provided for modeling discrete events systems using PrTN. Then the major contributions to enhance the modeling capability are discussed.

2.2 Predicate Transition Nets (PrTNs)

Predicate Transition Nets [24, 6, 25] are a class of high-level Petri nets where places are viewed as logical predicates instead of propositions. Places can be associated with a data type to define complex structured type data the places can hold. Transitions are specified using first order logic formulas that further constrain the enabling

conditions. Specifically, the enabledness of a transition is now determined by (1) the availability of the appropriate tokens in its input places, and (2) the tokens in the input places must satisfy its constraints. The dynamic behavior of PrTNs consists of all execution sequences where each execution may contain firings of multiple transitions. Formal definition of the structure and the behavior of PrTNs are discussed in the following subsections.

2.2.1 Formal Definition

Definition 2.2.1 (PrTN)**.** A PrTN is a tuple $N = (P, T, F, \alpha, \beta, \gamma, M_0)$, where,

1. P is a non-empty finite set of places.
2. T is a non-empty finite set of transitions, which disjoins P, i.e., $P \cap T = \emptyset$;
3. $F \subseteq (P \times T \ \cup T \times P)$ is a flow relation (the arcs of N);
4. $\alpha : P \to Type$ associates each place p in P with a *type* in $Type$. $Type$ defines the structure of the data the places can hold. It consists of basic types such as $String$, $Integer$, and $Real$ Numbers, and composite types defined using Cartesian product and power set;
5. $\beta : T \to Constraint$ associates each transition t in T with a constraint. Each constraint is a disjunction $\bigvee_i d_i$ for $i \geq 1$, where each disjunct d_i is a conjunction $pre_i \wedge post_i$ that define the enabling condition (precondition) and the processing result (post-condition) of a case of t respectively. The precondition contains only the variables appearing in the labels of incoming arcs, and the post-condition contains the variables appearing in the labels of outgoing arcs. The post-condition associated with continuous transitions may contain differential equations indicating the change rates;

6. $\gamma : F \rightarrow Label$ associates each arc f in F with a label in the form of a simple variable x or a set element $\{x\}$. An arc label denotes the data flow of a relevant transition, where the variable is instantiated with concrete token(s) during the transition firing;

7. $M_0 : P \rightarrow Token$, the initial marking, associates each place p in P with a set of tokens. Tokens in $M_0(p)$ are values respecting the type of p.

2.2.2 Dynamic Semantics

The dynamic semantics of HPrTNs are defined using the concept of markings (states) $M : P \rightarrow Token$ that are mappings from places to tokens.

Definition 2.2.2 (Enabledness of a tansition)**.** A transition t in T is enabled in a marking M if one of its precondition is true. Formally, $\forall p \in P.(\overline{\gamma}(p,t) : \theta \subseteq M(p) \wedge \exists i.\beta(t).pre_i : \theta)$, where $\overline{\gamma}(p,t)$ is a generalization of γ such that $(p,t) \notin F \Rightarrow \overline{\gamma}(p,t) = \emptyset$. $e : \theta$ is the result of instantiating all arc variables with tokens in p according to substitution θ.

Definition 2.2.3 (Firing of a transition)**.** An enabled transition t in marking M with substitution θ satisfying $\beta(t)$ is firable if it is not in conflict with another firable transition. The firing of a firable transition results in a new marking M' defined by: $\forall p \in P.(M'(p) = M(p) \cup \overline{\gamma}(t,p) : \theta - \overline{\gamma}(p,t) : \theta)$. We denote this firing as $M \xrightarrow{t/\theta} M'$

Definition 2.2.4 (Conflict and Conflict Resolution)**.** Two enabled transitions are in conflict if the firing of one of them disables the other. This conflict is resolved by selecting one randomly to fire.

Definition 2.2.5. Let T_i be a set of concurrently enabled non-conflict transitions with corresponding substitutions θ_i in marking M_i, and M_{i+1} is the resulting new

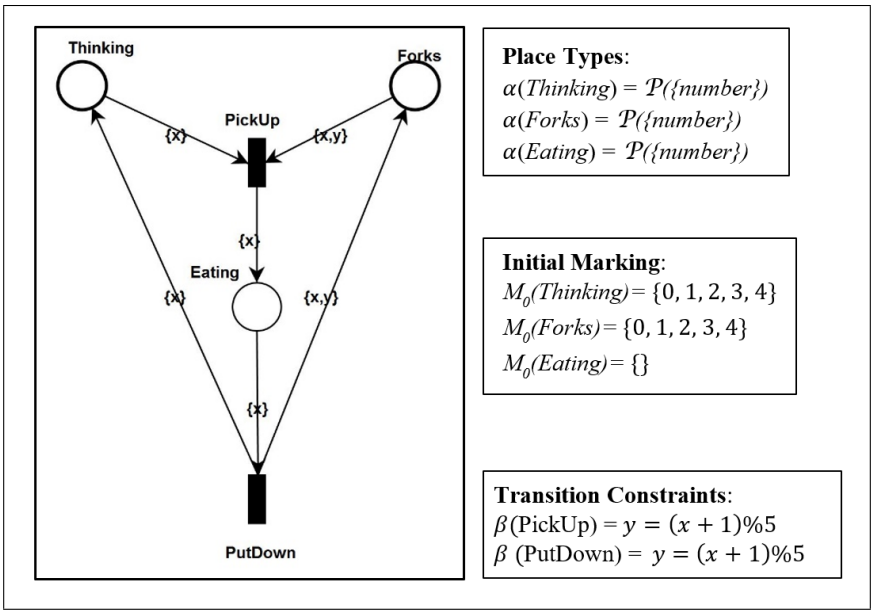

Figure 2.1: PrTN model of Five Dining Philosophers problem

marking after firing transitions concurrently. This transition step is denoted as $M_i \xrightarrow{T_i/\theta_i} M_{i+1}$. The behavior of a net N consists of the set of all the firing sequences $M_0 \xrightarrow{T_0/\theta_0} M_1 ... M_i \xrightarrow{T_i/\theta_i} M_{i+1} ....$ The set of all reachable markings is denoted as $[M_0 >$.

Here, an instance of a PrTN model is illustrated using the well-known five dining philosopher problem. Figure 2.1 shows model along with its inscriptions. The model has three places *Thinking*, *Fork*, and *Eating* and two transitions *Pickup* and *Putdown*. The places *Thinking* and *Eating* store the states of philosophers respectively, and place *Forks* store available forks. The data for this problem are philosophers and forks. The philosopher are assigned unique identifiers, the numbers from 0–4 in this example. The forks are also modeled similarly. All the places have the same datatype definition (α) power set of one-element token. This way they are designed to hold multiple tokens. Each token consists of a number. The number represents the identifier of a philosopher, thus each number models a philosopher.

Two transitions *PickUp* and *PutDown* model the possible actions of philoso-

phers, (1) acquire left and right forks, and (2) put down the forks respectively. The transition $Pickup$ has two input variables (arc labels γ) x and y. It is constrained by relational expression $y = (x + 1)\%5$. It enforces that for a philosopher x in the place $Thinking$, the two forks x (left) and $(x + 1)\%5$ (right) must be available in the place $Forks$. If this condition matches for some x, then the transition $PickUp$ will be enabled and ready to fire. If it fires then the token representing the philosopher x is transfered from $Thinking$ to $Eating$. The tokens representing the forks x and $(x + 1)\%5$ are removed from $Forks$ to make those unavailable. The transition $PutDown$ has one input variable x and two output variables x and y. Firing of this transition means that philosopher x finished eating, thus it should be removed from the place $Eating$ and put to $Thinking$. At the same time, the forks x and $(x+1)\%5$ should be made available by putting those back to the place $Forks$.

A careful reader might find that both $PickUp$ and $PutDown$ have same constraint but different outcome is expected. This is because the operator = (equals) has dual interpretations. In preconditions it is used as a relational operator and in postconditions it is used as an assignment operator. In $\beta(PickUp)$, both x and y are input variables. Their assigned values are known beforehand. In this case the operator = is interpreted as a relational operator and used to test the equality of the both sides. On the other hand, $\beta(PutDown)$ is a postcondition, since y is an output variable and is unknown until the transition fires. In this case, the operator = is used as an assignment operator and sets y with the value as computed by the expression on the other side.

In the initial marking M_0, all the tokens representing philosophers are in the place $Thinking$ and all the tokens representing forks are in the place $Forks$. The place $Eating$ does not have any token.

2.3 Model Development

2.3.1 Modeling States

PrTNs are well suited for modeling traditional concurrent and distributed systems with discrete behaviors. Discrete places constitute the discrete states. Each place represents a certain type of object or entity of the system. The attributes of an entity are modeled using datatypes. A datatype specifies the structure of the data the assigned places can have. The data structure is modeled using a multi-set of the sorts of basic data types, like $string, boolean, int, short, real$. PIPE+ supports only two types of sorts - *string* and *number*. The sort *string* represents string literals or text type data. The sort *number* is used to represent the numeric valued attributes, including both real numbers and integers. Previously, PIPE+ supported only *integer* numbers. Under this study, its capability is enhanced. This enhancement is discussed elaborately in section 2.5.1. If the entity has more than one attributes a multi-set of these two sorts is used. Finally, to model the scenario where there are more than one instances of the same entity may be present at the same time, the datatype of the corresponding place is marked as *powerset*. This allows that place to have any number tokens. An upper bound can be set to restrict number of allowed tokens for such places by setting the *capacity*.

2.3.2 Modeling Transitions

The discrete behavior of a system is modeled using the causal relationship between events and actions, and the flow relations. The events and actions are specified as the preconditions and post-conditions of the discrete transitions. The flow relations are specified by the connection among the places and the transitions via arcs. Both

preconditions and post-conditions are specified using first-order logic formulas. The tool PIPE+ supports full first-order logic formula, including all the logical and relational operations, quantifiers, basic arithmetic, and set operations. Table 2.1 shows the supported operators. All the operators preserve their original mathematical semantics. Apart from these basic operations, the inline $function$ construct is also supported in PIPE+. The function construct allows the modeler to compute any arithmetic operations that cannot be constructed otherwise. The tool comes with a default function interpreter that supports some frequently used arithmetic, trigonometric, string operations, and random number generators. It can also be extended to support other user-defined functions.

The preconditions and post-conditions are first-order logic formulas. These formulas can use constant terms or variable terms. The variables are derived from the arc labels. The variables derived from the labels of the incoming arcs are termed as input variables, and the variables derived from the labels of the outgoing arcs are called output variables. Some variables are not derived from arc labels. These are called *user variables* and are usually part of the *quantifiers* and *set operations*. The variables can be either input or output or both. The clauses in preconditions are mostly comparisons and contain only the input variables. Post-conditions define the values of output variables through the input variables and the relational equations in the first-order logic formulas. New values are generated by evaluating the expressions with the values held by input variables. If a variable represents a structured token with multiple fields, each field is accessed through indexing. For example, a variable x with a structured type $\langle string, string, number, number \rangle$ is accessed as $x[1]$ for its first field and so on.

Variables can be - (1) single-valued, or (2) multi-valued. Single valued variables can hold only one token. Multi-valued variables, on the other hand, hold a set of

Category	Operations	Symbols
Connective (Logical)	And	$\wedge$
	Or	$\vee$
	Not	$/$
	Implication	$\rightarrow$
	Equivalence	$\leftrightarrow$
Relational	Equals	$=$
	Not Equals	$\neq$
	Greater	$>$
	Less	$<$
	Greater or Equal	$\geq$
	Less or Equal	$\leq$
Algebraic (number)	Addition	$+$
	Subtraction	$-$
	Multiplication	$*$
	Division	$/$
	Remainder	$\%$
	Powers	$a\hat{}s$
Differential	Differential	δ
	Difference	Δ
	Time Symbol	τ
Predicate Logic	For All	$\forall$
	Exists	$\exists$
	Dot	$\cdot$
Parentheses	) (} {] [$\langle$ $\rangle$	

Table 2.1: Supported operations in PIPE+ and mapping symbols

tokens. If the data type of a variable, i.e., the datatype of the place associated with the variable, is not powerset, then the variable is by default single-valued. Otherwise, the variable can be either multi-valued or single-valued. A variable is specified inside a bracket, $\{x\}$, to be marked as single-valued. During execution, all the available tokens are assigned to a variable if it is multi-valued. Otherwise, one of the available tokens is assigned to the variable.

2.3.3 Modeling Time

The evolution of discrete systems depends on events rather than time. In real-time systems, events may be time-bound. Time-bound events can be modeled using timestamp carrying tokens and transition constraints for checking and updating token time stamps. The tool PIPE+ provides a simple means to access time. A special logical clock variable τ is used to define timing-related constraint in a first-order logic formula. PIPE+ initializes and maintains τ for each simulation run. By default, the logic clock starts with the timestamp 0 and increases by one unit after each execution step. Time increment frequency (probability) and the increment sizes are configurable. No specific time unit is assumed with the step size. It is modelers' responsibility to use τ consistently in all transition constraint definitions. Using this global logic clock variable τ greatly simplifies the resulting model structure for representing timing features.

2.3.4 Model Composition

For modeling complex systems, PIPE+ facilitates an incremental modeling approach. Here the whole system can be decomposed into smaller components. Each component can be modeled separately, and then can be merged to get the complete system model. Both synchronous and asynchronous composition is implemented. Synchronous composition of two PrTN models N_1 and N_2 is achieved by merging a place in N_1 with another in N_2. The asynchronous composition is achieved by merging a transition from N_1 with another in N_2. It can also be achieved by connecting a Place in N_1 and a transition in N_2 or vice verse.

2.4 Case Study

Several classic and benchmark discrete systems from the annual Petri net model checking contest 2015 [26] have been modeled in PIPE+ using the syntax and semantics of the PrTN. This section demonstrates the modeling of one of these systems. The *Bridge and Vehicles* system from the annual Petri net model checking contest 2015 is chosen for this purpose. In the contest, this system was modeled using Colored Petri net [7], a widely used high-level Petri net. The system is redefined using PrTN in PIPE+.

The *Bridge and Vehicles* system represents a single lane automated bridge that some motorized vehicles try to pass from both directions. The bridge has a limited capacity for a certain number of vehicles. The number of vehicles on the bridge can never be more than that capacity. A controller supervises the movement of the vehicles on the bridge. The controller ensures the safe passage of the vehicles and fair share of the bridge. The fair share of the bridge is ensured by limiting the number of vehicles from each side that can cross the bridge in a row. Here these constraints are discussed in terms of the tuple (V, P, N), where V is the number of vehicles on each side of the bridge trying to get to the other side, P is the maximum number of vehicles allowed on the bridge, and N is the maximum number of vehicles from the same side permitted to pass in a row. Here, P and N are the constraints of the system. P supports the capacity, and N ensures fairness of sharing the bridge.

There are several ways to model the system. In this case study, a bottom-up approach is used where each of the vehicles is identified separately. The flow is organized around the movement of the vehicles. The movement of the vehicles is categorized into several stages. As a vehicle approach the bridge, it needs to register with the system and then wait for its turn to get onto the bridge. When a vehicle

registers with the system, it can encounter one of the three situations - (1) there are other vehicles registered before it and waiting, (2) the bridge may be occupied with the vehicles from the opposite side, and (3) N is reached, and the direction is about to change. The waiting queue is maintained to accommodate these situations. When the system allows, the vehicle moves onto the bridge. Then at some point, it leaves the bridge. Thus, this model assumes four distinct phases of a vehicle - (1) available, (2) waiting, (3) on the bridge, and (4) exited.

These phases are modeled using separate *place*s. The change of phases is defined using separate *transition*s. The traffic movement from both sides is modeled using two different sets of such places and transitions. The controller also consists of a place and a transition. The place keeps track of the phase information, and the transition makes the switching between the sides. A pictorial diagram of the PrTN model of the system is shown in Figure 2.2. The places $RouteA$ and $RouteB$ hold the pool of vehicles that want to cross the bridge. $WaitA$ and $WaitB$ carry the tokens representing the vehicles registered with the system and waiting to move on the bridge. $OnBridgeA$ and $OnBridgeB$ hold the tokens representing the vehicles currently on the bridge. $ExitA$ and $ExitB$ hold the tokens for the vehicles that already crossed the bridge.

In this model, each vehicle is identified by a *string* literal. Thus the token representing a vehicle needs only one field. Again these places need to hold more than one tokens. Therefore, the datatypes of these places are a powerset of one element field of type *string*, as shown in Table 2.2. The places $NumberA$ and $NumberB$ hold the number of vehicles on the respective side waiting to move to the bridge. The place $Controller$ has the control information such as, vehicles from which side are currently on the bridge, how many are on the bridge now, how many vehicles crossed during this pass, and whether a switch is needed.

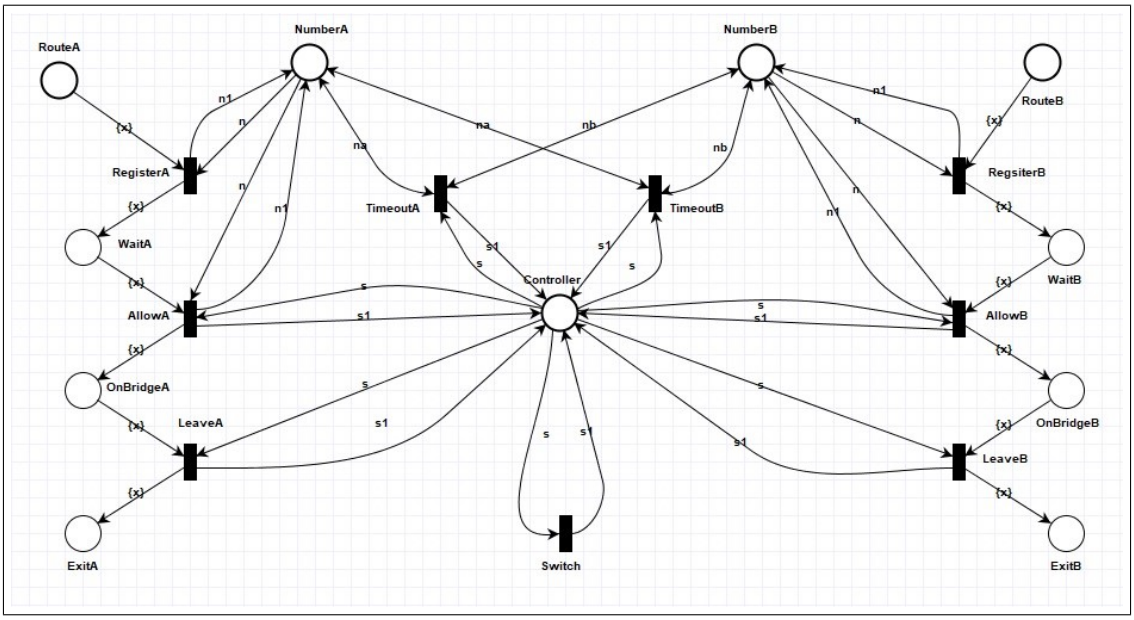

Figure 2.2: Pictorial diagram of a PrTN model of the Bridge system

$\alpha(RouteA) = \alpha(WaitA) = \mathcal{P}(string)$
$\alpha(ExitA) = \alpha(OnBridgeA) = \mathcal{P}(string)$
$\alpha(RouteB) = \alpha(WaitB) = \mathcal{P}(string)$
$\alpha(ExitB) = \alpha(OnBridgeB) = \mathcal{P}(string)$
$\alpha(NumberA) = \alpha(NumberB) = (number)$
$\alpha(Controller) = (string, number, number, string)$

Table 2.2: Data type definitions of the places

$M_0(RouteA) = \{\langle \text{“}Wa\text{”} \rangle, \langle \text{“}Xa\text{”} \rangle, \langle \text{“}Ya\text{”} \rangle, \langle \text{“}Za\text{”} \rangle\}$
$M_0(RouteB) = \{\langle \text{“}Wb\text{”} \rangle, \langle \text{“}Xb\text{”} \rangle, \langle \text{“}Yb\text{”} \rangle, \langle \text{“}Zb\text{”} \rangle\}$
$M_0(NumberA) = \langle 0 \rangle$
$M_0(NumberB) = \langle 0 \rangle$
$M_0(Controller) = \langle \text{“}A\text{”}, 0, 0, \text{“}N\text{”} \rangle$

Table 2.3: Initial marking of the net

$\beta(RegisterA) = (n1 = n + 1)$

$\beta(RegisterB) = (n1 = n + 1)$

$\beta(AllowA) = (n > 0 \wedge\ s[1] = \text{“}A\text{”} \wedge\ s[2] < 5 \wedge\ s[3] < 2 \wedge\ s[4] \neq \text{“}Y\text{”})$
$\quad \wedge (n1 = n - 1 \wedge\ s1 = \langle \text{“}A\text{”}, s[2] + 1, s[3] + 1, s[4] \rangle)$

$\beta(AllowB) = (n > 0 \wedge\ s[1] = \text{“}B\text{”} \wedge\ s[2] < 5 \wedge\ s[3] < 2 \wedge\ s[4] \neq \text{“}Y\text{”})$
$\quad \wedge (n1 = n - 1 \wedge\ s1 = \langle \text{“}B\text{”}, s[2] + 1, s[3] + 1, s[4] \rangle)$

$\beta(LeaveA) = (s[1] = \text{“}A\text{”} \wedge\ s1 = \langle \text{“}A\text{”}, s[2] - 1, s[3], s[4] \rangle)$

$\beta(LeaveB) = (s[1] = \text{“}B\text{”} \wedge\ s1 = \langle \text{“}B\text{”}, s[2] - 1, s[3], s[4] \rangle)$

$\beta(TimeoutA) = (s[1] = \text{“}A\text{”} \wedge ((na = 0 \wedge\ nb > 0) \vee\ s[3] = 2) \wedge\ s[4] \neq \text{“}Y\text{”})$
$\quad \wedge\ s1 = \langle \text{“}A\text{”}, s[2], s[3], \text{“}Y\text{”} \rangle)$

$\beta(TimeoutB) = (s[1] = \text{“}B\text{”} \wedge ((na = 0 \wedge\ nb > 0) \vee\ s[3] = 2) \wedge\ s[4] \neq \text{“}Y\text{”})$
$\quad \wedge\ s1 = \langle \text{“}B\text{”}, s[2], s[3], \text{“}Y\text{”} \rangle)$

$\beta(Switch) = (s[2] = 0 \wedge\ s[4] = \text{“}Y\text{”} \wedge\ s[1] = \text{“}A\text{”} \wedge\ s1 = \langle \text{“}B\text{”}, 0, 0, \text{“}N\text{”} \rangle)$
$\quad \vee (s[2] = 0 \wedge\ s[4] = \text{“}Y\text{”} \wedge\ s[1] = \text{“}B\text{”} \wedge\ s1 = \langle \text{“}A\text{”}, 0, 0, \text{“}N\text{”} \rangle)$

Table 2.4: Transition Constraints

The transitions $RegisterA$ and $RegisterB$ move the vehicles from the pool of available vehicles to the waiting phase. While doing so, these transitions increase the number of vehicles held by the places $NumberA$, $NumberB$. This is specified using the $\beta(RegisterA)$ and $\beta(RegisterB)$ in table 2.4. Similarly, the transitions $AllowA$ and $AllowB$ are used to move the vehicles from *waiting* to *onBridge* state and decrease the number of waiting for vehicles on the respective sides. Finally, the transitions $LeaveA$ and $LeaveB$ are used to move vehicles to *exited* phase. On the other hand, the transitions $TimeoutA$ and $TimeoutB$ are used to inform the controller that a switching may be needed. The need for switching is determined when one of the following conditions happens - (1) no vehicle is waiting on the currently active side, but the other side has some waiting vehicle; (2) the allowed

number of crossed vehicles (N) on the current side is reached.

The modeling approach mentioned here is just one way of modeling the system. There are several other ways of doing this. For example, instead of considering each vehicle's movement individually, the movement can be captured collectively by considering the number of vehicles on each phase.

2.5 New Features in PIPE+

2.5.1 New Type

There is no restriction in PrTN on the sort of data to use in the datatypes. But the realization of PrTN in the tool PIPE+ is limited to only strings and integers. Therefore, modeling using PIPE+ was limited to modeling systems with particular dynamics only. In reality, most of the systems, whether DES or not, deal with some attributes of some entities of the system having fractional numbers. For instance, consider that a modeler is interested in modeling a simple academic result calculator system for students. Here the modeler would like to keep track of the scores the students achieve on some courses. Then, eventually, the numerical and letter grades need to be computed. If a real-world problem is mapped with this hypothetical problem, then it would be clear to see that it is natural to model the attributes *score*, *numerical grade*, and *GPA* using fractions.

To resolve this limitation, a more generic term *number* is introduced, replacing *integer*. The sort *number* can be used to represent *real* numbers ($\mathcal{R}$). If a field is typed as `number`, it may hold a fractional number or an integer. From the modeling point of view, such distinction is redundant. It should suffice as long as the cor-

rect representation is achieved. However, realizing fractional numbers introduced technical challenges from the mathematical point of view. These challenges include,

- **Precision of Numerical Value**. When *arithmetic expression*s containing floating-point numbers are evaluated, the result's precision may differ depending on the runtime system. Uneven precision may produce an incorrect result or may lead to undesired circumstances. As a partial solution to this problem, the modeler can use the *round* function to round up the values directly in the transition constraints following the technique described in section 2.5.2.
- **Comparison of Numbers.** Arithmetic operations involving floating-point numbers is still inconsistent. The evaluation of two arithmetic expressions involving floating-point numbers may result in slightly different results whereas exact same values are expected. This may result in unexpected behavior when their equality is checked. A partial solution is provided in the simulation environment of PIPE+. Instead of comparing two numbers for equality, the difference is computed and if the difference is within a predefined tolerance value, the numbers are considered equal. Also, the modeler can utilize the `round` function before comparing for euality.
- **Suitable Arithmetic Operations.** The inclusion of *number* datatype urged on the addition of specialized mathematical operations suitable for fractional numbers. For example, exponentials, operations to work with precision - ceiling, floor, rounding, etc. To provide better utilization of *number*s, several new arithmetic operators and mathematical expressions are introduced as described in section 2.5.2.

Apart from these, there aroused another problem from the model checking perspective. Some model checkers, e.g., SPIN, do not support floating-point numbers.

There is no proper solution to this problem, but the floating-point numbers are rounded to the nearest integer at the time of translation.

2.5.2 New Mathematical Expressions

2.5.2.1 Function Constructs

To provide extended capabilities of modeling with *real* numbers, several arithmetic operations other than the basic ones need to be available. The main challenge in realizing a new operator in PIPE+ is that it is a complex task. Also, many operations can not be expressed as operators, e.g., *abs, min, max, transcendental functions*, etc. The concept of $function$ is introduced to resolve this. These $function$s are similar to the concept of *inline-functions* in popular programming languages. These are expected to produce one single value when evaluated. The resulted value can be used in conjunction with other mathematical operators. This approach provides a generic solution to the necessity of extending arithmetic capabilities without modifying the *grammar* that accepts those expressions and the *parser*s. This option also allows the modelers to include their functions as well. PIPE+ provides an easy mechanism to extend the function interpreter.

2.5.2.2 Tuples

In a PrTN model, tokens of places are generally structured data of one or more elements. An *equals (=)* relation is used in the specification of post-conditions to assign new values to these elements. During computation, one *equals* relation computes and assigns value to one element. Thus, the assignment of values to the elements of a token is distributed across multiple *equals* relational clauses. This is inconvenient, error-prone, and hard to debug, especially when the expression

is long and many logical branches are present in the expression. Instead of this approach, the concept of *tuple* is introduced. With this approach, assignments of new values to a token can be specified within the same clause. It is convenient. It also facilitates the detection of inconsistencies during modeling. Equations 2.1 and 2.2 show examples of these constructs. Here, x is an input variable, and y is an output variable.

$$((x[1] = \text{“}U\text{”} \wedge y[1] = x[1] \wedge y[2] = x[2]-1) \vee (x[1] = \text{“}D\text{”} \wedge y[1] = x[1] \wedge y[2] = x[2]+1)) \tag{2.1}$$

$$(x[1] = \text{“}U\text{”} \wedge y = \langle x[1], x[2]-1 \rangle) \vee (x[1] = \text{“}D\text{”} \wedge y = \langle x[1], x[2]+1 \rangle) \tag{2.2}$$

2.5.2.3 Quantifiers

The support for quantified formulas (both universal and existential) was provided in previous works on PIPE+ [13]. But the evaluation has several flaws. Also, the quantified formula involving multiple variables was not supported. The quantifier expressions are restructured to support multiple variables. However, quantifiers with multiple variables cannot be used directly; instead, an equivalent form should be specified. Table 2.5 shows the equivalent forms of quantifiers with two variables as supported in PIPE+. A similar construct can be employed for quantifiers with more variables. Also, quantifiers can be nested into multiple levels. PIPE+ does not set any restrictions on the depth of nesting. But various levels may increase the time complexity of the evaluation.

Quantifier	Original form	Equivalent form
Universal	$\forall x \in X, y \in Y \cdot (f(x,y))$	$\forall x \in X \cdot (\forall y \in Y \cdot (f(x,y)))$
Existential	$\exists x \in X, y \in Y \cdot (f(x,y))$	$\exists x \in X \cdot (\exists y \in Y \cdot (f(x,y)))$

Table 2.5: Equivalent forms of quantifiers with two variables

2.5.2.4 Set Operations

Four types of operations on sets are available in PIPE+, namely, (1) union, (2) intersection, (3) diff, and (4) makeset. Some of these functionalities were present in the previous version of PIPE+, but with limited capability. Previously the *makeset* operation was defined to create only a single element set. This behavior is quite inconvenient when a set with multiple elements need to be created dynamically. Also, it was not capable of filtering and transforming an existing set during execution. These shortcomings are resolved as part of the tool enhancement under this work. Apart from this, only static sets (specified by the multi-values set variables) could be used with the other set operations. These operations are re-designed so that they can operate on both static and dynamically created sets.

Operation	Example
Union	$Z = X \cup Y$
Intersection	$Z = X \cap Y$
Diff	$Z = X \setminus Y$
Makeset	$Y = \{y : Y \mid \forall x \in X \cdot (f(x) \wedge y[1] = g(x) \wedge y[2] = h(x) \wedge ...\}$
	$Y = \{\langle x[1], x[2], x[3] \rangle\}$

Table 2.6: Example of specifying set operations in PIPE+

Table 2.6 shows the example representations of the set operations. In the examples, the union, intersection, and diff operations are applied to set variables. These operations can be applied to the dynamic set created using `makeset` operation. Two different representations of makeset are shown. The second representation creates a one-element set. The element has three fields. In the first makeset example, filtering and transforming techniques are applied. Here, y is the user variable used to construct the elements of the output set Y. Here, $f(x)$ is filtering condition. The elements in X that satisfy $f(x)$ will be used to construct the elements of Y. $g(x)$

will be used to compute the first field of each element, and $h(x)$ will be used to compute the second field and so on.

2.5.3 Deterministic Choices

In most discrete control systems, generally, the system remains in only one control state at a certain point. These control states usually have a specific sequence. That is, a certain control state can be reached only from another certain state. Moreover, the systems usually behave differently in different states. Due to the concurrent behavior, it is hard to emulate this scenario in Petri nets. There are ways to achieve this, which are often inconvenient from the modeling perspective and computationally expensive. The concept of deterministic choice is introduced to address this problem. It is a technique of specifying the transition constraints in a specific way so that the effect of $if - elseif - else$ blocks can be achieved. For this, the constraint of a transition t should be of the form $\bigvee_i d_i$ for $i \geq 2$, where each d_i is a conjunction $pre_i \wedge post_i$. pre_i defines the enabling condition (precondition) and $post_i$ defines the processing result (post-condition). During execution, the first d_i with satisfying pre_i will be chosen for computing new marking, i.e., $post_i$ will be used to compute new marking other disjuncts will be ignored. Here each disjunct d_i may represent a distinct control state. With the appropriate reorganization of these disjuncts, the desired sequence of events and actions can be achieved.

2.5.4 Model Composition

Petri nets are a distributed state computation model where the marking defines the overall system state. This makes Petri nets naturally suitable for incremental system modeling. Incremental system modeling refers to the modeling paradigm in which

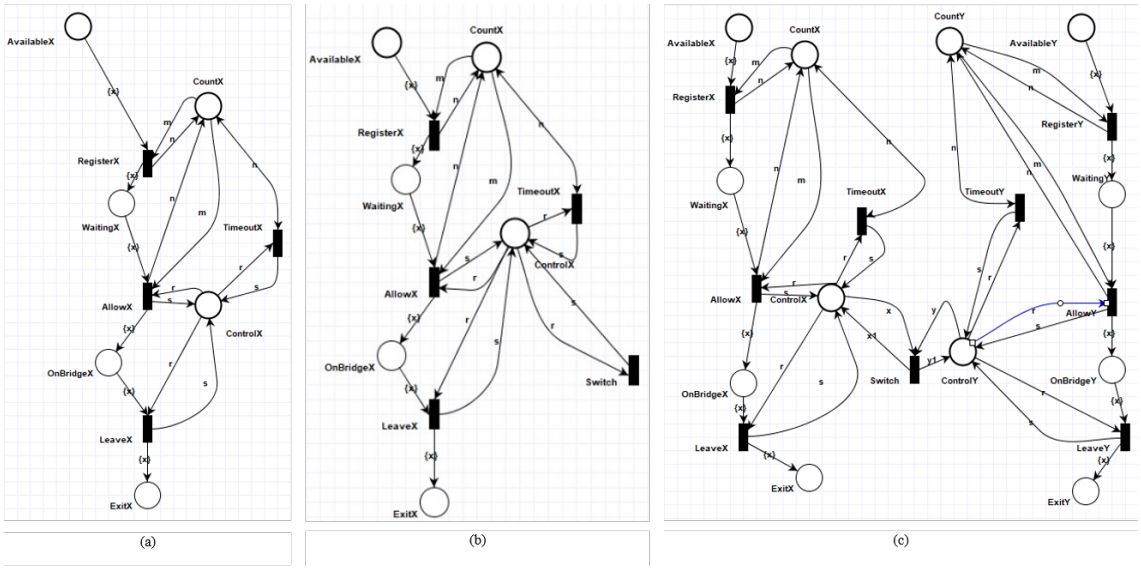

Figure 2.3: Incremental modeling of the Bridge system. (a) Model of a controller; (b) A controller is connected to a switch, and (c) Two controllers are connected to a switch

the bottom-up approach is used to model complex systems. In this approach, the whole system is broken down into smaller subsystems. These smaller subsystems are modeled separately. Then the model of the entire system is represented as a composition of those subsystem models. Incremental modeling can be of two types - (1) synchronous and (2) asynchronous. From the perspective of PrTNs, the asynchronous composition of the PrTN models N_1 and N_2 is obtained through some shared places, in other words merging a place in N_1 with another compatible one in N_2. The synchronous composition can be achieved by connecting a place in one net and a transition in the other, in other words, adding an arc from a place in one net to a transition in the other and vice verse. The synchronous composition can also be achieved by merging two discrete transitions from both nets. PIPE+ provides several strategies to facilitate both synchronous and asynchronous incremental modeling. Algorithm 1 shows a high-level overview of the algorithm used for net composition in PIPE+.

Algorithm 1: ComposeNet

Data: (N_1, N_2, s, d), where N_1 and N_2 are two nets, s and d are two nodes (place/transition) from N_1 and N_2 respectively

Result: N, the resulting net composed of N_1 and N_2

```
1  begin
2      N ⟵ create an empty net
3      add all elements of N1 to N
4      add all elements of N2 to N
5      if both s and d are places then
           /* Synchronous composition by merging two places        */
6          for all arc ∈ N2 do
7              if d is the source of arc then update source of arc with s ;
8              if d is the destination of arc then update destination of arc
                with s ;
9          end
10     end
11     if both s and d are not transitions then
           /* Asynchronous composition by adding an arc            */
12         arc ⟵ createNewArc()
13         set s as the source of arc
14         set d as the destination of arc
15         add arc to N
16     end
17 end
```

Figure 2.3 shows an example of the incremental modeling of the *Bridge and Vehicles* system. Figure 2.3.(a) shows the model of a controller that controls the traffic on one side. Figure 2.3.(c) shows that two of such controllers are connected to switch to complete the whole system model.

2.5.5 Logical Clock

The evolution of discrete systems depends on system events rather than time. It may seem logical to think that time information is entirely irrelevant here. But it is not completely true. There may be situations where certain events have a waiting

Algorithm 2: Pseudocode to initialize and update timestamp

```
function initializeTimestampOnStartSimulation()
Data: config, Timestamp configuration provided by the user
begin
    set system property "time.now" to config.begin
    set system property "time.prev" to config.begin
    set system property "time.step" to config.stepsize
end

function updateTimestampAfterFiring()
begin
    requireUpdate ⟵ determine whether update needed
    now ⟵ get system property "time.now"
    step ⟵ get system property "time.step
    set system property "time.prev" now
    if requireUpdate then
        updatedTime ⟵ now + step
        set system property "time.now" to updatedTIme
    end
end
```

period; that is, some events can not occur until a specific time. Also, the action may need a while to be finished. To model these scenarios, a representation of time and access to the current time from the transition constraints are necessary. Introducing exact time in Petri net to model timed dependent communication protocols resulted in timed Petri nets [27], where a time duration was associated with each transition. Other variations of timed and time Petri nets have been proposed since then, which either associate a duration with each place or associate a pair of bounds (lower and upper bounds) with each transition. These time and timed Petri nets can be adequately captured by high-level Petri nets [28] using time stamp carrying tokens and transition constraints for checking and updating token time stamps.

The modeling environment in PIPE+ provides a simple means to specify the time, which approximates a logical clock. This clock always starts from 0 when a simulation run begins and gives a non-decreasing interval of fixed size step. No

assumption on the step size is made. The modeler can assume any time unit for the step size. All the co-efficient of physical properties used need to be adjusted accordingly. For example, the equation (2.3) computes the velocity of a falling object. Here, a step size of 0.01 seconds is assumed, and the value of the gravitational coefficient is considered to be 0.098. The current time provided by the clock can be accessed using the operator τ. Algorithm 2 shows the algorithm used to initialize timestamp at the beginning of the simulation and the intermediate updates. The operator τ can be used in the transition constraints to model the dynamics of the attributes of the system. But, to impose additional constraint on the transitions' enabledness/disabledness the approach mentioned later in the previous paragraph needs to be utilized.

$$d1[1] = d[1] - 0.098 * \tau \tag{2.3}$$

Alternatively, if a more complex or precise timing is needed, the modeler needs to provide the strategy to define time. This can be done by adding a continuous place to the model [29]. The place would hold the logical clock value, and its evolution formula could be used to generate the time. In this case, it is the modeler's responsibility for the coherent evolution of the time.

2.6 Related Work

High-level Petri nets have been widely used for modeling discrete systems for several decades [30, 31]. Predicate transition nets [6, 25], colored Petri nets [7], algebraic Petri nets [8] are widely used classes of high-level Petri nets. The predicate transition nets presented in this chapter are closely related to the original predicate transition nets [6], and have a fully realized algebraic specification.

Various tools have been developed for high-level Petri nets. The best known tool is CPNTools [32] for colored Petri nets. PIPE+ is the state of the art tool for predicate transition nets.

2.7 Summary

This chapter gives an overview of Predicate Transition Nets (PrTNs). A formal definition of a class of PrTNs provided to define its structural and dynamic semantics. This gives a formal basis of realization of a version of PrTN in the tool PIPE+. A modeling methodology is also provided to model discrete event systems using PrTN is the tool PIPE+. Under this study, the modeling capability is increased in several ways by introducing new techniques and concepts.

First the application domain of PrTN is increased by introducing real numbers to represents system entities having numerical value. The addition of real numbers introduces several new challenges like precision mismatch, difficulty in comparison, etc. Some of these are highlighted and potential solution is provided. Also, modeling capability is enhanced by introducing several new arithmetic expressions, providing support for full first-order logic, incremental modeling via model compositions, providing a simple logical clock.

CHAPTER 3

ANALYZING DISCRETE EVENT SYSTEMS

PIPE+ supports several analysis techniques for discrete event systems modeled with PrTN. These techniques include simulation, model checking, and bounded model checking. The support for simulation is native to the tool. On the other hand, support for the model checking techniques leverages full-fledged external state of the art model checkers through model translation. PIPE+ provides a translator from PrTN to Z3 [33] supporting bounded model checking technique as presented in [34]. Translator to *Promela* to support model checking using SPIN [35] is initially presented in [36, 17] which is later refined in [18]. PIPE+ also provides a translator to Maude [37] to support a term-rewriting approach of model checking[38]. The translators to SPIN is fully automated, and the translated *Promela* code can be used in SPIN directly. However, other translated code for other model checkers needs a little customization.

Under this study, some of these techniques are thoroughly re-visited, and several significant improvements are made, especially the simulation environment and model translation for the SPIN model checker. The following sections will provide a detailed discussion of these improvements.

3.1 Simulation

3.1.1 Simulation Strategy

PIPE+ simulates a system model following the dynamic semantics of PrTNs. A simulation run consists of one or more execution steps. Each step involves in finding enabled transitions and then fire the transition(s) depending on the selected simu-

Algorithm 3: findAssignmentIfEnabled

```
Data: t, the transition to test enabledness
Result: θ, an assignment of values to the input variables of t
1  begin
2      P ⟵ find the set of input places of t
3      V ⟵ find the set of input variables of t
4      Θ ⟵ find the token combinations using P and V where each token
         combination is a valid assignment to each v ∈ V.
5      for θ ∈ Θ do
6          s ⟵ evaluate β(t) against θ ignoring postconditions
7          if s then
8              return θ
9          end
10     end
11     return null
12 end
```

lation modes as described in subsection 3.1.2. The constraints of a transition, the first-order logic formula associated with it, is evaluated against an assignment of tokens to its input variables from the available tokens from its input places. If there is no token available of some input place, then it is determined to be disabled without evaluating the formula. If some of the input places are *powersets* having multiple tokens, there are more than one potential assignments available for evaluation. The adopted strategy selects one of such assignments and evaluates the formula to test whether the preconditions are satisfied. If the current assignment is not a satisfying assignment and other assignments are available, the process continues to follow the steps until one satisfying assignment is found. If no satisfying assignment is available, the transition is marked as *disabled*. Otherwise, it is marked as enabled, and the satisfying assignment is used to evaluate the postcondition. The postcondition of a transition is computed if it is selected for firing. Algorithm 3 provides an overview of the strategy implemented in PIPE+ to determine the enabledness of a transition. Algorithm 4 shows the steps performed when a transition is fired.

Algorithm 4: fireTransition

```
Data: t, the transition to test enabledness
      θ, an assignment of values to the input variables of t
1  begin
2      V_i ⟵ find the set of input variables of t
3      V_o ⟵ find the set of output variables of t
4      for v ∈ V_o do
5          initialize v with default values
6      end
7      θ ⟵ θ ∪ V_o
8      evaluate β(t) against θ
9      for v ∈ V_i do
10         P ⟵ find the set of places associated with v
11         for p ∈ P do
12             remove the token held by v from M(p)
13         end
14     end
15     for v ∈ V_o do
16         P ⟵ find the set of places associated with v
17         for p ∈ P do
18             add the token held by v to M(p)
19         end
20     end
21 end
```

The user can run simulation one step at a time or for a fixed number of steps or indefinitely as long as it is possible to run. In this case, the simulation keeps running until there is no enabled transition.

3.1.2 Simulation Modes

In an execution step during simulation, there may be conflicts among transitions. Conflicts among transitions arise when there are multiple enabled transitions with shared input or output places or both. It is not always safe to fire all of these enabled transitions. Because firing of some transitions may disable other transitions with shared input places. On the other hand, enabled transitions with shared output

places may result in inconsistent marking, i.e., inconsistent system state. Thus the execution mechanism needs to synchronize the firing of these transitions. PIPE+ provides several modes of simulation to resolve this. The user selects the mode for a simulation run. There are three different built-in simulation modes - (1) interactive, (2) interleaving, and (3) concurrent. These modes are described below.

3.1.2.1 Interactive Mode

In this mode, the user selects which transition to fire in case of more than one enabled transition. Currently, the granularity level of interaction is the transition level, not the token level. That is, the user can select only a transition to fire. The user cannot choose a particular token combination (variable assignment) for the selected transition.

3.1.2.2 Interleaving Mode

In this mode, one of the enabled transitions is randomly selected for firing. For efficiency and performance, PIPE+ does not choose a random transition from the set of enabled transitions. Rather, it first picks an available (not disabled) transition randomly and then tests its enabledness. Interleaving mode is the default simulation mode in PIPE+. Algorithm5 shows the algorithm of the part of the scheduler that runs the execution step is interleaving mode.

3.1.2.3 Concurrent Mode

In this mode, multiple enabled non-conflicting transitions are fired in the same execution step. For this, sets of non-conflicting transitions are maintained. All the transitions in each such set are non-conflicting. This means that no pair of transitions in a set have shared input or output places. During execution, one such

Algorithm 5: executeInterleaved

Result: (t, θ), a pair consisting of the fired transition t and the assignment θ

```
begin
    D ⟵ the set of disabled transitions
    A ⟵ T \ D
    while A is not empty do
        a ⟵ remove a transition from A randomly
        θ ⟵ findAssignmentIfEnabled(a)
        if θ is null then
            add a to D
        end
        else
            fireTransition(a, θ)
            Q ⟵ find the set of dependent transitions of a
               /* equation(3.2) describes the dependent set      */
            remove Q from D
            return (a, θ)
        end
    end
    return (null, null)
end
```

is chosen randomly, and all the transitions in that set are evaluated and fired if enabled. If none of the transitions in the selected set is enabled, another set (if any) is chosen randomly. The simulation ends when there is no enabled transition.

3.1.3 Simulation Results

PIPE+ keeps track and records some information related to each execution steps performed in a simulation run. For each step, PIPE+ stores the name of the fired transitions, the timestamp. For each transition, the assignment table, i.e., the initialized list of variables are stored. The result viewer for DES shows these in a list view. These results can also be store on Filesystem for later analysis. PIPE+ stores this information in the configured file in JSON format. This information can be

used to replay the simulation or for rewinding. The stored information can help to debug an execution.

3.2 Model Checking

Model Checking is a technique to verify finite-state systems formally. This technique involves systematically checking *finite–state model*s of a system against the *specification* of the system to test whether the model satisfies the specification. Generally, the models are the mathematical representation of the system behaviors, and the specifications consist of a set of invariants (safety properties) and liveness requirements of the systems. A typical process of model checking involves generating the system's states following the mathematical representation of the system and checking whether the states violate some conditions of the specification. If a violation is found, then the system's model is deemed unsafe and requires a redesign. Usually, this process is carried on until a safe design is established.

There are numerous model checking techniques and tools are available. The model checker SPIN [35] is a very powerful and effective model checker and widely used in the industry. The focus of this study is to leverage the model checking power of SPIN to model check PrTN models. A PrTN model is first transformed into a model in Promela [39]. Promela is the underlying modeling language of SPIN. Then the generated Promela model can be used with SPIN to perform the verification. The following subsections will provide an overview of the essential Promela constructs related to the PrTN, an in-depth discussion on translation strategies for different components of PrTN, an informal proof of correctness of the translation, and a literature review on the recent development along this line and the techniques developed for the translation process.

3.2.1 SPIN Model Checker

SPIN [35] is a very efficient verification tool for distributed concurrent systems. This tool provides powerful techniques to check logical consistency of a specification, deadlocks, race conditions, incompleteness, and unwarranted assumptions. It has been used to verify logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, control software for spacecraft, nuclear power plants, and many others [40].

SPIN uses a high-level language, Promela, to model the behaviors of the system. The correctness properties can be specified using Linear Temporal Logic (LTL)[41] formula, using Buchi Automata or as SPIN *never* claims as part of the system model. SPIN uses efficient partial order reduction techniques for optimized verification runs. This tool supports random, interactive and guided simulation, and both exhaustive and partial proof techniques, based on depth-first search, breadth-first search, or bounded context-switching [40].

3.2.2 Promela

Promela is the underlying modeling language for SPIN to describe a system model. An operational model in Promela contains one or more processes, zero or more variables, zero or more message channels, and a semantics engine [35].

Processes are the central construct in a Promela model to define system behaviors and are defined using *proctype* declaration.

Message *channel*s are used to model the transfer of data from one state to another, which can store a finite number of messages as declared. Apart from storing data, channels provide a wide range of features to model message passing cleanly

Listing 3.1: Basic usage of message channel

```
1      chan qname = [8] of {int,short}
2      chan qname = [8] of {s_type}
3      qname!10, 20
4      qname?x, y
5      qname?x, eval(y)
6      qname?[x, eval(y)]
7      qname??[x, eval(y)]
8      len(qname)
```

and efficiently. They are FIFO queues but can also be used for random accesses. When a new message is sent to a channel, it is added to the end. When an attempt is made to retrieve a message, it always returns a message in front. It is also possible to query a channel for a specific message. By default, the channel removes messages as they are retrieved. However, it is also possible to get a message without removing it. Listing 3.1 lists some basic channel usages.

In the above example, lines 1 and 2 show the declaration of channels with basic data types and structured data types, respectively. Line 3 sends a message to the channel. Lines 4-7 show different ways to retrieve messages. Lines 4, 5, and 6 retrieve the first message. Lines 4 and 5 also remove the message after retrieving. Lines 5 and 6 checks whether the second element of the first message matches the value currently held in y. Line 7 searches for a match anywhere in the channel. Lines 6 and 7 retrieve a message without removing it (poll operation). Line 8 returns the number of messages in the channel.

Promela provides non-determinism by default. The *case selection* and *looping* shown in Table 3.1 provide non-determinism. Both constructs have a similar structure. Each case in the case-selection construct is marked with a guarded statement (started with ::). If there is more than one case, then the sequence of statements will be selected for which the guarded statement is executable. If there is more than

one such statement, then one of those will be chosen non-deterministically. If no such statement present, then the block is exited. A loop tries to find one executable guarded statement each time it completes execution. Promela also has a for loop construct, which is only to be used to iterate through channels.

Case selection	Looping
if :: case 1 :: case 2 :: case 3 fi	do :: case 1 :: case 2 :: case 3 od

Table 3.1: Examples of control constructs

The next interesting construct is *inline* functions. Although Promela does not have a general function concept but supports inline functions as macros. The inline functions are used to sorganize the translated code.

3.2.3 Translation of PrTNs to Promela

In this section, the techniques to convert PrTN models into Promela models are presented. From a high-level overview, the conversion procedure is done using three steps. In the first step, the net structure is transformed and modeled using the Promela formalism. In the second step, the data, in other words, the marking of the net, are transformed into Promela syntax. Finally, the dynamic behavior of PrTN is integrated by reproducing similar semantics using Promela constructs. In the following sub-sections, these are discusses in detail.

Some of these techniques were introduced in [42, 36]. Later these techniques were improved in [17, 18] under this study. The new translation method eliminates many limitations found there and supports many advanced features such as complex data structures and first-order logic formulas. The enabledness testing and firing of

transitions are combined with an atomic block to improve model checking accuracy. Furthermore, an additional process-based translation scheme is implemented, which proved to be more effective in checking some safety and liveness properties. A comprehensive discussion of these translation rules, as well as identifying problems, are provided in the following sections. Here, the bridge systems model (section 2.4) is used as a running example. The translation rules are illustrated by presenting the snippets of the translation of the bridge system.

3.2.3.1 Translation of Places

In PrTNs, places capture the state of the net by storing tokens. Furthermore, PrTNs are a data flow computation model where state changes occur through token movements. During net execution, tokens from some places may be moved to some other, some tokens may be destroyed, and even new tokens may be generated and stored in some places. The basic operations include looking for specific tokens, adding new tokens, and removing some tokens and finding the number of tokens. Places have another constraint on the maximum number of tokens they can store. Some places can have more than one token and are termed as powerset places. Each of these powerset places may have an upper limit of the number of tokens, termed as capacity. These two concepts must be taken care of to represent the places of PrTN models correctly in Promela.

The initial idea to achieve the above could be to define arrays of the structured data type to store data, implementing the operations as inline functions and macros. Instead, *channel*s are chosen to model *place*s. Channels provide many benefits over the initial idea. First of all, the built-in functions of channel make it easy to query specific tokens for checking transition enabling conditions, and adding/removing tokens for transition firing, checking the number of tokens. Secondly, the channel

makes it very easy to specify the properties to verify using the LTL formulas. Each place in a PrTN is thus translated into a channel with the same type of the place and the tokens in the place are translated into messages stored in that channel in the initialization. Thus, $\forall\ p \in\ P$, the following two lines are used as declarations.

```
1 #define bound_p const
2 chan place_p = [bound_p] of {type_p}
```

Here, *type_p* is a data type in Promela resulted from the translation of the data type associated to place p, which is discussed in the translation of data types, and *bound_p* denotes the capacity.

The basic operations on places like adding/removing tokens are translated as the sending/receiving messages to these channels. Searching for a specific token is translated as poll operation to channels. Consider the following code snippet –

```
1 Type_p x;
2 x.field1 = Wa;
3 place_WaitA??eval(x.field1);
4 place_OnBridgeA!x;
```

Here, line 3 tests whether the vehicle Wa is in the place $WaitA$ or not, if so then it is removed from place $WaitA$. Line 4 transfers the vehicle to the place $OnBridgeA$. Now consider the following property specification in the LTL format.

```
1 []!( nempty(place_onBridgeA) && nempty(place_onBridgeB))
```

The keyword *nempty* returns *true* if the input channel is not empty. Therefore, the above statement specifies the property that vehicles from both sides can never be on the bridge at the same time. It is a very convenient way, as well. No other data structure would provide this flexibility.

Thus, it can be concluded that the built-in support around the channel concept in Promela makes it an ideal choice to represent places. Nonetheless, for some cases in some models, the tokens of some places may be better represented as separate processes. However, these are extreme cases, and this option is yet to be explored. Also, sometimes places can be represented as a simple variable, especially if the place is not *powerset*, and it has only one element.

3.2.3.2 Translation of Transitions

Transitions are the core components of the dynamic semantics of a PrTN model and play an essential role in the execution of the model. The PrTN specification does not limit the number of transitions to be fired in one step to support true concurrency. Still, it is adequate to consider interleaved executions by firing only one transition to analyze state-based properties such as safety and liveness properties. As a result, it is suffice to focus on how to translate each transition firing correctly in Promela.

The execution of a transition has two parts – testing its enabling condition and firing it (computing new marking). In a PrTN, each transition is associated with a constraint specified in a first-order logic formula containing two parts – preconditions and postconditions. Preconditions consist of inputs to the transition and constitute its enabling condition. Generally, postconditions consist of the output of the transition. Usually, the outcome of the computation of the postconditions is the generation of new tokens that are used in conjunction with the input tokens to compute the new marking of the net when the transition is fired. The enabling

and firing of a transition are modeled using separate inline functions following the algorithm shown in Listing 3.2. These two functions work together as an atomic execution unit. For each transition $t \in T$, these inline functions are generated. The techniques adopted for evaluating precondition and postcondition are discussed in the subsection to translate transition constraints.

Listing 3.2: A general algorithm to translate a transition

```
1  inline t() {
2  for all combinations of values the input variables to t can
3      take, do
4    fire_t();
5    return;
6  done
7  }
8
9  inline fire_t() {
10   compute postconditions;
11   compute new marking;
12 }
```

In [42, 36], similar approach was adopted. However, that implementation suffered from serious performance and correctness issues. First of all, those two functions worked separately without working as an atomic unit. This was interpreted falsely by SPIN runtime during verification. Secondly, only one combination of inputs was checked at a time. When there is more than one input powerset places, it may result in unpredictable result. Another flaw was that it failed to produce a correct result when multiple places constitute the same input, i.e., multiple places are connected to the same transition using the same arc label. In the new translation scheme, these issues have been eliminated. Listing 3.3 is an illustration of the implementation of the algorithm for the transition *AllowB* in the example presented above.

Listing 3.3: An illustration of the translation of AllowB

```
1     inline AllowB() {
2       Type_3 n, n1;
3       Type_1 x;
4       Type_2 s, s1;
5       if
6         :: place_WaitB?[x] -> place_WaitB?<x>;
7         :: else -> goto checked_AllowB;
8       fi
9       for (s in place_Controller) {
10        for (n in place_NumberB) {
11          if
12            :: (n.field1>0 && s.field1==B && s.field2<10
13            && s.field3 < 10 && s.field4 != Y) ->
14            fire_AllowB_0();
15            goto endOf_AllowB;
16            :: else -> skip;
17          fi
18        }
19      }
20      checked_AllowB:
21      endOf_AllowB:
22    }
23
24    inline fire_AllowB_0() {
25      n1.field1 = n.field1 - 1;
26      s1.field1 = B;
27      s1.field2 = s.field2 + 1;
28      s1.field3 = s.field3 + 1;
29      s1.field4 = s.field4;
30
31      place_Controller?? eval(s.field1),eval(s.field2),
32                         eval(s.field3),eval(s.field4);
33      place_WaitB??eval(x.field1);
34      place_NumberB??eval(n.field1);
35      place_NumberB!n1;
36      place_OnBridgeB!x;
37      place_Controller!s1;
38    }
```

3.2.3.3 Translation of Arcs

Arcs help to specify the preconditions and postconditions during modeling PrTNs and determito determine those during translation. The arcs incoming to a transition t are termed as input arcs, and the places they connect are termed as input places to t. Similarly, the outgoing arcs from t are output arcs, and the places they connect are output places to t. In PrTNs, arc labels are known as the variables to t. Labels of input arcs are input variables, and labels of output arcs are output variables. Input variables constitute the preconditions. The clauses in the transition constraints of t contain only input variables that are part of the preconditions. During net execution, these input variables are initialized with some values from the input places; then, the preconditions are evaluated to test whether the input places have a satisfying assignment to the input variables.

Output variables are allowed only on the left-hand side of the assignment operator (=). Thus, the clauses of the transition constraints that have output variables are part of the postconditions, and computing postcondition means the assignment of computed values to the output variables. These computed values are used to compute the new marking of the net.

Although arcs are not the essential part of the translated model, they are used during the translation process to identify the preconditions and postconditions to the transitions.

3.2.3.4 Translation of Data Definition

The data definition of places are multisets over the set of sorts. These multisets are translated as structured types in Promela, where each sort in the data type definition is represented as a field in the structure.

Listing 3.4: An example of translation of data type definition

```
1     typedef struct type_p {
2          mtype field1;
3          short field2;
4          mtype field3;
5     }
```

PIPE+ supports only two sorts – string and number. These are used to define more complex data types for places through the Cartesian product. To translate these complex data types, first it is needed to determine the corresponding representations of the sorts.

Promela does not support *strings*. Thus a symbolic mapping representation is needed for strings in terms of available datatypes. Available options for this could be *int*, *byte*, *short* to represent *string*s. Since string type is introduced in a PrTN model to capture unique constant such as an identifier that does not change, mtype is selected to represent strings such that each string constant in the net are represented as a constant in mtype. Such a representation significantly reduces the number of states to be checked.

Another important consideration is to choose an appropriate representation of the sort number. Promela does not support real numbers but provides three types to support integers: int, short, and byte, in which the latter two help to reduce the state space. The *short* is used as the default representation and let a modeler select other choices during translation.

The data type definition of a place $p \in P$ is a tuple of these sorts. Each of these datatype definitions is modeled as structured data types in Promela, where each element in that tuple is translated as a field in the structured type. For example, a place p with data type, $\alpha(p) = \langle String, Number, String \rangle$ is translated into a Promela structured data type shown in Listing 3.4.

In the previous example, the tokens accepted by p are all tuples of three elements of type string, number, and string, respectively. In PIPE+, these tuples are implemented using Lists, and each element is accessed by their respective index. For example, if x is any token in p then, the first, second, and the third element are accessed by the expression $x[1]$, $x[2]$, and $x[3]$ respectively. The similar phenomenon is achieved in the translated Promela model as $x.field1$, $x.field2$ and $x.field3$. Table 3.2 shows an illustration of these mapping.

PrTN	Promela
	Type_p x;
x[1] =“MOVE LEFT”	x.field1 = MOVE_LEFT;
x[2] = 100	x.field2 = 100;
x[3] = “METER”	x.field3 = METER;

Table 3.2: The mapping of sorts and data type definitions

3.2.3.5 Translation of Transition Constraints

The constraint of a transition $t \in T$ is defined using a first-order logic formula. This formula specifies the relationships among input and output variables of t using certain algebraic, relational logical, and set operations. The translation process can identify the preconditions and postconditions from these formulas and these two parts are modeled following different strategies. In the following sub-sections, the available mathematical operations and their corresponding translation rules, and the method adopted to translate preconditions and postconditions are discussed.

Another major improvement in recently improved PIPE+ is the short-circuit evaluation of the transition constraints, which is extremely useful when one wants to simplify the net structure by combining multiple transitions performing similar tasks. This also helps minimize the size of the net and, in turn, fewer states to consider during model checking. In the example presented in section 2.4, the constraint

Listing 3.5: Example of conditional branching

```
1  inline fire_Switch_0() {
2      . . .
3  }
4
5  inline fire_Switch_1() {
6      . . .
7  }
8
9  inline Switch() {
10   . . .
11   for (s in place_Controller) {
12     if
13     ::(s.field1==B && s.field2==0&&s.field4==Y) ->
14             fire_Switch_0();
15     ::(s.field1==A && s.field2==0&&s.field4==Y) ->
16             fire_Switch_1();
17     :: else -> skip;
18     fi
19   }
20   . . .
21 }
```

for the transition Switch is specified in such a way to take advantage of this concept. The constraint here is mentioned in the Disjunctive-Normal-Form, i.e., two clauses are joined using a disjunction ($\vee$) operator. Each clause is a conjunction ($\wedge$) expression. Note that each clause contains both preconditions and postconditions. Also, both clauses generate similar tokens with slightly different values based on different preconditions. When a transition constraint exhibits these two characteristics, each clause is considered as a separate conditional branch. An example of the translation of the conditional branching is illustrated in Listing 3.5.

3.2.3.6 Translation of Operators

PIPE+ supports a variety of algebraic, relational, logical, and set operators. A PrTN model uses a subset of these operators to define their behavior. Table 3.3

shows the supported operators in PIPE+ and their equivalent representations in Promela. Some of the operators do not have direct representations. For those equivalent representations are provided. Some operators are overloaded while some others excluded from translation. These are discussed here.

Category	Operations	Operators	
		PIPE+	Promela
Connective (Logical)	And	$\wedge$	&&
	Or	$\vee$	||
	Not	$\neg$	!
	Implication	$\rightarrow$	
	Equivalence	$\leftrightarrow$	
Relational	Equals	$=$	==
	Not Equals	$\neq$	!=
	Greater	$>$	$>$
	Less	$<$	$<$
	Greater or Equal	$\geq$	>=
	Less or Equal	$\leq$	<=
Algebraic (number)	Addition	$+$	$+$
	Subtraction	$-$	$-$
	Multiplication	$*$	$*$
	Division	$/$	$/$
	Remainder	%	%
Predicate Logic	For All	$\forall$	
	Exists	$\exists$	
	Dot	$\cdot$	

Table 3.3: Operators in PIPE+ and their correspondence in Promela

The equals operator, =, is overloaded in PIPE+. When it is used in a precondition, it is a comparison. On the other hand, when used in a postcondition, it is treated as an assignment. In the translation of precondition, this operator is represented using the logical equals operator, ==, as mentioned in Table 3.3.

In Promela, the implication has a separate semantic, which is not aligned with the usage of implication in the first-order logic formula. Thus, it cannot be used directly; instead, its equivalent logical expression is used. For example, suppose

A and B are two Boolean expressions in PIPE+, then the expression $(A \rightarrow B)$ is represented as $(!A' \mid\mid B')$ where A' and B' are equivalent representation of A and B in Promela.

Similarly, the equivalence operator $(\leftrightarrow)$ is also not available in Promela. Thus, to represent this, the Boolean expression having this operator is transformed into its equivalent logical form. For example, the Boolean expression $(A \leftrightarrow B)$ is represented as $((A' \;\&\&\; B') \mid\mid (!A' \;\&\&\; !B'))$ where A' and B' are equivalent representation of A and B in Promela.

The set operators and quantifiers in Table 3.3 also do not have equivalent representations. The translation of quantifiers is given later. The translation of set operations is partially completed and is not further discussed in this paper since they are rarely used.

3.2.3.7 Translation of Preconditions

The translation rules described so far to represent data type definitions and operators are applied to convert Boolean expressions consisting the preconditions into equivalent form accepted in Promela. Consider the Boolean expression $((x[1] \geq 3.5 \wedge y[1] \neq 10) \rightarrow z[1] = 2 * x[1] + 1)$ is translated to the equivalent expression in Promela is shown in the following listing.

```
1 !(x.field1>=3&&y.field1!=10)||z.field1==(2*x.field1+1)
```

3.2.3.8 Translation of Postconditions

Translation of postconditions involve evaluation of the expressions and assignments of the values to the designated output variables. Alike preconditions, the evaluation

Listing 3.6: Illustration of computing postconditions

```
1       s1.field1 = A;
2       s1.field2 = s.field2+1;
3       s1.field3 = s.field3+1;
4       s1.field4 = s.field4;
```

is almost the same. The only difference is that the evaluated values are either constant (numeral or string) or numbers. The evaluated values are then assigned to the designated output variables. These output variables are used as messages to corresponding output channels during new marking generation. For example, Listing 3.6 shows the transformation of the postcondition, $(s1 = \langle \text{“}A\text{”}, s[2]{+}1, s[3]{+}1, s[4] \rangle)$, of the transition AllowA.

3.2.3.9 Translation of Quantifiers

The translation rules discussed so far assume that the inputs are all simple, consisting of one token only. However, sometimes it is necessary to work on a set of tokens. Checking that all the tokens of a place meet certain criteria or some of the tokens meet that criterion or even none of the tokens exhibits a property, maybe some users. To be more specific, suppose, to design a system that takes a fixed number of jobs to process. It then sends a notification when it finishes processing all the jobs at hand. To achieve this behavior, one may define a place, p, to store all the jobs with their status. Initially all "Unfinished". As the jobs finished processing their status are updated (possibly by some transitions). Then a transition, tn, can be defined to check whether the status of all the jobs are *"Finished"* or not. If all the jobs are marked as *"Finished"*, it is executed to send the notification. It is easy to define transition constraint $R(tn)$ as follows

```
1      R(tn) = (∀x∈X.(x[2]="finished") ∧ z="done")
```

Here, X is the set of tokens representing the jobs, and z is the output variable of tn. It is assumed that the tokens in X are tuples having two elements. The first element is $jobId$, and the second element is $status$.

PIPE+ supports two types of quantifiers – universal ($\forall$) and existential ($\exists$). Quantifiers are used mostly as a part of preconditions. A universal quantifier is used to test whether all the tokens exhibit the property. In contrast, the existential quantifier is used to test whether at least one token has this property. A first-order logic formula is used with quantifiers to specify that property which is called the quantified formula. A quantified formula often (1) has its own set of local variables, (2) may contain global variables (arc labels), and (3) may be nested. Based on these, the following steps are adopted to translate quantifiers – (1) introduce a Boolean variable to store the evaluation result of the quantified formula, (2) introduce an inline function implementing algorithm shown in Listings 3.7 to evaluate the quantified formula, (3) modify the algorithm shown in Listing 3.2 according to Listing 3.7 and (4) repeat these steps for all quantifiers including inner parts in case of nested quantifiers. While repeating these steps, unique names for these variables and inline function names are ensured. The algorithms for universal and existential quantifiers are a little bit different. This is only for illustration purposes. During the translation process, only one of these is created.

3.2.3.10 Translation of Data

Steps to translate data include – representing tokens in a consistent form of messages in Promela; and storing and retrieving those messages to/from the channels. In section 3.2.3.4, some idea of the transformation of PrTN tokens to Promela mes-

Listing 3.7: Algorithm to evaluate quantified formula

```
1  bool q_x = false;
2  inline check_universal_quantifier_x {
3    q_x = true;
4    For all token in the Set do the following
5      call nested quantifiers if any
6      q_x = evaluate the quantified formula
7      if q_x is false return;
8  }
9
10 inline check_existential_quantifier_x {
11   q_x = false;
12   For all token in the Set do the following
13     call nested quantifiers if any
14     q_x := evaluate the quantified formula
15     if q_x is true return;
16 }
```

sages is provided. In this section, the ways to send messages to channels is shown. Basically, there are two ways to send messages to Promela channels – extended and compact forms. The following table shows an example of these two forms. Consider the examples given in section 3.2.3.4,

Extended Form	Type_p x;
	x.field1 = MOVE_LEFT;
	x.field2 = 100;
	x.field3 = METER;
	place_p ! x

Compact Form	place_p ! MOVE_LEFT, 100, METER

Table 3.4: Formats of generating postconditions

The compact form is used in places when all the elements of a message are known, i.e., while translating the initial marking. The other form is used during the computation of postcondition and generating new marking.

Listing 3.8: Passive non-determinism based translation

```
1  proctype Main() {
2    do
3    :: is_t1_not_disabled -> t1();
4    :: is_t2_not_disabled -> t2();
5    . . .
6    :: is_tn_not_disabled -> tn();
7    od
8  }
9
10 init {
11   //initial marking goes here
12   run Main();
13 }
```

3.2.3.11 Translation of Dynamic Behavior

So far, the techniques of translating each component of PrTNs into Promela's constructs are discussed. To complete the translation, an overall execution structure based on the dynamic semantics of PrTNs is needed. There are two essential ways to select a transition firing in Promela – (1) as a passive inline function to be non-deterministically selected in a Do loop within a centralized process. As long as there is an enabled transition, the loop continues, or (2) as an active process selected for execution by the SPIN runtime if the enabling condition is true. Apart from these, another strategy – a combination of these two, agent-oriented translation is also implemented. In the following subsections these are discussed.

3.2.3.12 Passive Translation

Listing 3.8 shows a self-describing example of this approach. A similar approach was used in our prior works, where enabledness checking, and firing of a transition are executed separately, which is semantically incorrect. This problem is resolved in the new implementation by combining transition enabling checking with firing.

Here the $is\_ * \_not\_enabled$ are control variables to provide a way to mark a transition as disabled and a controlled way to reach the end state for approach 1. In our prior work, this was not considered. As a result, the simulation using the translated Promela code was a never-ending process. It also suffered from severe performance issues like the repeated selection of disabled transitions slowing down execution, sometimes preventing the progress of execution, or even often resulting in a non-progress cycle. The introduction of these control variables works as follows. Initially, these variables are set to true. When the precondition of a transition is not satisfied, its corresponding control variable is set to false. Once all these control variables become false, none of the do loop options is executable, and the execution exits to the end state.

However, this makes it vulnerable to the premature end of the execution. To prevent this, it is needed to reset some of these control variables to true again periodically. This is done when a transition is fired. The control variables of the transitions dependent to the fired transition are reset to true. This is correct because when a transition is fired, new marking is generated, which in turn may make some previously disabled transitions enabled again. But fortunately, the transitions that may be enabled again can be predicted from the PrTN structure. This set of transitions is called the dependency set. Suppose P_{t_o} and P_{t_i} are the sets of output and input places of transition t. Also, suppose that T_{p_i} is the set of transitions for which p is an input place. Then the set of dependent transitions of t, $dep(t)$ is computed using the equations 3.1 and 3.2.

$$P_t = P_{t_o} \cup P_{t_i} \tag{3.1}$$

$$\text{dep}(t) = \cup_{p \in P_t} T_{p_i} \tag{3.2}$$

Listing 3.9: Active process based translation

```
1  active proctype process_t1() {
2    do
3    :: should_continue -> t1();
4    :: else -> break;
5    od
6  }
7
8  Active proctype process_t2() {
9    do
10   :: should_continue -> t2();
11   :: else -> break;
12   od
13 }
```

3.2.3.13 Active Process based Translation

In this strategy, instead of using a centralized process, different active processes are used for each transition. In most cases, this strategy is slower than the previous one but results in much better model checking results in detecting violations of some safety properties and is important to check liveness properties when weak fairness assumptions are needed. This scheme is illustrated in the Listing 3.9. In this translation scheme, one single control variable, *should_continue*, is used. This is set to true when at least one of transitions' control variable ($is\_ * \_not\_disabled$) is true. When none of the transitions is enabled all the processes exit to end state.

3.2.3.14 Agent Oriented Translation

A third translation scheme is provided that supports agent-oriented incremental system modeling [15], where a system model is created by weaving agent nets and supported by SAMAT tool [14]. The idea is to group a set of transitions and create a process for each agent net. The implementation of the process follows

Listing 3.10: Agent based translation of the bridge system

```
1  active proctype process_ControllerX() {
2    do
3    :: is_registerX_not_disabled -> registerX();
4    :: is_allowX_not_disabled -> allowX();
5    :: is_leaveX_not_disabled -> leaveX();
6    :: is_timeoutX_not_disabled -> timeoutX();
7    :: should_continue -> skip;
8    ::!should_continue break
9    od
10 }
11
12 active proctype process_Switch() {
13   do
14   :: is_switch_not_disabled -> switch();
15   :: should_continue -> skip;
16   ::!should_continue -> break
17   od
18 }
19
20 active proctype process_ControllerY() {
21   do
22   :: is_registerY_not_disabled -> registerY();
23   :: is_allowY_not_disabled -> allowY();
24   :: is_leaveY_not_disabled -> leaveY();
25   :: is_timeoutY_not_disabled -> timeoutY();
26   :: should_continue -> skip;
27   ::!should_continue -> break
28   od
29 }
```

the technique presented in section 3.2.3.12. For example, consider the bridge system described earlier. It is decomposed into two different agents – controller and switch. The controller controls the flow of traffic from one side only, and the switch switches among sides. Two controllers representing traffic from each side and the switch are integrated to form the whole system, shown in Figure. 3.1 (c). In this translation scheme, one process is created for each agent. The processes are non-deterministically selected to execute. This translation skeleton for this example is shown in Listing 3.10.

3.2.4 Translation Correctness

The correctness of the translation method covers completeness and consistency. Completeness measures whether all features of PrTNs are translated into Promela. Consistency refers to the equivalence between a PrTN model's dynamic behavior and the dynamic behavior of the translated Promela program. The concurrency transition firings in PrTNs that do not affect the satisfiability of safety and liveness properties that are necessarily state-based. Therefore, only the interleaved executions between a PrTN and the translated Promela program need to be considered. Each interleaved execution starts from the initial marking and continues by firing one enabled transition at a time. Using an induction proof principle, it can be only shown that (1) the initial marking is translated correctly, (2) each transition is translated correctly, i.e. the enabling condition and firing result are translated correctly, and (3) each enabled transition will be selected to fire. Step (1) is trivially true in our translation method. Step (2) is arguably true based on our careful design and extensive testing. Step (3) is true for both our overall model execution strategies discussed in the previous section. However, the formal proof of a general translation method is not easy, which is why few compilers have been formally verified.

3.2.5 Experiment Results

Several benchmark systems from the annual Petri net model checking contest 2015 (MCC'2015) [26], including **BridgesAndVehicles** (section 2.4) from MCC'2015, have been modeled and analyzed to evaluate the translation methods. Only the high-level Petri net models available in the contest are considered while carrying out the experiments. These models were defined using colored Petri nets (a type of high-level Petri nets). These systems are redefined using PrTNs in PIPE+. Only the *Bridges*

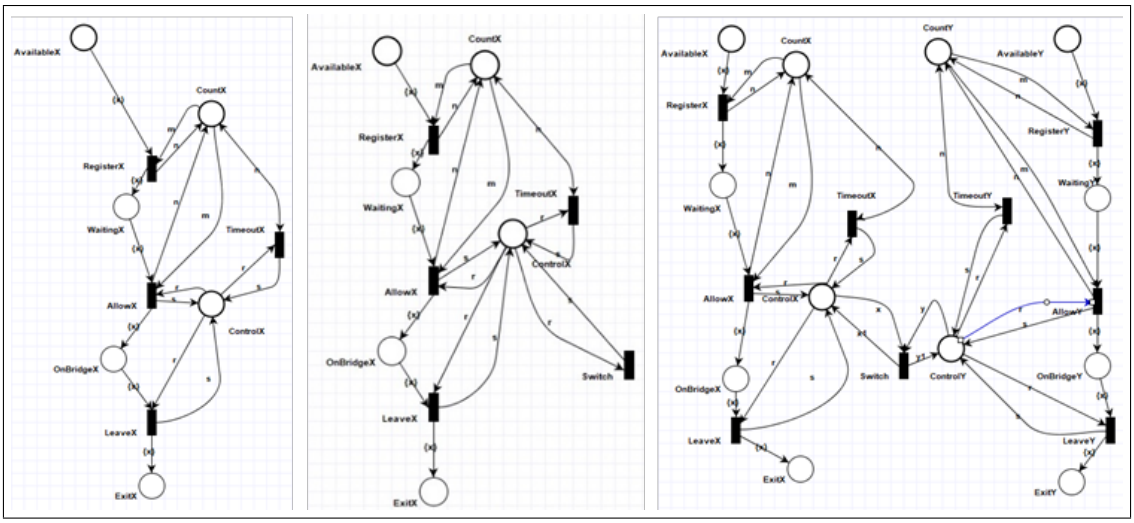

Figure 3.1: (a) The model of a controller; (b) A controller connected to a Switch and (c) Two controllers are connected to one switch.

and Vehicles system is presented here. However, the PrTN models of both systems with different parameters, the generated Promela models with properties, and the model checking results, can be found at https://bitbucket.org/ptnet/pipe/. The translated Promela models were analyzed in SPIN to verify the following properties,

1. The number of vehicles on the bridge never exceeds the maximum allowed
2. All the vehicles on the bridge must move in the same direction,
3. All the vehicles eventually cross the bridge, and
4. It cannot happen when some vehicles are at the starting point, and others have crossed the bridge.

Properties (1) to (3) are the desirable – expected to hold, but property (4) is undesirable – expected to fail. These properties are specified using LTL expressions as shown in Listing 3.11. Line 1 specifies the property (1), here P is a constant that denotes maximum number of vehicles allowed on the bridge. Line 2 specifies property (2). Property (3) is specified in lines 3 and 4. Line 5, specifies property

(3) for the vehicles on the one side and the line 6 specifies the same property for vehicles on the other side. Finally, line 5 specifies the property (4) .

Listing 3.11: LTL reperesentation of the properties used in the experiments

```
1 []!((len(place_onBridgeA)>P || len(place_onBridgeB))>P)
2 []!((len(place_onBridgeA)>0 && len(place_onBridgeB)>0))
3 for all x, [](routeA(x) -> <>exitA(x))
4 for all x, [](routeB(x) -> <>exitB(x))
5 []!(len(place_routeA)>0 && len(place_exitA)>0)
```

In these experiments, properties (1) to (3) on all the Promela models up to (20,10,10) have been checked successfully without any violation, and however, due to the complexity of these models, SPIN quickly reached the allotted memory-bound of 2048 Mbyte. Thus, only limited state space is searched. The reported violation can be false positive. All verification runs of the above properties in a particular model resulted in similar results except slight time differences since the state space explored is the same bounded by the allotted memory. A summary of SPIN model checking results of property (1) in the translated Promela models using the translation schemes mentioned in sections 3.2.3.12, 3.2.3.13, and 3.2.3.14 are shown in Tables 3.5, 3.6 and 3.7 respectively. These experiments show that the first scheme explores much smaller state space and is very fast than the second one, and the third one sits in-between as expected. However, the second scheme effectively detects violations in some safety properties, which found a counterexample of property (4) in 12 milliseconds, while the other schemes failed in finding a counterexample in bounded search space when DFS search is performed. The results are different when the BFS search strategy is used. Table 3.8 summarizes the results of the same experiment setting for the property (4) with the BFS search strategy. For this prop-

Parameters		State Transitions	Search Depth	Time (s)
(4,5,2)	Depth	18187377	5283	14.8
	Breadth	8495181	34	10.7
(10, 10, 10)	Depth	14616315	6927	12.8
	Breadth	7730107	46	10.6
(20, 10, 10)	Depth	11022109	7859	11.7
	Breadth	4527824	66	7.62

Table 3.5: Model checking results of property (1) using passive scheme.

Parameters		State Transitions	Search Depth	Time (s)
(4,5,2)	Depth	58279449	99999	25.5
	Breadth	43700528	34	31.3
(10, 10, 10)	Depth	49004374	99999	21.6
	Breadth	37330392	44	29.7
(20, 10, 10)	Depth	38461300	99999	19.9
	Breadth	28856176	65	27.8

Table 3.6: Model checking results of property (1) using active process based scheme.

erty, When the BFS search strategy was used, the third approach performed even better. It found the counterexample within 12 milliseconds.

These experiments show that adopting the BFS search strategy to explore the state space graph covers almost half the state space that could be covered by the DFS strategy within the memory limit irrespective of models and translation schemes.

However, SPIN was unable to run large models due to the state-space explosion problem, which also happened the Petri model checking contest where none of the participating tools could verify the above high-level Petri net models.

Parameters		State Transitions	Search Depth	Time (s)
(4,5,2)	Depth	46185791	73933	31
	Breadth	22159456	34	19.6
(10, 10, 10)	Depth	37131611	74109	27.6
	Breadth	18748818	44	18
(20, 10, 10)	Depth	18775896	75469	21.7
	Breadth	9910622	65	14.5

Table 3.7: Model checking results of property (1) using agent-based scheme.

Approach	State Transitions	Depth	Time	Result
1	9323528	2841	23.3	Error Not Detected
2	234740	27	0.222	Error Detected
3	9324	21	0.012	Error Detected

Table 3.8: Checking results of property (4) with parameter (4, 5, 2)

3.2.6 Related Work

In recent years, translations among the representation of system models have become very common. It is especially useful when properties specified in one model can be best analyzed using another model's techniques. In recent years, many of the well-known models are translated into another. In [43], the author introduced the idea of integrating model checking with theorem proving as a primary design decision. In this work, the theorem prover PVS [44] was integrated with a CTL [45] model checker SMV [46] by defining CTL operators in PVS [44] and using SMV as a decision procedure for the CTL fragment. In the tool *p2b* [47], the author translated Promela code to the input language of symbolic model checker SMV. There are several projects and frameworks like Veritech [48], Model Checking Kit [49] provided translation between SMV, Murphi[50], SPIN, and STeP [51].

Translation based verification model is also extensively used in the context of software verification. The Java PathFinder (JPF) software verification tool [52], Bandera [53] are some of the earliest prominent examples. Both Bandera and earlier versions of JPF translate Java source code to Promela. However, later JPF moved away to model check natively using a customized java virtual machine. Similarly, translating the C program with varying complexity to Promela for model checking using SPIN also got popular. The techniques developed in [54, 55, 56] are some significant earlier works along this line.

Formal verification of Petri net models using SPIN has also been explored in the past several years. In [57], a simple technique was proposed to translate low-level Petri nets to Promela. Several other similar techniques were proposed in the literature to translate low-level Petri nets, but few works are available for high level Petri nets.

In [36, 14, 15], the authors proposed similar techniques to translate PrTNs to Promela and implemented some of them with some restrictions in the PIPE+ tool environment. Those concepts and their implementations suffer from several limitations. Some of the important limitations are – (1) the schemes were not generic enough to support a wide range of system models, (2) did not support advanced features like quantifiers in full first-order logic formulas, (3) the translation schemes were not flexible enough to tailor the translation process, (4) conceptual misunderstandings. Under this study, these problems are addressed.

The translated Promela code generated by the methods presented in [36] shows several technical discrepancies. First of all, *places* are represented using *arrays*. There is no technical problem with it but it needed to add three extra functions for each place to provide *add*, *remove*, and *test* functionalities. These can be easily achieved using built-in support for some other data structure, i.e., *chan*. Second,

the behavior is translated using a *do-while* loop where an *atomic* statement for each transition as guard. SPIN holds a guard *true* if the first line of the guarded statement is executable. Thus, all the guards are technically *true* whether the corresponding transition is enabled or not. This can lead to starvation of truly enabled transitions. This can also increase the state space by infinitely many times. Third, two different inline functions are generated for each transition for checking its enabledness and firing, but nothing is mentioned about how the *preset*, and *postset* are evaluated. Fourth, for universal and existential quantifiers, separate inline functions are generated, but their evaluation method is not mentioned. Finally, the *semantic consistency claim* does not hold due to the second discrepancy.

In translation methods presented in [14] and [15] addressed some of the issues mentioned in the previous paragraph and introduced several others. The new translation mechanisms eliminate these problems. Apart from these, two additional translation schemes are introduced. The model translation feature in PIPE+ is fully automated. Once a model is translated, powerful features of SPIN can be explored to verify the model's constraints and properties using iSPIN graphical user interface. The new agent-oriented translation scheme facilitates incremental system modeling and analysis of cyber-physical systems [19], and can help detect modeling error before the whole system is built.

3.3 Summary

In this chapter, the techniques available in PIPE+ to analyze discrete systems are discussed in detail, along with newly added features. The simulation experience is enhanced in several ways. The simulation algorithm is by optimizing the selection of transitions and the evaluation of their constraints. Several other optimizations are

made in constraint parsing and evaluation engines. New functionalities are incorporated to realize the full first-order logic, set theory fully. The translation of PrTN models to Promela also improved. A more accurate, complete, and generic translator is provided, which translates all the elements of a PrTN model appropriately and consistently. Also, some additional methods of translation are introduced.

CHAPTER 4

MODELING HYBRID SYSTEMS

In this chapter, a formal definition of HPrTNs along with a modeling methodology is provided. This chapter starts with discussing some of the characteristics of CPSs that are considered while defining the formalism (section 4.1.1) and an overview of the scope of the methodology. Section 4.2 provides a formal definition of HPrTNs along with its dynamic semantics. In section 4.3, a modeling method is using HPrTNs is discussed. Section 4.4 demonstrates the application of HPrTNs via two case studies. This chapter is based on the publications [19, 20].

4.1 Overview

4.1.1 Characteristics of CPSs

A cyber-physical system (CPS) integrates digital computations (cyber components) with monitoring and control of entities in the physical world, i.e., operations and processes in physical devices (physical components). The cyber components monitor and control the physical components, usually with feedback loops where physical processes affect computations and vice verse. Figure 4.1 shows a straightforward workflow of how the components of a CPS interact with one another. Sensing and manipulation of the physical component usually occur locally in the devices. Cyber components analyze these sensor data and send out control signals as needed. These components may be embedded inside the physical devices or may be external. These components communicate with one another in some ways to produce the feedback control loop. Thus, CPSs are real-time, embedded, and distributed systems with heterogeneous components. A CPS model must capture these behaviors.

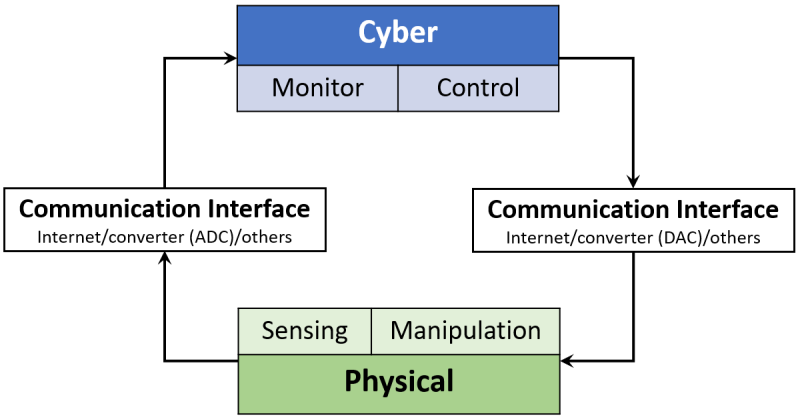

Figure 4.1: A simplest CPS workflow

Physical processes are compositions of many co-occurring things, unlike software processes, which are deeply rooted in sequential steps. Physical processes are compositions of many parallel processes. Measuring and controlling the dynamics of these processes by orchestrating actions that influence the processes are the main tasks of embedded systems [58]. Thus, CPSs intrinsically is highly concurrent.

4.1.2 Scope

In reality, CPSs exhibit several other characteristics. Some have strict QoS requirements; some are not. QoS requirements such as accuracy of the communication, timeliness, resiliency, fault-tolerance, and security dictate how individual components of the system under consideration coordinate with one another [59]. To summarize, going beyond the traditional distributed, concurrent and embedded systems, CPSs show some defining characteristics - (1) potential cyber capability in physical components, (2) high degree of automation, (3) communication at multiple scales, (4) integration multiple temporal and spatial scale, and (5) reorganizing and re-configuring dynamics [60].

A CPS model comprises models of the physical processes, the models of computation, and the feedback loop. The models of the feedback loop between the physical processes and the computation include modeling sensors, actuators, physical dynamics, and networks with failure, delays. The complete and faithful modeling of such systems is extremely challenging, typically involves a vast number of heterogeneous components. Although these components can be modeled following a bottom-up approach incorporating composition semantics, a rigorous effort to model all of these subtleties may result in a model too complex to analyze, which defeats the purpose of modeling - analyzing the essential system behaviors.

Therefore, we limit our scope to provide a modeling methodology that captures the essential system behavior and abstracts away the underlying complexities of networking. We have proposed Hybrid Predicate Transition Nets (HPrTNs), an extension to PrTNs, introducing the capability to capture the behaviors of physical processes and the feedback loop without networking. HPrTNs are well suited for capturing distributed, concurrent nature of CPSs. HPrTNs also provides a convenient means to model composition. Thus HPrTNs offer a natural correspondence to capturing the essential elements of CPSs. In this chapter, we will define the HPrTNs formally along with its dynamic semantics. We also will provide modeling strategies using HPrTNs. We will demonstrate these strategies though modeling of some well-known CPSs.

4.2 Hybrid Predicate Transition Nets (HPrTNs)

HPrTNs are extensions to PrTNs where all syntax and semantics of PrTNs are preserved. Thus HPrTNs retain the functionalities and capabilities provided by PrTNs. Several new concepts and ideas are introduced to HPrTNs, especially con-

tinuous place and token evolution. Continuous places are designated to store the system attributes. Tokens in continuous places can evolve without explicit transition firing. The rules of the evolution are specified using differential equations which are associated with the corresponding continuous places. The following subsections provides the formal definitions of HPrTNs and its semantics.

4.2.1 Formal Definition

Definition 4.2.1. An HPrTN is a tuple $N = (P, T, F, \alpha, \beta, \gamma, \mu, \lambda, M_0)$, where

1. $P = P_d \cup P_c$ is a non-empty finite set of discrete places P_d and continuous places P_c (graphically represented by circles and double circles respectively);
2. T is a non-empty finite set of discrete transitions (graphically represented by bars and boxed bars respectively), which disjoins P, i.e., $P \cap T = \emptyset$;
3. $F \subseteq P \times T \cup T \times P$ is a flow relation (the arcs of N);
4. $\alpha : P \to Type$ associates each place p in P with a *type* in $Type$. $Type$ defines the structure of the data the places can hold and consists of basic types such as $String$, $Integer$, and $Real$ numbers, and composite types defined using Cartesian product and power set. Each continuous place can only have $Real$ number components in its type definition;
5. $\beta : T \to Constraint$ associates each transition t in T with a constraint. Each constraint is a disjunction $\bigvee_i d_i$ for $i \geq 1$, where each disjunct d_i is a conjunction $pre_i \wedge post_i$ that define the enabling condition (precondition) and the processing result (post-condition) of a case of t respectively. The precondition contains only the variables appearing in the labels of incoming arcs, and the post-condition contains the variables appearing in the labels of outgoing arcs;

6. $\gamma : F \rightarrow Label$ associates each arc f in F with a label in the form of a simple variable x or a set element $\{x\}$. An arc label denotes the data flow of a relevant transition, where the variable is instantiated with concrete token(s) during the transition firing;

7. $\mu : P_c \rightarrow (\mathrm{R} \times \mathrm{R})^n$ associates each component of a continuous place with a pair of lower and upper bounds, which define the valid range of that component;

8. $\lambda : P_c \rightarrow (DifferentialEquation)^n$ associates each component of a continuous place with a differential equation that defines its evolution;

9. $M_0 : P \rightarrow Token$ associates each place p in P with a set of tokens. Tokens in $M_0(p)$ are values respecting the type of p. For a continuous place, the following constraint is imposed: $\forall p \in P_c.(|M_0(p)| \leq 1 \wedge \bigwedge_i \mu(p)[i][1] \leq M_0(p)[i] \leq \mu(p)[i][2])$, i.e. each continuous place contains at most one token and the values of its components are within the bounds.

4.2.2 Dynamic Semantics

The dynamic semantics of HPrTNs are defined using the concept of markings $M : P \rightarrow Token$ that are mappings from places to tokens. Markings of the discrete places constitutes the states of the discrete components and the markings of the continuous places constitutes the states of the continuous components. State of the discrete component changes via the firing of the enabled transition(s). On the other hand, changes in continuous states occur via token evolution. The dynamic semantics of the continuous components are defined using the concept of token evolution in continuous places and the feedback loops between the discrete components and the continuous components.

Definition 4.2.2 (Enabled Transition)**.** A transition t in T_c is enabled in a marking M if one of its preconditions is true. Formally, $\forall p \in P.(\overline{\gamma}(p,t) : \theta \subseteq M(p) \wedge \exists i.\beta(t).pre_i : \theta)$, where $\overline{\gamma}(p,t)$ is a generalization of γ such that $(p,t) \notin F \Rightarrow \overline{\gamma}(p,t) = \emptyset$. $e : \theta$ is the result of instantiating all arc variables with tokens in p according to substitution θ.

Definition 4.2.3. The firing of an enabled transition t results in a new marking M' defined by: $\forall p \in P.(M'(p) = M(p) \cup \overline{\gamma}(t,p) : \theta - \overline{\gamma}(p,t) : \theta)$. We denote this firing as $M \xrightarrow{t/\theta} M'$.

The firing of a transition is instant and does not consume time. Two enabled transitions are in conflict if the firing of one of them disables the other. Non-conflict transitions can fire concurrently.

Tokens in continuous places are continuously evolving according to the differential equations governing their change rates as long as their bounds are not violated. The token evolution in continuous places is similar to the firing of transitions. The difference is that, to be able to fire a transition have to be enabled. On the other hand, the token of a continuous place can be modified only if the new values of all the elements of the token are within the specified range.

Given a marking M, we use $[M]$ to denote state space covering all possible continuous token evolution with the same token distribution.

Definition 4.2.4 (Token Evolution in Continuous Places)**.** Let, $\theta_i = \lambda(p)[i](M(p)[i])$ be the result of the evaluation of the differential equation of ith element of the token of a place $p \in P_c$, then the evolution of the token of the place p is defined as $M'(p) = \forall_i.(\bigwedge_i \mu(p)[i][1] \leq \theta_i \leq \mu(p)[i][2])) \wedge \cup_i(\theta_i)) \vee M(p)$

Definition 4.2.5. Let T_i be a set of concurrently enabled non-conflict transitions with corresponding substitutions θ_i in marking $[M_i]$, and $[M_{i+1}]$ is the resulting

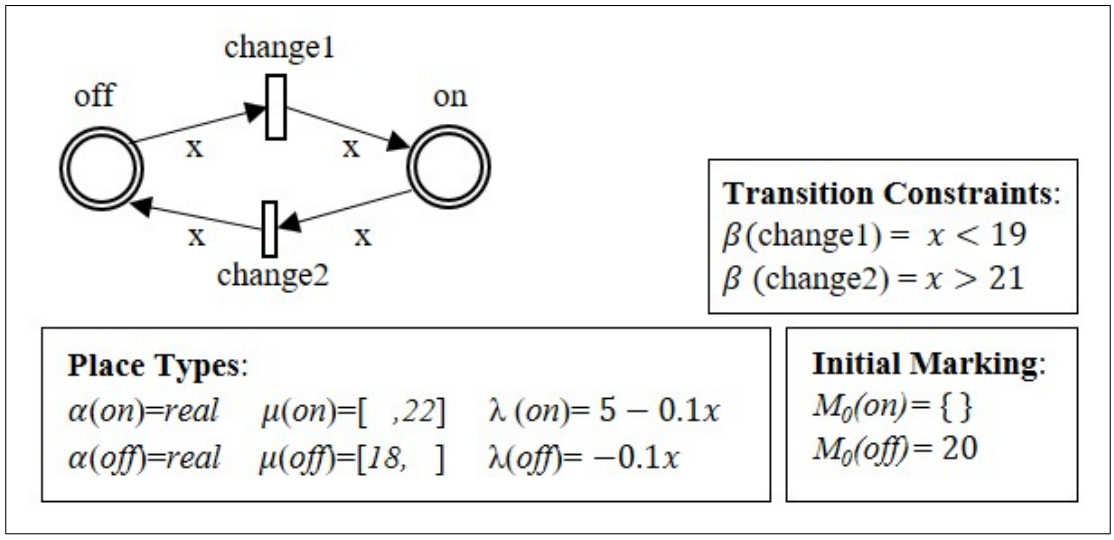

Figure 4.2: Thermostat system: An HPrTN model

new marking after firing transitions concurrently. This transition step is denoted as $[M_i] \xrightarrow{T_i/\theta_i} [M_{i+1}]$. The behavior of a net N consists of the set of all the firing sequences $[M_0] \xrightarrow{T_0/\theta_0} [M_1] ... [M_i] \xrightarrow{T_i/\theta_i} [M_{i+1}] ....$ The set of all reachable markings is denoted as $[[M_0] >$.

4.3 Model Development

HPrTNs preserves the syntax and semantics of PrTNs. Thus, modeling discrete components is similar to as described in section 2.3.1. In this section, only the modeling of the continuous components and their interaction with the discrete components are elaborated.

4.3.1 Modeling States

The entities having continuously changing dynamics are modeled using continuous places. A continuous place is represented with a double circle in PIPE+ (see Figure

4.2). A continuous place captures the distinctive mode of a continuous variable and thus can hold at most one token at any time. When the entities having continuous dynamics and their attributes are identified, a datatype (data structure) is created to represent those entities as a token. Each attribute is specified as a field in the datatype definition. The fields of a datatype for continuous components are a little different than that of the discrete components. The fields of a datatype designed to be associated with continuous place must have a valid range $lower < upper$ which dictates the possible minimum and maximum values the corresponding attribute can hold. This range serves as an invariant. The evolution of a continuous variable is modeled through the changes of token values inside a continuous place. These changes can be defined using ordinary differential equations (ODEs) or difference equations or arithmetic expressions.

In case when multiple entities are involved in a specific mode, all the attributes of all the entities should be part of the same continuous place. For example, the model shown in Figure 4.4, three aircrafts are involved in the collision avoidance maneuver. The datatype designed for a place consists of all the attributes needed to describe all the three aircraft in that particular mode.

4.3.2 Modeling Evolution

In all prior works on hybrid Petri nets, including [4, 61, 19], continuous transitions were used to define continuous state evolution. Following the similar concept, the concept of continuous transitions is introduced to HPrTNs [19]. However, the presence of continuous transition increases the modeling complexities in several ways. First of all, the concept of a transition is inherently discrete and the use of continuous transitions creates conceptual confusion. Secondly, it introduces multi-way

conflicts, i.e. conflicts between continuous and discrete transitions, continuous transitions and discrete places, continuous places and discrete transitions. Resolving these needs extra level of constraints which unnecessarily complicates the modeling. It also assumes modelers' responsibility to produce conflict-free models.

In the revised methodology [20], the concept of continuous transition is eliminated by introducing a novel idea – token evolution (Definition 4.2.4). HPrTNs is redefined such that continuous state evolution is defined by the changing values of the tokens residing in continuous places while discrete state change is represented by token movement caused by discrete transition firings. A continuous place shown in double circle (see Figure 4.4) captures the distinctive mode of a continuous variable denoted by a token. The evolution of a continuous variable is modeled through the token value change inside a continuous place and defined using an ordinary differential equations (ODEs). The ODEs can be linear or non-linear and first-order or higher-order. The evolution of a continuous variable can also be represented using difference equations. The ODEs and the difference equations are fundamentally similar in terms of expressiveness. But they differ in the way they are evaluated. During evaluation, these equations are numerically solved to compute the amount of change. Solutions to the ODEs are more precise and error margin is less than the solutions to difference equations. The evolution rules can also be expressed using explicit arithmetic formulas which are basically closed form of the ODEs.

In the explicit form, an ordinary differential equation is a function $y'(t) = f(t, y)$, where $y'(t)$ is the derivative of y with respect to time, f is a function of time t and the current state y. A solution to y' is differentiate-able function Y of time that satisfies the differential equation when substituted into it, i.e. substituting $Y(t)$ for y and time-derivative $Y'(t)$ of Y at t for $y'(t)$. That is, time-derivative of the solution at each time is equal to the right hand side of the differential equation [62].

This is analogous for higher-order differential equations, i.e. involving higher-order derivatives such as $y''(t)$ or $y^{(n)}(t)$ for $n > 1$.

Representing first order ODE in PIPE+ is straightforward. The ODE needs to be specified in a specific format. The format is explained here with the example of a falling object. The dynamics of a falling object at a specific time t is the height $h(t)$ and velocity $v(t)$. The relationship between h and v is $\frac{\delta h(t)}{\delta t} = v(t) \implies \delta h(t) = v(t)\delta t$ or simply $h' = v$. On the other hand, the v depends on the acceleration i.e. g the gravitational force in this particular case. Now, suppose, the continuous place used to represent these dynamics has the datatype $\langle v : number,\ h : number \rangle$. Then the general form of the differential equations specifying these dynamics and their representations in PIPE+ are shown in the equations (4.1) and (4.2), in which h contains the value of the initial height and v hold the value of the velocity at the beginning of the interval. The format of the representation of the ODEs in PIPE+ starts with δ followed by an identifier t, and then inside parentheses there are two comma separated elements. The first one is the actual ODE expression and the second one is the initial value. The ODE is a string literal expressed as a function f of t. Note that, here t is the change variable as it follows the symbol δ. PIPE+ expects the ODE to be a string literal. The string is passed to the differential equation solver as it is. If current values from input tokens or computed values of other sorts are to be used to construct the ODE, then those need to be converted to string. This can be done by concatenation of their string representations. Concatenation can be done via the '*concat*' function. As shown in equation (4.2), the current value of v is used in the ODE of h. So, '*concat*' function is used there to convert v into a string literal. The '*concat*' function is a variable argument function, it can take as many number of argument as necessary. The ODE must be a function of time t. So, the change variable for this case should always be t. Moreover, using other symbols may

produce undesirable results. The interval of change is provided by the logical clock variable τ internally.

Solutions to higher order and non-linear ODEs are complex and are often approximated through simpler representation. Often such ODEs are represented in the state space form, $\dot{x} = f(x)$ where $x \in \mathrm{R}^n$ for $n \geq 1$ is a vector. For example the second order non-linear ODE $\beta x_3 + \gamma x_2 + \alpha x_1 = 0$, where, $x_3 = \dot{x_2}$, $x_2 = \dot{x_1}$, and $x = \begin{bmatrix} x_1 \\ \\ x_2 \end{bmatrix}$, can be represented by equation (4.3). To model this kind of complex dynamics the modeler needs to specify the ODE in state space form. Equation (4.4) shows the general format of specifying ODE in state space format in PIPE+. The equivalent representation of the state-space form shown in equation 4.3 in PIPE+ is specified by equation 4.5. Like the linear ODE, the state space representation also needs to be specified as a string literal. Also, the values of the dynamics used in the state space representation should be in the same token and in a specific order. For example, the datatype for that continuous place holding that token should be designed in such a way that the second element of a token is the derivative of the first element, the third is the derivative of the second and so on. The pendulum (section 4.4.2) uses state-space form to specify the dynamics of the pendulum.

$$h' = v, \ v' = -g \tag{4.1}$$

$$h1 = \delta t(concat(v), h) \wedge v1 = \delta t(\text{“} - 0.098\text{”}, v) \tag{4.2}$$

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ \\ -\frac{\gamma}{\beta}x_2 - \frac{\alpha}{\beta}x_1 \end{bmatrix} = f(x) \tag{4.3}$$

$$\delta x\left([state\_space\_representation], [initial\_values]\right) \tag{4.4}$$

$$y1 = \delta x([\text{“}x[1], -\gamma/\beta * x[1] - \alpha/\beta * x[0]\text{”}], [y[1], y[2]]) \tag{4.5}$$

$$x1 = \Delta x(u + \alpha * \tau) \tag{4.6}$$

$$x1 = v * \tau + 0.5 * \alpha * \tau^2 \tag{4.7}$$

The equation 4.6 shows an example of difference equation. A difference equation starts with the symbol Δ. The following symbol denotes the changing variable. Then, inside a parentheses the expression is specified as an arithmetic expression. The expression is evaluated to compute the change. The equation 4.7 shows as example of using arithmetic expression directly to compute the evolution of the dynamics x.

4.3.3 Modeling Feedback Loop

Feedback loop is the way the discrete and the continuous components interact to each other. Its purpose is to control the evolution of continuous component. It can be done in the following two ways

- Updating control parameter. If the token evolution of a continuous place uses some parameters to control the changes then changing the values of those parameters suffice to control the evolution
- Token movement. If there is no parameters to govern the changes in the rate of token evolution or the evolution rule changes as a whole then the token is removed from the current continuous place to be inserted to another continuous place having the desired evolution rule.

Both ways should be triggered by firing of designated transition. In the first option, the parameters should be part of the token in the continuous place. The transition should update the parameters by modifying the token held by the continuous place. In the second options, the transition should remove the token from the original continuous place and put the token in the designated continuous place.

4.4 Case Studies

HPrTNs were used to model several well-known hybrid systems [63], including bouncing ball, thermostat, robotic motion controller, and obstacle avoidance etc. In this section, we demonstrate the application of HPrTNs through two additional well-known CPSs.

4.4.1 Air Traffic Collision Avoidance

In air traffic control, collision avoidance maneuvers are used to resolve conflicting flight paths that arise during free flight [1]. These are very important and complex applications. Due to the complicated spatial-temporal movement of a traditional airplane, simply braking and waiting is not an option to resolve conflicts, because the plane must maintain minimum speed for sufficient lift [1]. Consequently, the combination of discrete control choices and temporal evolution of the continuous dynamics of air traffic maneuvers result in very complex hybrid behaviors. Several prominent approaches are proposed in the literature to tackle this problem. In [1], the author provided a comprehensive survey of many of these approaches, introduced some maneuvers for air traffic collision avoidance, and provided formal verification of their correctness. As our case study, we model one of these maneuvers – Fully Flyable Tangential Round-About Maneuver (FTRM). Figure 4.3(a) shows the protocol cycle

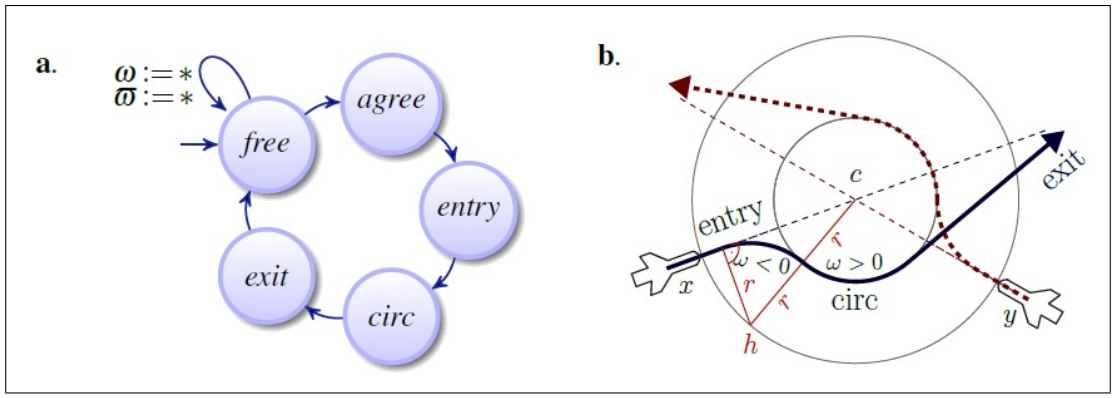

Figure 4.3: Protocol cycle (a) and construction (b) of FTRM [1]

of FTRM and in Figure 4.3(b) the corresponding flight phases are shown for two airplanes. FTRM is a horizontal collision avoidance maneuver, thus only planar dynamics are considered.

According to the protocol, during the $free$ flight, two airplane move by choosing arbitrary angular velocities ω and $\bar{\omega}$ respectively. When two airplanes come closer within a safe distance they enter the *agree* phase. In this phase, the roundabout center (c) and radius (r) are determined. As the airplanes approach to the center, they come to a compatible entry point and enter the *entry* phase. Approaching the roundabout circle in a right curve with $\omega < 0$ around an arbitrary anchor point (h), they come to a tangential position to the circle and enter the *circ* phase. In this phase, the airplanes follow the roundabout circle with $\omega > 0$. Finally, the airplanes *exit* the roundabout.

The dynamics of an airplane is represented by the position $a = (a_1, a_2) \in \mathrm{R}^2$, the flying direction, $d = (d_1, d_2) \in \mathrm{R}^2$ and the angular velocity $\omega \in \mathrm{R}$. Where d represents both the linear velocity $\|d\| = \sqrt{(d_1^2 + d_2^2)}$ and the orientation of the aircraft in the space. The evolution of the flight dynamics of an aircraft is given by the differential equations, $\dot{a} = d$ and $\dot{d} = \omega d^{\perp}$ where, the vector $d^{\perp} = (-d_2, d_1)$ is the orthogonal complement of d.

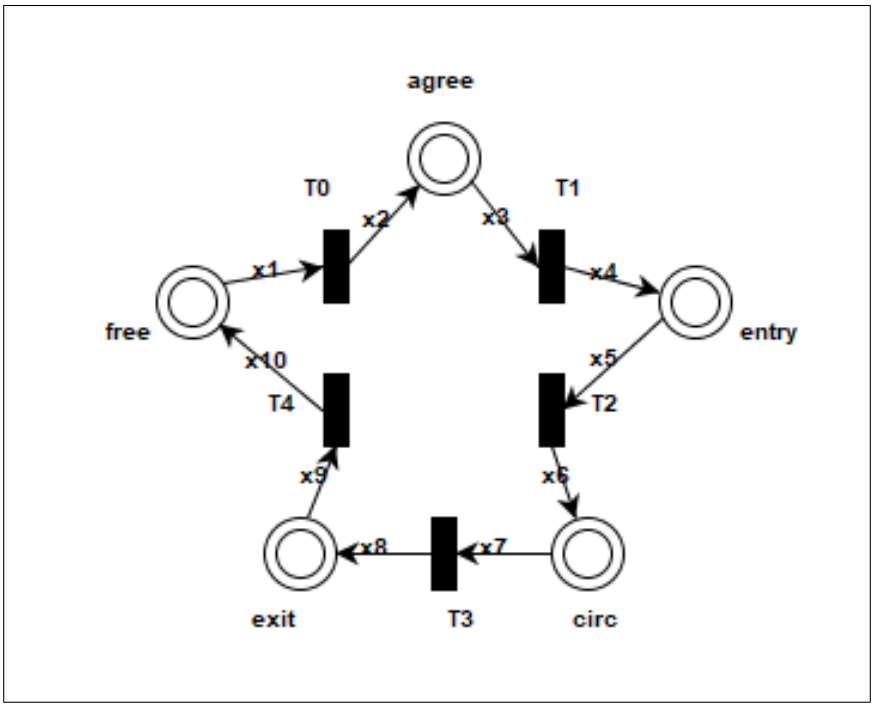

Figure 4.4: A pictorial diagram of the HPrTN model of the FTRM maneuver involving three airplanes

There are several ways to choose the roundabout center c, radius r, entry point x and the anchor point h. In our model, we choose c as the center of the circle (C) that goes through the locations of the airplanes x and y. The radius r is half the radius of C, that is $\|x - c\| = 2r$. The entry point x is determined such that, $\|x - c\| = r\sqrt{3}$. During the entry and *circ* phase the angular velocity ω is chosen such that $\|\omega\| = \|d\|/r$ to maintain the same ground speed. The anchor point h is also located on C and $\|h - c\| = 2r$ and $d = -\omega(x - h)^{\perp}$. In our model, we assume that the all airplanes have the same ground speed. These can be extended to use different speeds and orientations by modifying the initial markings of some places.

Figure 4.4 shows the pictorial diagram of the HPrTN model of FTRM generated in PIPE+. This model captures the five distinct models of airplane maneuvers in the protocol using five continuous places and thus is easy to understand. A structured token is used to represent three airplanes participating the FTRM, which captures

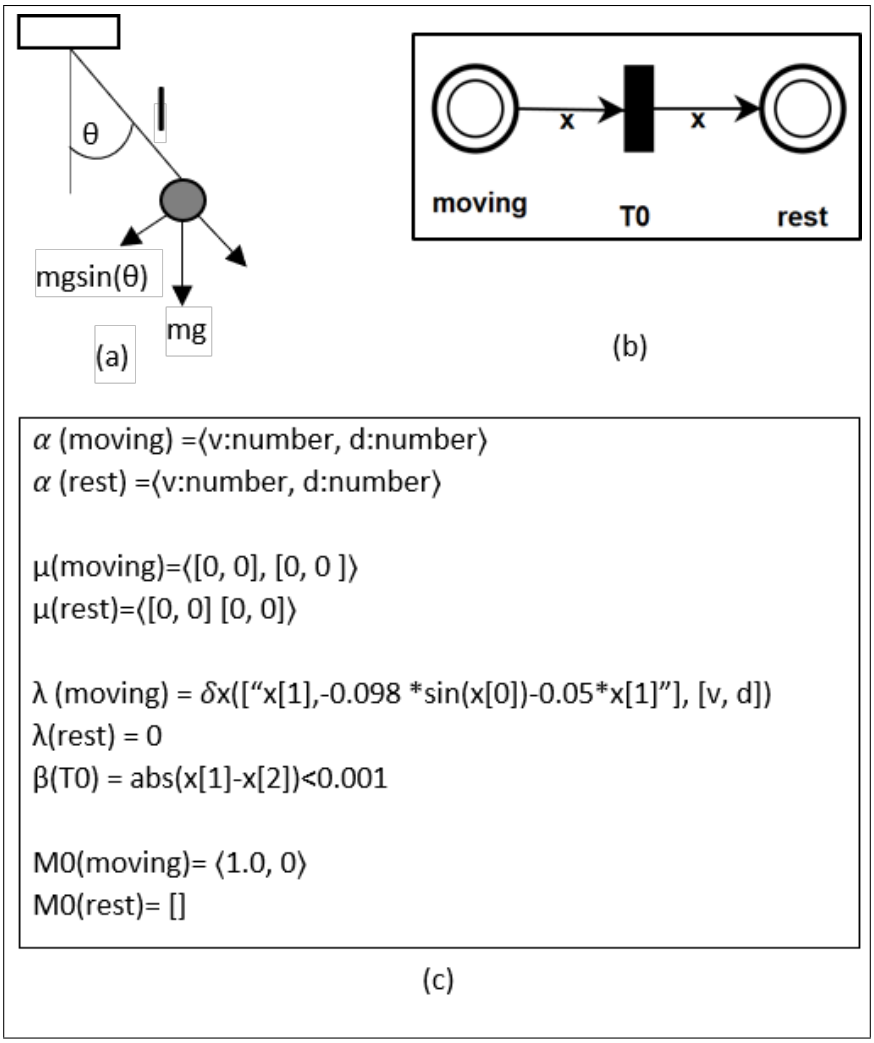

Figure 4.5: (a) Free body diagram of a Pendulum; (b) HPrTN model diagram of the Pendulum; and (c) Inscription of the HPrTN model of the Pendulum

location, *velocity*, and angular displacements of the airplanes. These continuous variables of airplanes are defined by corresponding differential equations. Furthermore, additional fields of the token store the agreed upon shared parameters – such as the center and the radius of the roundabout and the angular velocity along with the current phase of FTRM the airplanes are performing. Each discrete transition uses the corresponding phase change criterion as precondition and updates the control parameters.

4.4.2 Pendulum

We model the motion of a simple pendulum using HPrTNs. Although this example is not a typical CPS system, we choose this system to show the expressive power of HPrTNs to model non-linear dynamics using transfer function in PIPE+. Here we model only the effect of gravity on a pendulum. Figure 4.5 (a) shows the free body diagram of a simple pendulum. Suppose $\theta(t)$ is the angle the pendulum makes with the vertical line at time t, l is the length of the pendulum, m its mass and d is the dissipation coefficient. The ODE (4.8) determines the evolution of θ, which is a non-linear second order ODE. This equation is solved through transformation into to equivalent state space form, $\dot{x} = f(x)$. Assuming $x = \begin{bmatrix} x_1 \\ \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \\ \dot{\theta} \end{bmatrix}$, the equivalent state space representation of equation (4.8) is given by equation (4.9). This model does not contain Zeno execution since the time diverges when the movement of the pendulum is within a small threshold (0.001) of displacement, the discrete transition fires and takes the token from continuous place *moving* to *rest*.

$$ml\ddot{\theta}(t) + dl\dot{\theta}(t) + mgsin(\theta(t)) = 0 \tag{4.8}$$

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} x_2 \\ \\ -\frac{g}{l}sin(x_1) - \frac{d}{m}x_2 \end{bmatrix} = f(x) \tag{4.9}$$

4.5 Related Work

4.5.1 Modeling Hybrid Systems

Many formal models have been proposed to combine discrete reactive systems and continuous-time dynamic systems to describe complex systems where both of these dynamics naturally come together. *Hybrid automata* [64, 3] is one of the earliest formalisms. In the simplest form, a hybrid automaton is an extended finite-state machine where each state is different control state. Each states has a set of real-valued variables and constrained with continuous evolution rules. Its simplicity made it widely accepted. To work with more complex systems several extension to hybrid automata were introduced. To model hybrid systems having heterogeneous component a hierarchical approach is proposed in *Ptolemy* [65]. For distributed and concurrent systems a compositional approach was proposed in *Hybrid I/O Automata*[66]. In recent years, a logic based approach, *Differential Dynamic Logic (Logic d$\mathcal{L}$)* [67], gained popularity.

The concept of extending Petri nets formalisms to provide means to model continuous and hybrid systems was first presented in [68]. Based on this concept, several other extended Petri net models were proposed.

In [68], the authors combined a continuous Petri net representing continuous dynamics with a discrete Petri net capturing discrete behaviors. Subsequently, the authors extended their formalism to provide a distinction between autonomous and timed hybrid Petri nets and provided rules to resolve conflicts between continuous and discrete part [5, 69]. Hybrid Petri nets are based on low-level Petri nets where tokens in continuous places are numerals and change rates associated with continuous transitions are simple difference equations. A slightly different approach was introduced in [70]. Here new kind of places and transitions were introduced,

namely differential places and differential transitions. Differential places constitute the continuous state of the system being modeled. Differential transitions are always enabled and associated with a firing frequency, where first-order ordinary differential equations are used to represent the evolution rules. Another class of hybrid Petri nets is fluid stochastic Petri nets introduced in [71], which extended stochastic Petri nets to model hybrid stochastic systems. Apart from these, several other prominent works were published to extend other classes of Petri nets, batch Petri nets, hybrid flow nets, etc., to support modeling of hybrid systems. Along with the research on the extension of the low-level Petri nets, several classes of high-level Petri nets have also been extended for modeling hybrid systems. One of such early approaches was proposed in [11], where a method was presented to extend timed hierarchical object-related nets (THORNs). In this extension, the author introduced real data type to THORNs to represent the continuously changing state variable and continuous transitions to capture the continuous evolution. In this approach, a continuous transition was enabled or disabled by inhibitor arcs and the evolution was specified using ordinary differential equations. However, this approach was not fully developed and supported by any tool. Among other classes of high-level Petri nets, Colored Petri nets were studied extensively and several approaches for extending them to model hybrid systems were proposed in [9, 10, 72].

4.5.2 Hybrid Petri Net Tools

Although, both low-level and high-level Petri nets had been undergone rigorous studies and many extensions were proposed to model hybrid systems, not many efforts were made to provide proper tool support. Among low-level hybrid Petri net tools HYPENS [73], SimHPN [74] and HISim [75] are some of the well known

tools. Both HYPENS and SimHPN are not native Petri net tool and were based on MATLAB and Simulink. They do not provide proper net editing capabilities. A user needs to use MATLAB/Simulink components to specify the semantics of the Petri net model of the system being modeled. HISim, on the other hand, integrated modeling and simulation in a unified tool but is functionally incomplete. In [10], the authors proposed a different approach to creating a model using MATLAB components for simulation and provided a methodology to translate that into CPN for analysis. Among the tools in this context, Snoopy [72, 76] provided a unified experience of creating a graphical model, simulation, and analysis; but focused on modeling biological systems. This tool supports several hybrid low-level and high-level Petri nets but not suitable for CPSs.

4.6 Summary

In this section, the concepts of HPrTNs are defined formally. HPrTNs preserves the definition and semantics of PrTNs. The concept of continuous place is introduced to capture continuous dynamics. HPrTNs also introduces a novel concept, token evolution, for the evolution of continuous states. Token evolution nicely resolves the conflicts present in other related works. A modeling method is presented to model hybrid systems using HPrTNs. The method presents the ways to capture different aspects of the hybrid systems. The method also shows the ways to represent both linear and non-linear dynamics using different forms of differential equations. Modeling of two hybrid systems, the pendulum (section 4.4.2) and the air traffic collision avoidance (section 4.4.1), are shown. The pendulum system has non-linear dynamics modeled using second order differential equation. The other exhibits simpler dynamics but involves the interaction of multiple entities having similar dynamics.

CHAPTER 5

ANALYZING HYBRID SYSTEMS

This chapter focuses on the techniques to analyze HPrTN models. This chapter starts by providing an overview of the types of analysis techniques established in the literature. Section 5.2 discusses the analysis techniques via simulation as implemented in PIPE+. In section 5.3, reachability analysis techniques using SpaceEx is provided. Section 5.4 discusses some recent development regarding the analysis of hybrid systems.

5.1 Overview

Several techniques are available for analyzing PrTN models in PIPE+, including simulation, model checking [13, 17, 18], bounded model checking [34], and term rewriting [38]. These techniques are suitable for discrete systems. However, not suitable for analyzing hybrid systems because of the technical limitation to compute the system evolution. In discrete systems, the system state change via transitions. On the other hand, in hybrid systems, the system state evolves continuously following the evolution rules and transitions that usually separates the evolution modes. The evolution rules are specified using differential equations and that need to be solved to compute the evolution. The existing techniques cannot do that. Also, being infinite-state systems, it is not possible to fully explore the state space. Thus the existing techniques are not directly applicable to analyze HPrTN models.

Simulation is a widely used technique to get an overview of the system behavior under known conditions. In the simulation, the system model is executed step by step from the initial state. In each step, the successor states are computed following the evolution rule. A simulation run usually follows a single trail of execution. It is

useful to have a quick insight into the expected functioning of the system. However, it does not guarantee that it will behave correctly under all circumstances. For this, systematic methods, model checking, need to be employed. Several model checking techniques have been developed for hybrid automata [77].

Being infinite-state systems, explicit state model checking is not an appropriate method to verify hybrid systems. **Reachability Analysis** (also termed as *symbolic reachability analysis*) is one of the few techniques to verify hybrid systems. In this approach, the main goal is to decide whether a set of states reachable from the initial states of the system via all possible execution paths are safe or not. These techniques are incredibly challenging and computationally infeasible, depending on the complexity of the model and the complexity of the dynamics. Some techniques have been developed for reachability analysis of a subclass of hybrid automata - linear hybrid automata, where the $init$, inv, $flow$, and $jump$ functions are Boolean combinations of linear inequalities [77]. For systems with complex and non-linear dynamics, *theorem proving* is one of the available analysis techniques, which is extremely difficult and requires much expertise. However, no model checking technique is available for hybrid Petri nets, not even for low-level hybrid Petri nets [61].

Under this work, functionalities for simulating HPrTN models in PIPE+ are implemented. Also, a translation-based reachability analysis technique is provided. The simulation can be done directly from the tool PIPE+. For reachability analysis, the HPrTN models are translated into the input language of `SpaceEx`, one of the states of the art reachability analysis tools. The translated models can be analyzed with SpaceEx. In the following sections, these are elaborated.

5.2 Simulation

5.2.1 Simulation Strategy

The simulation strategy for HPrTNs is similar to that for PrTNs, as mentioned in section 3.1.1. The net is executed step by step following the dynamic semantics of HPrTNs. The difference here is the presence of both discrete and continuous dynamics. Since the evolution of continuous state is different from discrete state transitions, different strategies are needed. The discrete state transitions follow the technique described in section 3.1.1. For continuous evolution in every step, all the continuous places are evaluated. During the evaluation, the first-order logic formula containing the invariants and differential equations associated with the continuous places are evaluated. After evaluation, the marking of a continuous place is updated with the values computed by solving the differential equations only if its invariants are satisfied. The whole net can be executed one step at a time or can be configured to run a certain number of steps. If the simulation is run for multiple steps, then the simulation may stop before the configured number of steps if there no evolution. This may happen if all the discrete transitions are disabled, and invariants of all the continuous places are not satisfied in the current marking.

5.2.2 Analyzing Results

To visualize the simulation results, the marking of the continuous places can be plotted in charts. Only, 2D time-series and scatter plots are available to plot the values. The time-series charts show the evolution of a configured dynamics against time or steps to be more specific. In the scatter-plot, the evolution of one dynamics can be plotted against another. Only configured charts will be presented to the user.

Apart from the charts, the complete trace of the simulation is stored. These traces can be used with other third-party tools for more sophisticated analysis.

5.2.3 Technical Challenges

The major challenge during simulation is to numerically solve the differential equations associated with continuous places to capture the dynamic evolution of continuous variables denoted by tokens. Java is the implementation language of the PIPE+ environment for which no sophisticated numerical method is available to solve linear and non-linear ODEs with transfer functions. Some external libraries support linear ODEs but have an unsatisfactory performance - either too slow or having high approximation errors. Experiments with the *Python* `scipy` module reveals better results in solving ODEs. Using a Python package inside a Java application requires a means for inter-process communication between a Java process and a Python process. To avoid the costly operations such as creating a separate process for every ODE and to relieve the user from installing additional special Java communication libraries through Java Native Interface (JNI), A cleaner way has been found for process communication by implementing a lightweight python interpreter that is capable of performing only a few commands necessary to evaluate ODEs. This interpreter runs one background python process. The interpreter and the python process communicate to each other via designated input and output streams created for the python process. The major advantage of this approach is its simplicity and the creation of less number of sub-processes. However, this approach is not scalable. This should be replaced with other implementation in the future of the tool. This can also be improved to let the users to select the numerical methods, approximation error thresholds, and other controls available in the scipy library.

5.2.4 Case Studies

5.2.4.1 Simulation of FTRM)

As a case study, this model of the aircraft collision avoidance, as presented in section (4.4.1 is simulated with different sets of initial conditions, i.e., initial locations, velocity and directions of the airplanes, different safe distances. Figure 5.1 shows the plot of locations of the airplanes A, B, and C as they move towards each other and participated in FTRM. In this simulation run, the initial locations of the airplanes A, B, and C are $(-50, 0)$, $(-19, 46)$ and $(46, 19)$ respectively. Their directions are 0, $13\pi/8$, and $9\pi/8$ degrees in radians, respectively, from the positive x-Axis. All of them have an equal ground speed of 5 $miles/minute$. These initial values are chosen so that the airplanes would have met each other at location $(0, 0)$ if they did not perform the collision avoidance maneuver. Finally, the safe distance is assumed to be 25 $miles$. In this case study, only the collision avoidance maneuver is modeled. How the airplanes would follow their original course after exiting the roundabout is not considered.

5.2.4.2 Simulation of Pendulum (4.4.2)

The model was simulated using various initial values and parameters in PIPE+. Figure 5.2 shows the simulated evolution of the angular displacement (x_1) and the velocity (x_2) of the Pendulum model and Figure 5.3 shows the phase plane trajectory of the model.

5.3 Reachability Analysis

Hybrid systems are infinite-state systems since the state variables represent continuous dynamics and are represented using real numbers. So, the explicit state model

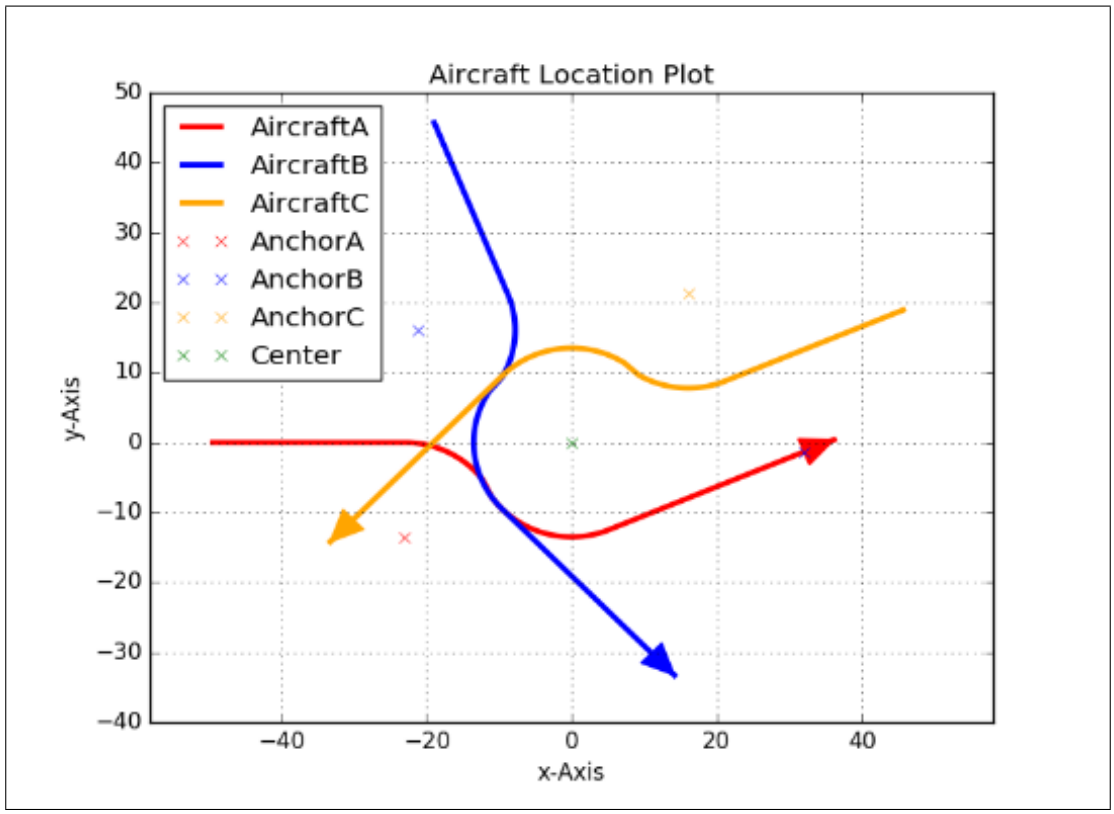

Figure 5.1: Simulation result of the model in Figure 4.4

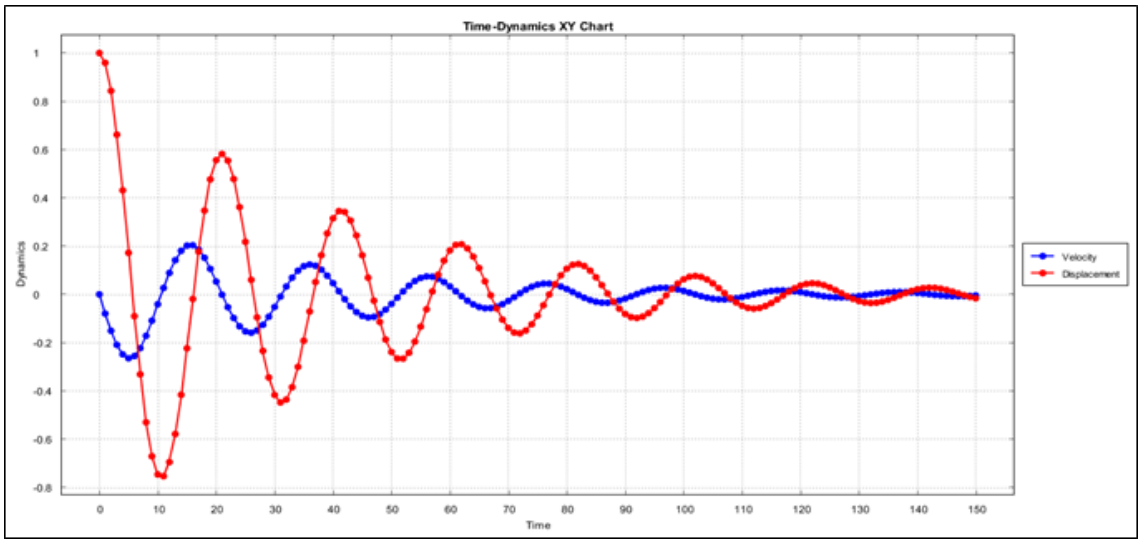

Figure 5.2: Trajectory of the dynamics of the Pendulum

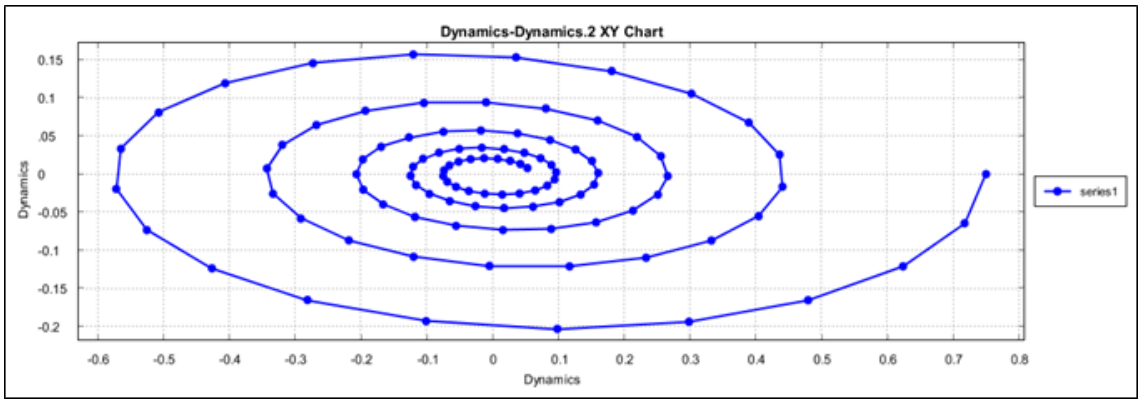

Figure 5.3: Phase plane plot of the trajectory of the Pendulum

checking techniques available for discrete event systems are not applicable. Reachability analysis is one of the established techniques to analyze hybrid systems. Here the reachable set is computed symbolically from the initial states by following the dynamic evolution rules. If some states fall inside the pre-defined set of unsafe states, then it is determined that the model is not sound and needs refinement. There are several challenges to compute these reachable sets. Many algorithms and techniques are available to tackle specific areas of these challenges. However, a few support is available to analyze hybrid Petri nets. Under this study, a translation-based method is developed to provide reachability analysis for HPrTN models. In this method, the tool *SpaceEx* [12] is being used. In the following subsections, an overview of the tool *SpaceEx*, the translation method, and analysis results are presented.

5.3.1 SpaceEx Format

SpaceEx models are represented using an XML based format known as `sx`. `sx` models are similar to the hybrid automata but more hierarchical. These models are either a single hybrid automaton or a network of hybrid automata. In `sx`, the definition and the instantiation of the corresponding hybrid automata are done separately.

Some of the essential elements of `sx` models concerning the elements of HPrTNs are described here briefly.

5.3.1.1 Components

Components are the top-level elements in `sx` models. A model is made up of one or more components. There are two types of components – *base component* and *network component*. Each base component essentially represents one hybrid automaton when translated to hybrid automata and defines its variables, structures, transitions. Network components, on the other hand, instantiate other components (either base or network). These components represent a parallel composition of several hybrid automata. These components consist of –

- **Parameters.** A formal parameter can be a continuous variable with arbitrary dynamics, a constant variable, or a synchronization point. Formal parameters are part of the interface of a component. Parameters can be local to the component of or can be globally available for the entire model.
- **Locations.** Locations are part of base components. Each of these locations represents the location of the corresponding hybrid automaton. Each of these locations are associated with *invariants* and *flow* relations. Flow is a set of differential equations defining the time-driven evolution of the continuous variables. The system can be in the same location as long as the *invariant*s are satisfied.
- **Transitions.** Transitions define the discrete jumps from one location to another. Each transition is associated with a *guard*. A jump can take place only when the guard condition is satisfied. A transition usually changes the dynamics. The immediate modification of the dynamics is specified using *assignments*

to the continuous variables. They also associated with a synchronization *label* and *assignment*.

- **Bindings.** Bindings are part of network components only. Bindings are used to specify the connection between other components. These connections may be parallel or in series. There are some established restrictions on the binding.

Components can be nested. But recursive nesting is not allowed. A *state* consists of a location and the instantiation of the parameters. Execution of the automaton is a sequence of discrete jumps and continuous trajectories. A state is reachable if an execution leads to it. Given a set of forbidden states, the system is safe only when no execution leads the system to one to he forbidden states.

5.3.1.2 Instantiating Components

The definitions of the components do not make them part of the system automatically. Those must be instantiated. Instantiation takes place in network components only. The process of instantiation of a component `A` inside a network component `B` is done according to the following the steps – (1) creation of a copy of `A`, (2) association of that copy with a name unique within the scope of `B`, (3) binding of each of the formal parameters of `A` with either a formal parameter of `B` or with a constant value. Component of `A` can be instantiated as many times as needed.

5.3.1.3 Component Composition

The composition of components is defined inside a network component. A network component is the parallel composition of its member components. These member components can be independent or synchronized. Synchronization takes place via shared parameters or shared labeled transition. If a shared parameter is used for synchronization, only one of the participating base components should control it.

Every other component needs to mark that parameter as uncontrolled explicitly. There is no such constraint while synchronizing via labeled transition. Nevertheless, only one participating component should specify the guard condition to avoid ambiguity and conflict.

5.3.1.4 Dynamic Semantics

When SpaceEx parses an `sx` model for analysis, each base component is translated into a hybrid automaton. Each location of the component becomes the location or vertex of the hybrid automaton with the usual semantic mapping of the location's elements to those of the vertex. Each transition of the component is translated as a jump. A jump may take place only when the guard condition of the transition is satisfied. When a jump occurs, it may change the values of the continuous attributes according to the associated assignment. The system may remain in the same location as long as it satisfies the invariants. All behavior starts from a given set of initial states. Execution of an automaton is a sequence of discrete jumps. Each of the network components is translated into the parallel composition of its sub-components. The parallel composition of hybrid automata is also a hybrid automaton.

Semantically, `sx` models are non-deterministic and acausal. Any component can declare any variable and can impose the restriction and constraints on the variables and their derivative. Variables can be local or global. Different components can impose constraints on the same variable independently. If the constraints are contradictory, then there will be no solution to the differential equation for that variable. There may not be any trajectory for that computational path and time stops for the model. This scenario may occur due to modeling errors, or this may be the desired behavior. The modeler needs to be careful about this.

5.3.2 Translation Strategies

Components are the building blocks of `sx` models. An `sx` model constitutes a set of components and their synchronous or asynchronous compositions. Being a distributed state computation model, `HPrTN` is well suited for modular or component-oriented and incremental system modeling. Where smaller components of the whole system can be modeled separately and then by the synchronous or asynchronous compositions of these models, the whole system can be obtained. The asynchronous composition can be achieved by sharing a discrete place or connecting two places on different components. The synchronous composition can be achieved by sharing a discrete transition. It can be argued that whether modeled using composition or not, an HPrTN model can be viewed as a composition of one or more components. These components can be translated into base components in the `sx` model.

Now one strategy to translate an HPrTN model to an `sx` model would be to identify the components, then translate those components individually and then apply techniques to make the composition of these components. In the following subsections, some strategies to achieve these are discussed.

5.3.2.1 Identifying Components

The basic idea behind the concept of components is that each component is viewed as a hybrid automaton and eventually will be transformed into one by SpaceEx during analysis. So, without loss of generality, we can safely call each of these components a hybrid automaton interchangeably. Generally, the nodes of a hybrid automaton specify the evolution rules of similar dynamics and take part in the transiting system involving these dynamics. Similarly, a continuous place also specifies the evolution rules of similar dynamics. It may form a network consisting of other continuous places that define the rules for the same dynamics and discrete transi-

Algorithm 6: Algorithm for finding candidate components

Data: $N = (P, T, F)$, an instance of HPrTN model
Result: C, a set of strongly connected components

```
1  begin
2      A ⟵ empty set
3      C ⟵ empty set
4      for each arc a ∈ F do
5          if a is connected to a continuous place then
6              A ← make node a
7          end
8      end
9      for each a ∈ A do
10         if a is not visited then
11             visit(a, A)
12         end
13     end
14     while A is not empty do
15         Q ⟵ empty set
16         a ← remove first from A
17         append a to Q
18         for b ∈ A do
19             if find(a) = find(b) then
20                 remove b from A
21                 append b to Q
22             end
23         end
24         append Q to C
25     end
26     D ⟵ empty set
27     for each arc a ∈ F do
28         if a is not connected to a continuous place then
29             D ← make node a
30         end
31     end
32     append D to C
33     return C
34 end
```

Algorithm 7: Supporting algorithms used in Algorithm 6

```
function visit(a, A) begin
    mark a as visited
    U ⟵ the set of arcs connected to/from endpoints of a
    for u ∈ U ∩ A do
        if a and u can belong to the same component then
            if u is not visited then
                visit(u, A)
                union(a, u)
            end
        end
    end
end

function union(u, v) begin
    if find(u) ≠ find(v) then
        if u.rank >= v.rank then
            v.parent ← find(u)
            u.rank ← u.rank == v.rank?u.rank + 1 : u.rank
        end
        else
            u.parent ← find(v)
        end
    end
end

function find(u) begin
    if u.parent = u then
        return u
    end
    else
        u.parent ← find(u.parent)
    end
    return u.parent
end
```

tions connecting them. If such a network or a connected component formed in this fashion can be found in the HPrTN graph, then it can be said that this component is equivalent to a hybrid automaton and can be translated to a component in the `sx` model. Other such connected components can be identified if there is any. Also, the continuous places that do not form such a network can be thought of as independent components. It is important to note that discrete places can be ignored safely while finding connected components because discrete places necessarily are not part of the transitioning system of continuous dynamics. Finally, once all the components involving continuous places are identified, the only nodes left in the graph are discrete elements. One separate component can be formed for these remaining discrete elements. The continuous components connected to these discrete elements can be in a composition in the final translated model.

Algorithm 6 summarizes the overall idea implemented in PIPE+ to identify the components. It first tries to find the components consisting of continuous places and the transitions that connect them. This starts with finding the arcs that are connected to continuous places (lines 4-8). It then picks an arc a and starts traversing the net in depth-first fashion following both successor and predecessor arcs of a. It follows an arc b only if b is compatible with a. In the current strategy, the compatibility is determined using the datatype that is associated with the continuous places connected to a and b. If the datatypes are the same, it is expected that both of these continuous places are manipulating the same dynamics. So, both a and b will be part of the same component. The algorithm uses the `union-find` data structure with path compression and rank, as explained in [78]. When the whole net traversal is completed, all the arcs having a continuous place as an endpoint are associated with a component. The association then separates these arcs, and the corresponding component sets are created (lines 14 - 25). Another set is created for

the arc that does not have a continuous place as an endpoint, this set determines the component for the discrete elements.

5.3.2.2 Translation of Components

Once the candidate components are identified, each of them is converted into a base component of the translated`sx` model. The translation strategy is described in Section 5.3.3. Once the base components are defined, a new `network component` is created for each of these components so that each of them can be individually analyzed. Also, a network component is created, combining, and compositing all the base components to get the whole system's instance.

5.3.2.3 Compositing Components

The composition of base components is done inside a network component in an `sx` model, usually by binding some params of each component. SpaceEx provides two different ways of composition, namely – parallel (synchronous) composition and sequential (asynchronous) composition. The first one is done via synchronizing on transition labels. If two components have some transitions with the same label, they can be bind to some shared labels defined in the network component. The other form of composition is done by variable chaining. In other words, by attaching an output variable to the input of another. This can also be done by binding a variable from different components to the same variable defined in the network component.

5.3.3 Translation Methods

The components identified following the strategies explained in the previous subsection (5.3.2) are good approximations of the subsystems of the whole system.

Typically, a subsystem has several discrete control states, where each control state has different evolution rules and a way to transition among these discrete states. Another aspect of these subsystems is that they can evolve independently irrespective of the rest of the system. They may be connected to other parts of the systems using a minimal number of connections forming sequential composition. The components identified in an HPrTNs model also have similar characteristics. These components constitute continuous places containing the evolution rules, discrete transitions that define the transitioning system, arcs that define the flow of transitions. Each of these elements plays different roles. To translate these components or subsystems, these elements need to be translated appropriately. In these subsections, the translation methods of these elements within the context of a component or subsystem are discussed.

5.3.3.1 Translation of Continuous Places

The continuous places are used to store the values of continuously changing dynamics while defining the evolution rules. Ideally, different continuous places should define different evolution rules. Depending on the discrete control state, one and only one of the continuous places within a component should be active. These semantics closely resemble the `locations` of hybrid automata, where each location provides different evolution, and only one of the locations of a hybrid automaton is active at a certain moment.

On the other hand, a component in `sx` models will eventually be translated into a hybrid automaton, where a location in the component will be mapped to a location in the resultant automaton. We can conclude that the continuous places in HPrTN can be best represented as locations within the appropriate component.

Although **location** is the correct choice to represent continuous places, this may fail to translate the exact semantic of continuous places. Semantically, as long as a continuous place contains a token, it is considered as active. Now, if multiple places considered to be part of the same component have tokens, they would be all active and evolving according to the semantic of HPrTN. Nevertheless, their representations in sx or a hybrid automaton cannot have the same behavior because only one location can remain active at the same time. The algorithm can detect this situation at the time of translation but cannot guarantee the semantics' soundness. It depends on the modeler to ensure soundness.

5.3.3.2 Translation of Discrete Places

From the perspective of hybrid systems, discrete places do not have a much important role in the actual flow of the systems. Instead, they are mostly used as part of the communication and control. Here, communication means the exchange of evolution or evaluation data between two or more components. In other words, discrete places may be used as synchronization points or the basis for the continuous components' sequential composition. Discrete places can be used as controllers as well. For example, the evolution of some continuous components is parameterized. By controlling the value of the parameters, the evolution of the continuous components can be controlled. Discrete places can also be used as part of the feedback control loop. Figure 5.4 shows a hypothetical hybrid system model showing some of these usages. In the example, the place `P4` is being used as a source of parameters, the place `P8` as a feedback control mechanism. The place `P7` along with the discrete transitions `T7` and `T8` establish a sequential composition between the two-hybrid components. The discrete places are translated to facilitate these usages.

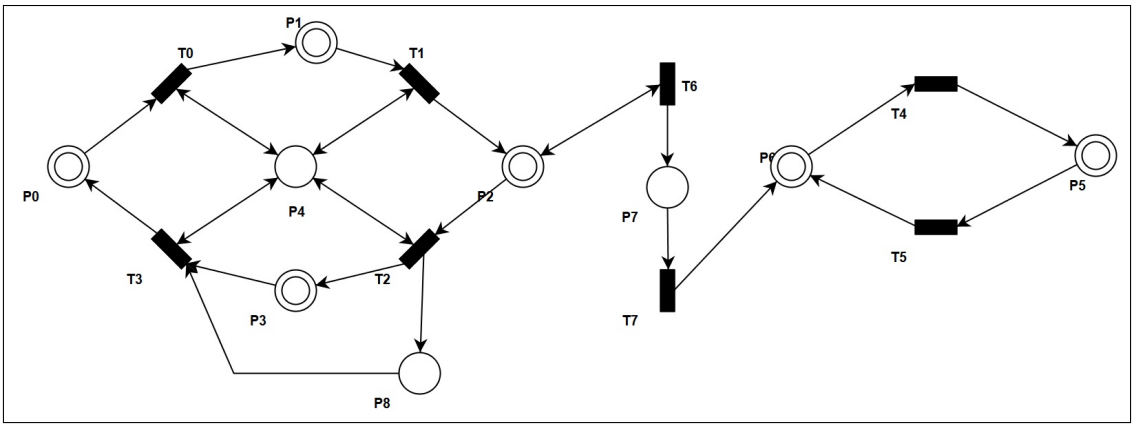

Figure 5.4: Hypothetical model showing example components and their composition

To facilitate control, discrete places can be effective in two places. One is the flow relation of the locations of the components and the assignment of the transitions between locations. The flow relation defines the dynamics and the assignment specifies the new values to be transferred as a result of the transitions. In `sx` models, the flow relations are specified in the form of $\frac{dx}{dt} = ax + cu$, where x is the dynamic variable, a and c are constants and u is a set of any arbitrary non-deterministic variables. The discrete places can be transformed to represent these variables. One way to achieve this is to convert the discrete places to the formal parameters to some components specially designated for the discrete components. With the proper sequence of composition, those variables can be bound to some of the parameters obtained by translating the discrete places.

Although transforming discrete places into parameters can be an excellent way to support those cases, problems may arise if the discrete places are designed as power sets. In this case, these places can potentially have multiple tokens. That means there will be multiple sets of control variables, only one of which can be used.

5.3.3.3 Translation of Transitions

The translation strategies and techniques for discrete transitions can also be explained in terms of their usage. Like discrete places, discrete transitions can be used in the HPrTN model in several ways. A transition can be completely part of a hybrid component, for example the transitions `T0 - T5` as shown in figure 5.4. Transitions can be completely part of the discrete control component (e.g. `T6`), completely part of a continuous component, or shared among multiple components.

In the first case, the translation is straightforward. The transitions are part of the hybrid component. They directly mapped to the transitions between two locations. It is a safe operation because the transition is part of a strongly connected component, which by construction ensures that the transition has at least one input continuous place and one output continuous place. In the translated model, the transition can expect at least one source location and one destination location. It is desired that a transition has at most one input location and at most one output location. Technically is okay for a transition to have multiple continuous places in HPrTN, but it causes some unnecessary complexity in the translated `sx` model. For example, if a transition has two input and two output places, there will be four different possibilities of transitions in the translated model. However, it is possible to detect distinct paths from the transition constraint, which may lead to fewer transitions, but the computation would be very complicated. Moreover, this may bring an extra level of non-determinism, which may cause adversely in the verification performance.

The translation of the second case is also similar to the first one. Each of such transitions will be translated as a transition element within the designated component for the discrete component. The guard condition of this transition element would be the translated version of the precondition, and the translated version of the

postconditions will represent the element assignment. The name of the transition will be used as the label.

5.3.3.4 Translation of Arcs (F)

Arcs provide the flow of the evolution of the system. In other words, arcs help to identify the input entities and output entities of a transitioning system. However, the information and insights the Arcs provide are helpful during the translation process. However, in the translated model, they do not have any significant role. That is why Arcs are not translated into an essential element of the target network.

5.3.3.5 Translation of DataTypes (α)

The type mapping, $\alpha : P \leftarrow Type$, associates each place to a $Type$, which defines the structure of the data the place can have. A $Type$ is a multi-set consisting of one or more available sorts, *string* and *number*. Each of the sort mentioned in a type definition is termed as a `field`. Each field represents a dynamic attribute or constant or a label and can be accessed by a name. Whether discrete or continuous, each place must be associated with a $Type$. Multiple places can share $Type$. In `sx`, the concept of structured type is not available. Rather, each of the dynamics, constants, or labels are accessed individually. These are defined as `parameter`s (*param*) at the component level.

The $Type$s are translated as a set of *param*s. Where each $field$ in a Type is translated as a separate *param*. Each of these fields in HPrTN has a name, a sort(type), and a range defining the lower-bound and upper-bound of the allowed values for this field. On the other hand, apart from the corresponding *name* and *type* attributes, a *param* has several other attributes. Table 5.1 summarizes the available attributes for fields and params. The translation method maps the attribute `name`

of a field to the `name` of a param. The attribute `type` of field is mapped to `type` of param but the values undergo a little conversion. The sort *tring* is represented using *label* and the *number* as *real*. The values of other attributes of param are inferred according to the context the param is used. All of the fields are one dimensional, so values for $d1$ and $d2$ are always 1. For strings, the dynamics are assumed to be constant. All the variables are considered global, and if some variables take part in synchronization during composition then the corresponding params in the network component are marked as controlled

Field in HPrTN	*name*, *type*, *min*, *max*
Param in sx	*name*, *type*, $d1$, $d2$, *dynamics*, *local*, *controlled*

Table 5.1: Attributes of fields and params

Due to the strategies adopted for the automated translation, the naming of the fields should follow conventions. First of all, the variables are considered global at least up to component level. Thus, if some places in the same connected component have different data types, then the field names of those data types should be exclusive. Since the fields of all the data types used by the places of a connected component will be converted as params for the corresponding `xs` component, if fields from different data types have the same name, they will be mapped to the same param. If two fields from different data types do not represent the same dynamics/variables, their names should be different. This reasoning also applies to the field naming of the data types of different components. However, if two components are completely independent and never involved in a composition, then the naming conflict will not arise. The modeler needs to be aware of the naming conflict in the translated `sx` model.

5.3.3.6 Translation of Transition Constraint (β)

Discrete transitions in HPrTN are translated into transitions of components in `sx` models. While doing it the transition constraints are broken down into two parts – preconditions and postconditions. Preconditions are the logical operations to decide whether the transition is enabled and ready to fire. In hybrid automata or `sx`, the equivalent concept is the guard condition. When the guard condition is met, the jump can happen. Thus the preconditions are translated as the guard conditions. The preconditions in HPrTN are first-order logic formula specified in terms of the input variables or incoming arc labels. On the other hand, the guard conditions are also logic formula but specified in terms of the params of the component they reside in. The preconditions are transformed to replace the arc label with the equivalent params and then set as the guard condition.

Postconditions, on the other hand, are used to compute the outcome of a transition when it takes place. The equivalent concept in Sx mode is the element `assignment`. Similar to preconditions, the postconditions are also specified using incoming and outgoing arc labels. Similarly, the assignments need to be specified using the params. The translation method provides automated conversion of the postconditions to replace the arc labels with the appropriate params.

5.3.3.7 Translation of Labels (γ)

Arc labels (γ) are not an essential part of the translated model and are not directly converted to any element. However, these are used to identify the correct params during the translation of β.

Description	$\alpha(p_c) = (id : string,\ v : number)$ $\mu(p_c) = ((\mu_{0l}, \mu_{0u}), (\mu_{1l}, \mu_{1u}))$
Translation	$< component\ name = \text{“}c_{10}\text{”}\ ... >$ $< param\ name = \text{“}id\text{”}\ type = \text{“}label\text{”}\ ... / >$ $< param\ name = \text{“}v\text{”}\ type = \text{“}real\text{”}\ ... / >$ $< location\ id = \text{“}p_c\text{”}\ name = \text{“}p_c\text{”}\ ... >$ $< invariant >\ \mu_{1l} <= v < \mu_{1u} < /invariant >$ ... $< /location >$ .... $< /component >$

Table 5.2: Example translation of bounds

5.3.3.8 Translation of Bounds (μ)

The bounds, $\mu : P_c \longrightarrow (\mathcal{R} \times \mathcal{R})^n$, are introduced to HPrTN to impose an extra layer of constraint to ensure that the evolution of the corresponding dynamic attribute remains within the specified range. It will be an undesired situation if the evolution goes beyond the range. Thus, these bounds can be viewed as invariants to the corresponding dynamic attributes of the continuous place. A continuous place, P_c, is translated as a location, l_c inside a component. $\mu(P_c)$ will be invariant for that location. Table 5.2 show an example of the translation of bounds. The first row gives some hints about the relevant description of the place p_c. The second row shows a sample translation of $\mu(p_c)$.

5.3.3.9 Translation of Differential Equations (λ)

Differential equations, $\lambda : P_c \longrightarrow (ODE)^n$, associates evolution rules to the continuously changing dynamics designed by a continuous place, P_c. These equations are

translated to the `flow` relation of corresponding location in the `sx` model.

The tool PIPE+ provides several different representations to specify these differential equations, namely – linear ODE, non-linear ODE, and difference equations. Besides linear algebra, it also allows us to use some trigonometric function and some built-in inline functions. SpaceEx has some limitations on how these differential equations can be provided. SpaceEx supports only linear ODEs. The ODEs should be of the form $\frac{dx}{dt} = ax + bu$. Here, x is the dynamic variable and its change is defined concerning time t. Having these dynamics, the format of the flow relation is shown in the following expression.

$$< flow > x' == ax + bu < /flow >$$

5.3.3.10 Translation of Initial Marking (M_0)

In HPrTN, the initial markings define the initial state of the system. During simulation in PIPE+, these are used to initialize the system instance. These initial values are part of the systems. On the other hand, in SpaceEx, the initialization is configured separately. The initial values are not part of the model. Rather, the initial values need to be configured outside the model as a configuration file. Currently, no support is provided for the automated creation of the configuration files. However, the initial marking should be used to initialize the formal parameters of the system component. Also, the continuous places that have the initial marking should be the initial active location of their corresponding components.

5.3.4 Translation Correctness

The correctness of a translation method covers completeness and consistency. Completeness proves that all the elements of the translating model are translated. Con-

sistency refers to the semantic equivalence of the two models. The correctness of the translation method based on these two concepts is discussed here.

The translation method presented here is complete. Each of the elements of HPrTN has specific roles both in the structural and dynamic semantics. The roles of each of the elements are covered in the translation process, and an equivalent representation is provided. Also, by construction, all the elements mentioned in a particular instance of the HPrTN model are represented with appropriate elements in the translated model.

The translation method is consistent. It can be shown by construction. The previous subsections provide a way to construct an sx model from the HPrTN model. If an HPrTN model can be obtained from a given sx model, then it would be sufficient to say the translation method is consistent. From a given sx model an HPrTN model can be obtained in the following manner,

- Create a separate datatype from the formal parameters of each base component
- Create a continuous place p for each base component in the sx model and set $\alpha(p)$ with the corresponding datatype. Set the bounds $\mu(p)$ of some fields with the invariant of the component. Assign the flow relation to $\lambda(p)$.
- Create a transition t from each transition element in the sx model. Add an arc between a continuous place and the transition depending on the source and target attributes. Assign guard as the precondition and assignment as a postcondition of t and then obtain $\beta(t)$ as the conjunction of these two.
- From each network component, find the mappings. For all the mapping involving labels, merge the corresponding transition. All mappings involving variables, add an arc from a transition to place if the variable comes in the

guard of that transition and an arc from the place to transition if the variable is part of the assignment of the transition.

Thus, the elements of an HPrTN element are extracted from the given sx model and the inscriptions.

5.3.5 Case Study

Both simulation and reachability analysis are the available analysis techniques in the tool `SpaceEx`. The support for reachability analysis fundamentally operates on symbolic states. The tool also provides a generic platform so that different independent reachability analysis algorithms can be developed. Three different algorithms are available in SpaceEx, LeGuemic-Girard (LGG) algorithm [79], STC algorithm, and PHAver[80]. LGG is the primary analysis algorithm. In this algorithm, the set of the reachable state is over approximated by a set of polyhedra. STC algorithm is a recent enhancement of LGG, which produces fewer convex sets for a given accuracy and produces more precise images of discrete transitions. Each of these techniques different configurations.

For performing analysis, a configuration file is needed along with the `sx` model. The configuration file specifies the initial assignment to the unbounded or unmapped variables declared in the model and configuration for the chosen analysis algorithm. To demonstrate the correctness of the translation method, several systems modeled in HPrTN are translated to `sx`. The translated models are simulated in the SpaceEx tools to compare the results of simulation of these models in PIPE+.

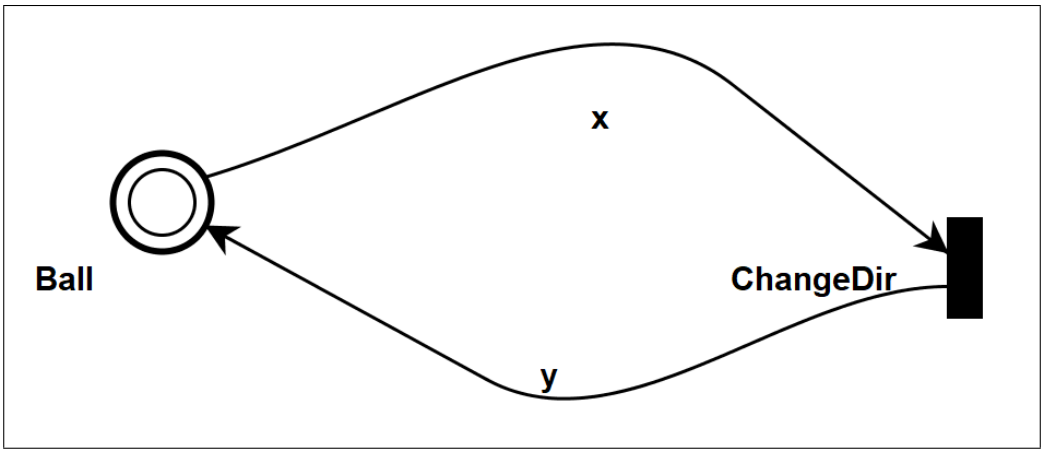

Figure 5.5: Pictorial diagram of the bouncing ball

$\alpha(Ball) = \langle v : nubmer, h : number \rangle$
$\beta(ChangeDir) = ((x[1] \leq 0 \wedge x[2] \leq 0) \wedge y = \langle -0.75 * x[1], x[2] \rangle)$
$\mu(Ball) = \langle (0,0), (0,\infty) \rangle$
$\lambda(Ball) = (h1 = \delta x(concat(v), h) \wedge v1 = \delta x(\text{“}-0.5\text{”}, v))$
$M_0(Ball) = \langle 0, 10 \rangle$

Table 5.3: Inscription of HPrTN model of the bouncing ball

5.3.5.1 Bouncing Ball

In this model, the physics of a bouncing ball, i.e., its motion before, during, and after the impact against another surface, is modeled. The state of the ball is captured when it falls freely from a place above the surface. The state is captured in terms of velocity (V) and height (h). Only the effect of gravity is considered in this model. The dynamics are rather straightforward. The Variables v and h are weakly coupled to each other. However, they form a piecewise affine dynamics, which makes it suitable to be analyzed with SpaceEx. Figure 5.5 shows a pictorial diagram of the hybrid Petri net model. Table 5.3 lists the inscriptions of the net. Here, the continuous place *Ball* is used to store the velocity and height, and their evolution. The transitions `ChangeDir`, switch the direction of the ball when its height becomes

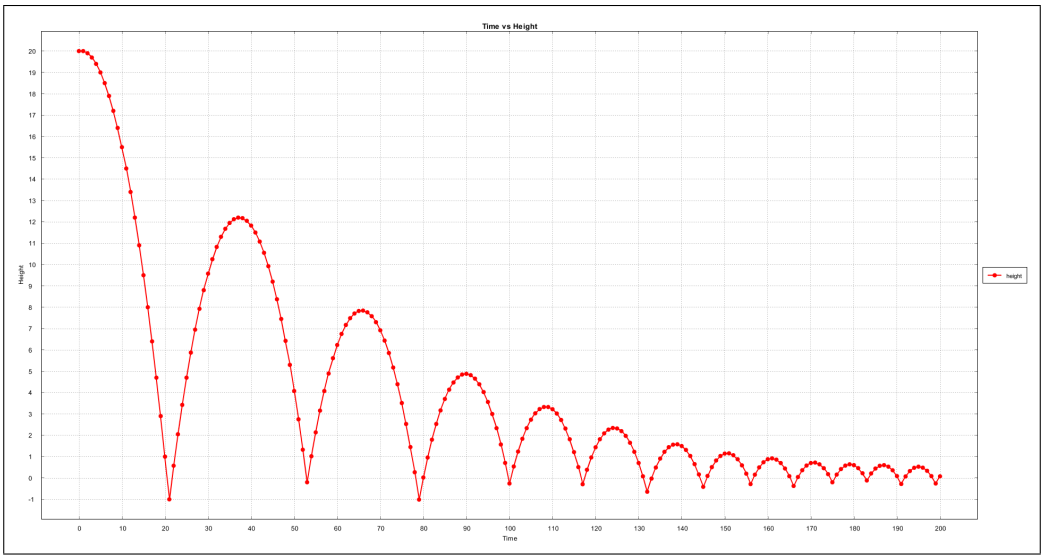

Figure 5.6: Trajectory of height of the bouncing ball

zero while going down. Figure 5.6 and Figure 5.7 show some of the simulation results of the bouncing ball in PIPE+.

The translation of this model is also straightforward. It consists of a single base component having one location and one transition. The transition makes a self-loop each time the dynamics see one of the equilibriums. The translated model was run with the tool using the available analysis algorithms. Similar analysis steps also carried out using an available model of bouncing ball modeled directly using the SpaceEx tool. This pre-built model is mentioned here as the original model. Table 5.4 shows some of the metrics collected during the analysis with both the translated and original models. Here, *time* denotes the time taken to finish the analysis of the symbolic state-space, and the *fixpoint* denotes the number of steps taken to find the fixpoint. Figure 5.8 also shows state spaces as captured in the SpaceEx tool. These charts show that the state-space generated using the simulation techniques provided in PIPE+ is similar to that obtained from the analysis techniques supported by SpaceEx. This also shows the consistency of the translation method.

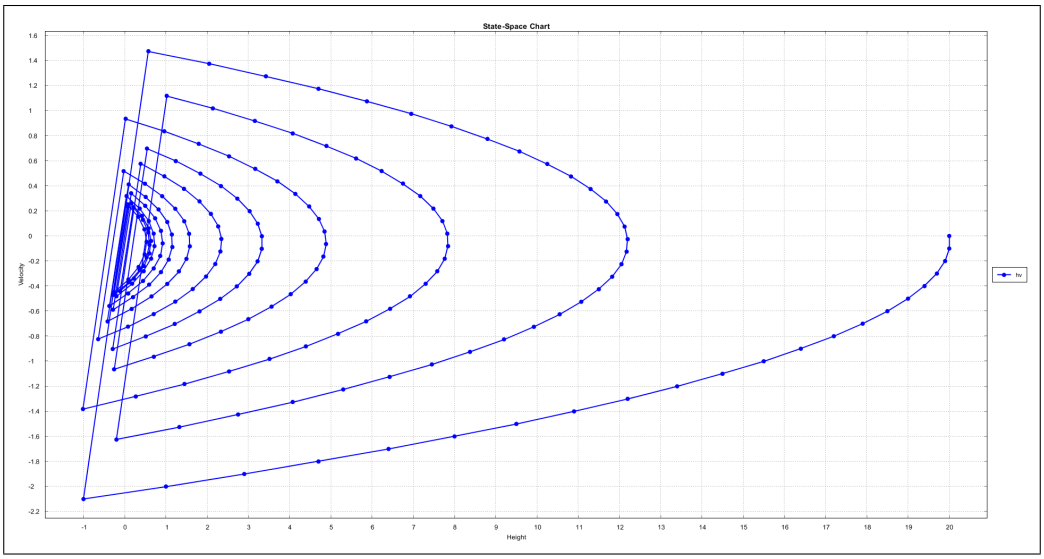

Figure 5.7: Bouncing ball state-space

	Translated model			Original model		
	Fixpoint	Time	Memory(KB)	Fixpoint	Time	Memory (KB)
STC	26	1.69	3036	26	1.714	3036
LGG	8	0.4	3040	8	0.41	3040
Simulation		0.847	3036		0.841	3040

Table 5.4: Statistics of runtime summary of bouncing ball

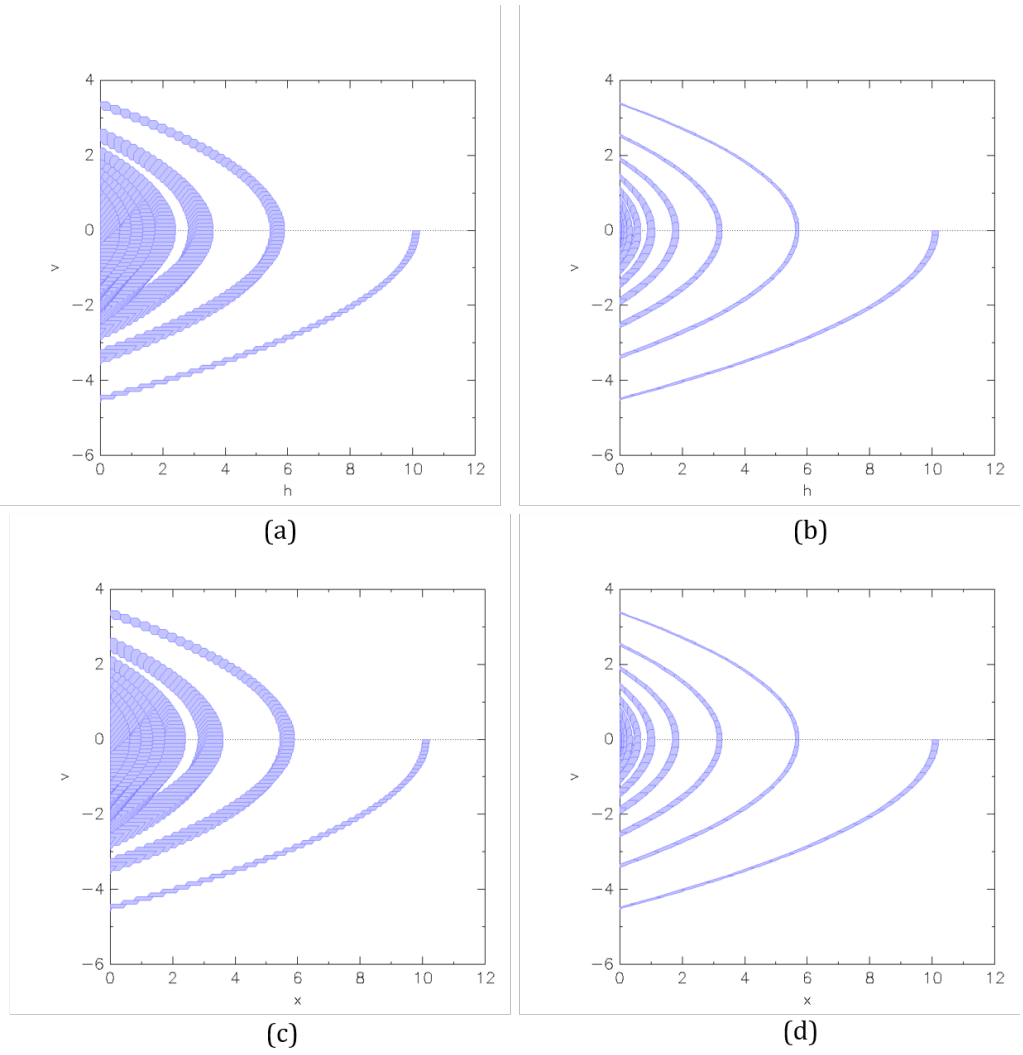

Figure 5.8: State spaces captured in SpaceEx. (a) Original model with LGG, (b) original model with STC, (c) translated model with LGG, and (d) translated model with STC.

5.3.5.2 Rail Gate Controller

The behavior of the well-known railroad gate control system [77] is studied here. A gate secures the railroad crossing. The system consists of three components: a train, a controller, and a gate. The train communicates with the controller, and the controller communicates with the gate as follows:

- Sensors send an "approach" signal to the controller when detecting the approaching train, sending an "exit" signal to the controller after the train left railroad crossing.
- The controller reacts to an "approach" signal by sending a "close" signal to the gate and an "exit" signal by sending an "open" signal to the gate, respectively.
- The gate reacts to a "close" signal with closing the gate and an "open" signal with opening the gate.

The three subsystems are modeled using HPrTNs shown in Figure 5.9. In the train subsystem, one continuous variable representing a train has an initial distance of 1200 meters. The speed of the train is assumed to be 30 meters per second. The crossing is assumed to be located between the location -100 and 100. The state of the gate is modeled using the angle it forms with the horizontal line. If the angle is 0, then it is assumed that the gate is closed. If the angle is neat $\pi/2$ then it is assumed that the gate is fully open.

The whole system is obtained by merging the common transitions from the three subsystems. Merging transitions achieve synchronization among subsystems. The constraint of a merged transition is the conjunction of those of the shared transition in the subsystems. Both synchronous subsystem composition, as shown in this example and asynchronous subsystem composition, can be nicely represented in HPrTNs, as shown in Figure 5.10, where the net inscription is omitted to keep

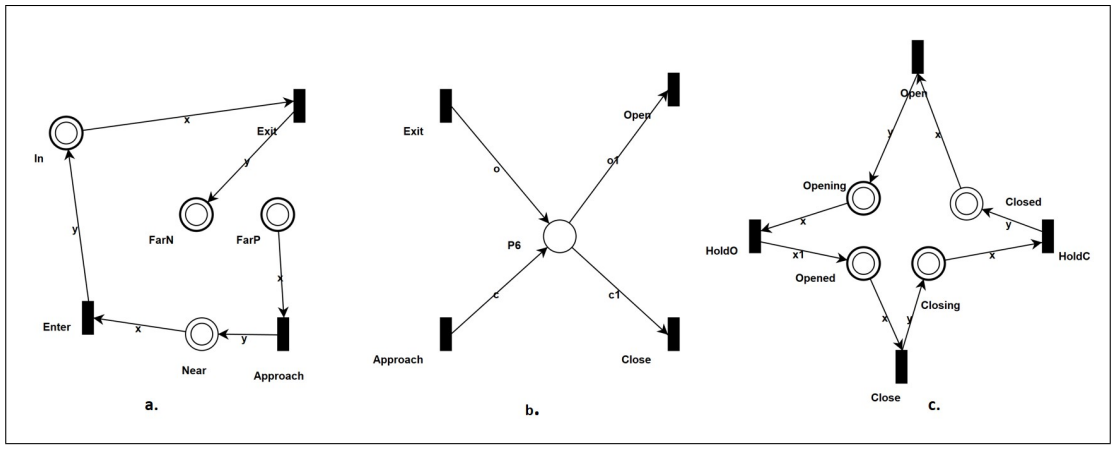

Figure 5.9: The RailGate system. a. The train subsystem, b. The controller subsystem, and c. The gate subsystem

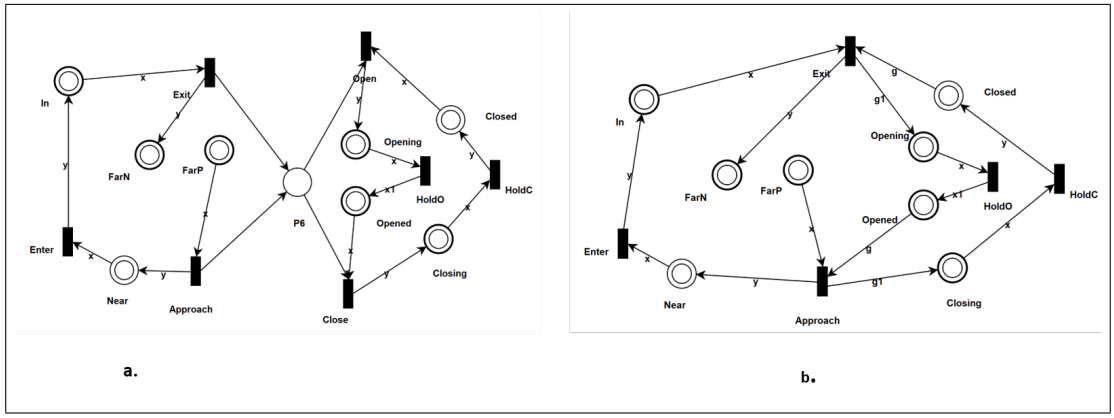

Figure 5.10: The composition of the railgate subsystems. a. Asynchronous, and b. Synchronous

the diagram more readable. In Figure 5.10(a), the controller and the train and the controller and the gate are composed synchronously. Nevertheless, the communication between the train and the gate is asynchronous. In Figure 5.10, synchronous communication is achieved. We will study the behavior of these two approaches.

The figure 5.11, shows the trajectory of the gate and the train against time. The train starts moving from 1200. As time passes by it, location is decreasing. Initially, the gate is completely open. At some point, the gate starts to close and then remains closed a certain amount of time and then starts to open again and

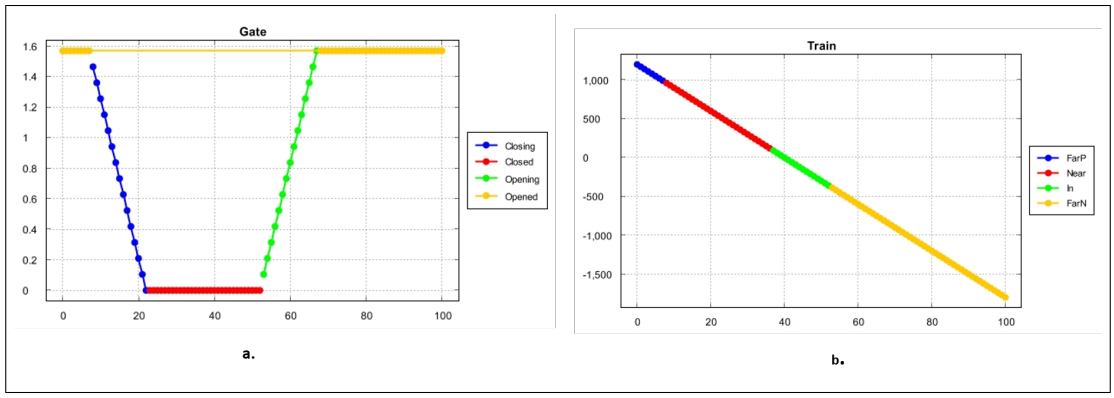

Figure 5.11: The simulation result of the railgate system in PIPE+. a. Trajectory of the Gate against time, and b. Trajectory of the Train against time

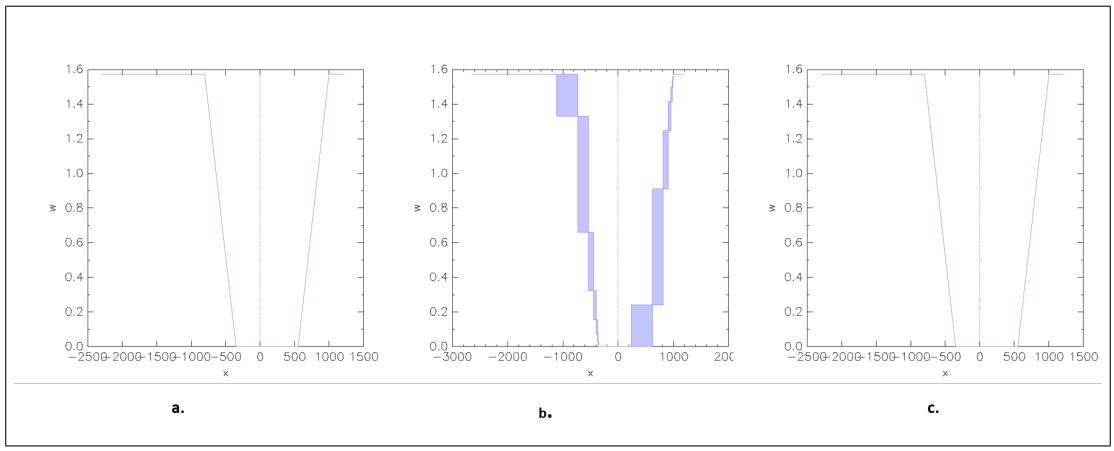

Figure 5.12: Analysis results of translated synchronous model in SpaceEx a. Simulation, b. LGG Scenerio, c. STC scenerio

eventually keeps open. The simulation results are similar for both of these two approaches of composition. These show that the models appear correct. Figure 5.11(a) shows that the train was in the crossing during the time between 35 and 50. At that time, the gate was closed entirely. Also, when the train is outside of the crossing, the gate was open. These models are translated to the sx format for reachability analysis with the tool SpaceEx.

Figure 5.12 shows the reachability analysis results of the synchronously composed model. Figure 5.12(b) and Figure 5.12(c) show the reachable states. The

Symbolic State	Time	Condition	Reachable
$-100 < x < 100 \& w > 0.01$	0.102	Unsafe	No
$x > 1000 \& w < 1.56$	0.095	Unsafe	No
$x = 1000 \& w < 1.56$	0.107	Unsafe	No
$x < -1000 \& w >= 1.56$	0.102	Safe	Yes
$130 < x < 1000 \& w <= 1.56$	0.076	Safe	Yes

Table 5.5: Statistics of the analysis summary of the synchronous model

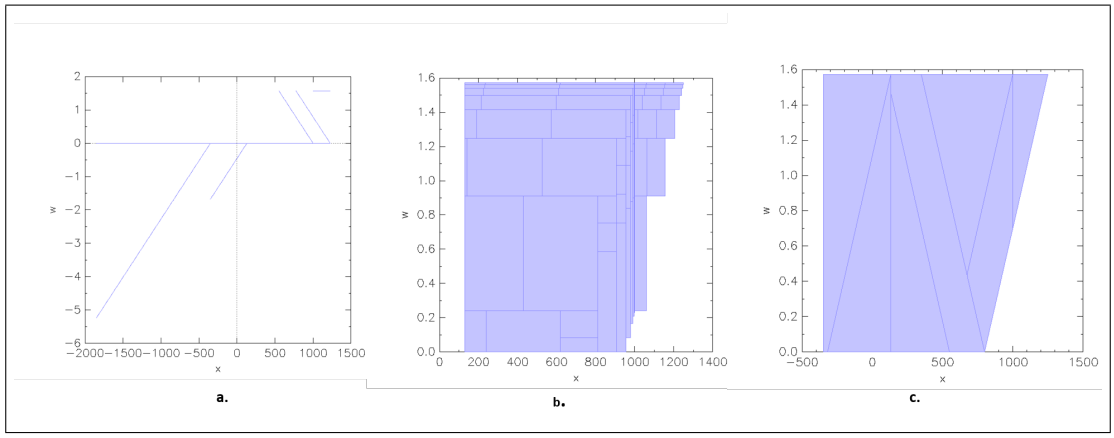

Figure 5.13: Analysis results of translated asynchronous model in SpaceEx a. Simulation, b. LGG Scenerio, c. STC scenerio

computed states are as expected since these states are safe. No unsafe states are reachable. Table 5.5 summarizes the results of reachability analysis of the synchronous model. Here, the analysis is done to see if some specific safe and unsafe states are reachable or not. The results show that unsafe states are not reachable. The reachable stats of the asynchronously composed model are shown in Figure 5.13. It clearly shows that the unsafe states could be reachable even when the system starts from safe states.

5.4 Related Work

Being infinite-state systems, explicit state model checking is not an appropriate method to verify hybrid systems. Reachability Analysis (also termed as *symbolic reachability analysis*) is one of the few techniques to verify hybrid systems. In this approach, the main goal is to decide whether a set of states reachable from the initial states of the system via all possible execution paths is safe. The set of safe states are pre-defined. The set of reachable states is computed iteratively from the initial states, and in each step, the reachable set is compared with the safe states. The key challenge in this approach is the appropriate representation of the states for the iterative reachability algorithm. There are several tools available for this approach. *HyTech* [81] was the first model checker to apply this technique with a specific representation of reachable set named *n-dimensional polyhedra*. Here, n is the number of the system's real-valued attributes, and each polyhedron is represented using a conjunction of linear inequalities over these variables. However, this approach is restricted to the class of *Linear Hybrid Automata* and, thus, is not scalable. It is not easy too. To deal with this problem, a strategy, *over-approximation* of the reachable set over polyhedral representation, was introduced by tool *Checkmate*[82] and refined later by *d/dt*[83]. Later several other approximation-based approaches, like *flowpipe approximation*, *convergent approximation*, etc. were proposed. These methods use over-approximative geometric and/or symbolic representations of states set, e.g., convex polytopes, zonotopes, ellipsoids, support functions, or Taylor models. The complexity of these methods is typically exponential in the number of dimensions. There are several other studies to deal with complexity and replacing polyhedral with alternative representations. So far, most scalable representation utilizes zonotopes and support functions in tool *SpaceEx*[12].

However, another prominent problem with the reachability analysis techniques is that they are limited to linear hybrid systems since those need to calculate the reachable sets. Deductive verification, on the other hand, does not exhibit this limitation. Here theorem proving is used to generate proofs of correctness of the systems. The tool *KeYmaera X*[84] provides support for deductive verification.

5.5 Summary

Simulation and reachability analysis are two established methods to analyze hybrid systems. In this chapter, both these two techniques to analyze hybrid systems modeled using HPrTN are discussed. The tool PIPE+ provides built-in support for simulating HPrTN models. For this, a new simulation algorithm is incorporated. New techniques are introduced to simulate the evolution of continuous states. New techniques are also introduced to visualize the results. As part of the support for reachability analysis, HPrTN models are translated to models suitable for use in the tool SpaceEx. A brief overview of the SpaceEx models and the available analysis techniques are discussed in this chapter. Then a complete step by step translation method is discussed. This translation method considers all the elements of HPrTN definition. During translation, all the elements in an HPrTN model are translated to the equivalent form. This chapter also discussed some techniques to automatic discovery of components in an HPrTN model and their composition in the translated model. In the end, analysis results of the translated models of two hybrid systems are discussed.

CHAPTER 6

REDESGNING PIPE+

PIPE+ is a Java-based desktop application for modeling and simulation using PrTN. It is built on top another application Platform Independent Petri net Editor (PIPE) [22]. PIPE is an open-source tool developed by Imperial College of London in 2003 for modeling and analyzing low-level Petri nets. Although PIPE is intended to manipulate low-level Petri nets, it provides some generic functionalities, which can be adapted for high-level Petri nets. PIPE+ utilizes those functionalities as a base and gradually introduced new functionalities for modeling and analyzing PrTNs.

Despite having excellent potential, PIPE has design and implementation flaws. PIPE+ automatically inherited those flaws. PIPE+ itself also introduced several technical debts, which urges on to a complete redesign and rebuild. In the following subsections, the limitations of the existing PIPE+ tools are discussed. Later an overview of the adopted architecture and some benefits of the redesigned PIPE+ are presented.

6.1 Limitations of PIPE+

The fundamental flaw with PIPE+ is its dependency on a system that is built using old technology and no upgrading with the latest development of that dependency. In this section, some of the identified flaws are elaborated.

6.1.1 Legacy Systems

PIPE+ uses several third party libraries. Many of these libraries are discontinued. The adaptation of PIPE into PIPE+ did not utilize appropriate design patterns

available for the base technology. Also, PIPE+ lacks proper documentation. This made it very difficult to upgrade. In the following subsections, these are elaborated from the viewpoint of the underlying technologies and dependencies

6.1.1.1 Technologies

PIPE+ adapted the version 2.0 of PIPE. This version was built using Java 1.4. Java itself has a considerable evolution from that version. Most of the Java API used in the implementation of PIPE+ has become obsolete, and there are better and improved replacements. However, due to the dependency on ancient technology, PIPE+ can not leverage those improvements.

6.1.1.2 Dependencies

Both PIPE and PIPE+ depend on other third-party libraries. Many of those libraries have reached their end of life (EOL). Many of these libraries are discontinued due to the availability of better and modern replacements. However, due to the fundamental limitations, it is complicated to utilize the latest development.

6.1.2 Quantitative Analysis

Several freely available profiling tools are used to measure the code quality of PIPE+ source code quantitatively. In the following subsection, some of these are categorized, and the analysis results are presented.

6.1.2.1 Static Analysis

Static analysis of the PIPE+ code was performed using an *IntelliJ Idea* plugin - *QAPlug* with *FindBugs* and *CheckStyle*. Running static analysis with default set-

tings revealed nearly 7000 violations of standards. Some of these violations are very critical and prone to cause bugs. Table 6.1 summarizes the result of the analyses.

Category	Issue	Violations
Efficiency (183)	Performance	160
	Others	23
Maintainability (1210)	Bad Practice	23
	Cyclometric Complexity	102
	Boolean Expression Complexity	87
	Inappropriate Modifier Usage	831
	Others	167
Reliability (3720)	Correctness	48
	Malicious Code Vulnaribility	72
	Others	3600
Usability (1862)	Dodgy	136
	Constants	105
	Naming	1097
	Hidden Field	208
	Others	316

Table 6.1: PIPE+ source code static analysis result

6.1.2.2 Circular Dependencies

When one module of a software system becomes dependent on another for functioning correctly, a dependency relation arises. When there are cycles in a dependency chain of two or more modules, then it is called circular dependency. It is quite harmful from the software engineering perspective. It creates tight coupling among components, are prone to cause bugs, and it has many more negative effects [85, 86].

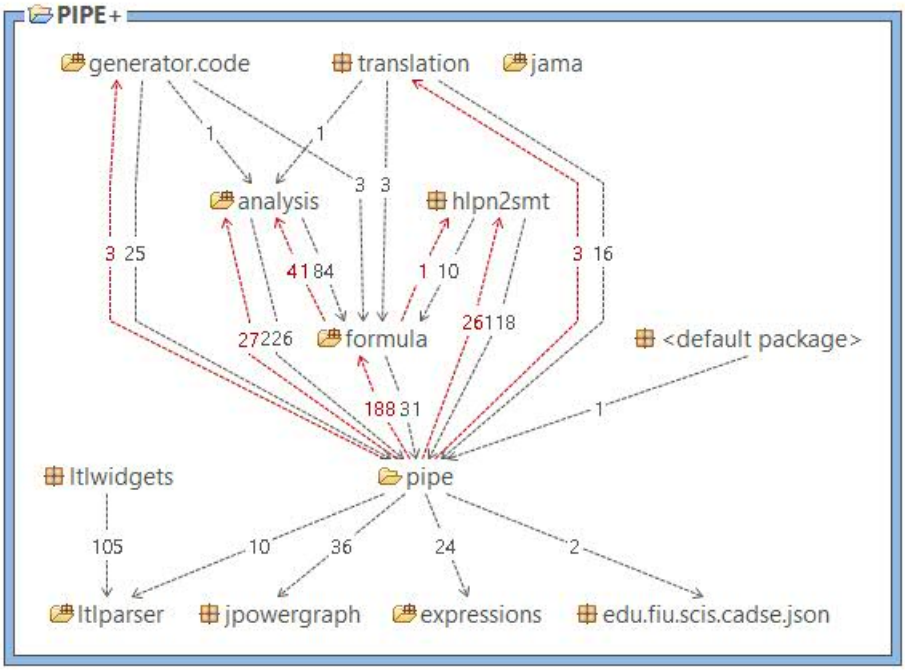

Figure 6.1: PIPE+ package-level dependency graph

Static analysis was performed using *Stan4J* on the PIPE+ source code to identify the standard package-level circular dependency. It reports the level of cyclic dependency in terms of *Tangledness* and *Average Component Dependency*. A higher value of these metrics indicates higher coupling among the components. Table 6.2 summarizes some of these metrics. Figure 6.2 and 6.1 shows the distance metrics among the packages of PIPE+ and the dependency graph. All of these metrics confirm the high circular dependency among PIPE+ packages.

Metric	**Index**
Tangled	24.39%
Average Component Dependency - Package	58.43%
Average Component Dependency - Unit	32.30%

Table 6.2: PIPE+ circular dependency test result

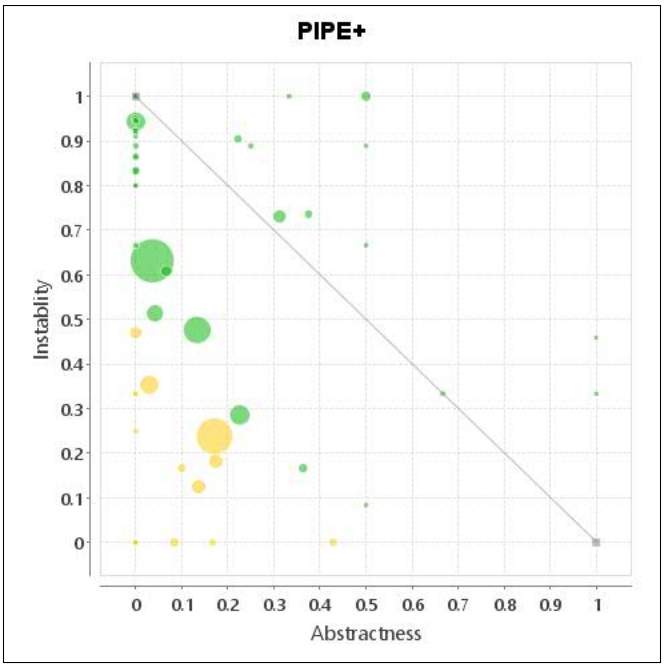

Figure 6.2: PIPE+ packages distance indices

6.1.2.3 Usability Issues

As reported in Table 6.1, the QAPlug bug finding tool, found 1862 usability issues including inconsistent naming, repeated usage of duplicated constant values, carelessly choosing local variable names hiding the member variables. Apart from these, there are 136 issues as categorized as Dodgy that must be avoided. Some of the most appeared issues under this category are declaring never used local variables, redundant null checks where the variables are non-null, exact floating point number equality checks, modifying static variables from instance method.

6.1.2.4 Portability Issues

There are several portability issues, including usage of thread safety classes where the objects are not shared between threads. For example, usage of *Vector* instead of other collection classes, usage of *StringBuffer* instead of *StringBuilder*, etc. Also,

primitive objects are instantiated by explicitly using the *new* keyword, whereas it is the best practice to use the appropriate utility methods.

6.1.2.5 Maintainability Issues

Table 6.1 also shows several severe maintainability issues. These include many bad practices, like having specialized methods without implementing the appropriate interface. For example, defining the *clone* method in a class without implementing *Cloneable* interface. Other violated maintainability issues are breaking the contract between equals and hashCode methods, Serializable class without serialVersionID, comparing string literals using '==' operator, invoking System.exit method, suspicious reference checks.

6.1.3 Qualitative Analysis

6.1.3.1 Functional Ambiguity

The PIPE tool was developed primarily for low-level Petri nets. While adapting it, PIPE+ deliberately inserted its required functionalities. As a result, functionalities for both low-level and high-level Petri nets co-exist without appropriate distinction. This confuses even to experienced users. Sometimes, it becomes tough to differentiate between high-level and low-level Petri net functionalities.

6.1.3.2 Usability Issues

The tool is not very intuitive. A user needs to follow some specific steps to model a system and analyze it properly. Moreover, there is no proper documentation describing these steps, which makes it very hard for the new users.

6.1.3.3 Performance Issues

Several functionalities are implemented in an unnecessarily complicated way. In many cases, proper measures are not taken to simplify implementation for improved performances. For example, the formula parser used to parse the transition constraints creates very tall syntax trees. If the grammar or the parser generator is optimized correctly, then the depth of the new syntax tree would be the half. In that case, the evaluation of the formula would take half of the time as it required earlier. The simulator has another severe performance issue related to formula parsing. The formula associated with a transition was parsed each time the transition is selected for evaluation. However, a formula is immutable during the lifetime of a simulation run. So technically, a formula could be parsed only once in a simulation run using memoization. This can be further optimized with proper caching and cache invalidation techniques. There are many other issues where optimation could be possible.

6.1.3.4 Lack of Concurrency

Another severe flaw with the implementation of PIPE+ is that it does not offer concurrently. Since the tool's usage is highly interactive, in most cases, the user does not experience any difficulty in using it. But this problem becomes apparent during the multi-step simulation. In a multi-step simulation, the user configures the simulator to run for a given number of steps. The user then expects to observe gradual evolution in the chart at the end of each step. The current implementation cannot do that. It just runs all the steps at once in the **foreground** and shows the charts after all the steps are done. During the whole period of time, the UI is unavailable to the user. So, if the user wants to cancel in the middle, he cannot do that as well.

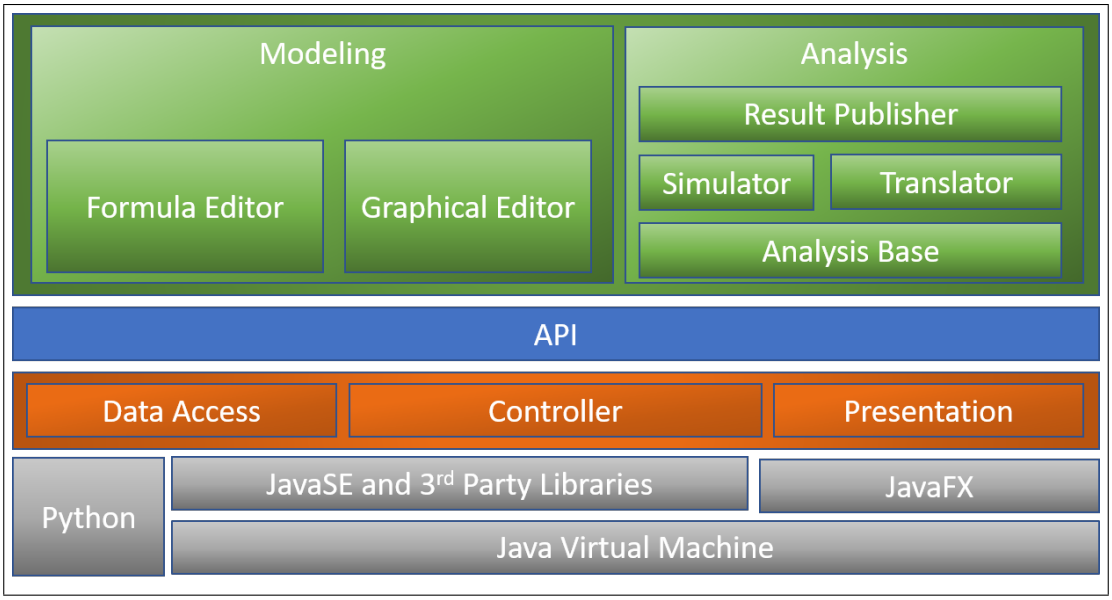

Figure 6.3: High-level overview of the architecture of PIPE+ Redesigned

6.2 Redesign

6.2.1 Architecture

PIPE+ Redesigned is a Java-based desktop application implemented following a multi-layered architecture. Figure 6.3 shows a high-level overview of the architecture of the redesigned tool. The core layer of this tool consists of three modules - the data access module, the controller module, and the presentation module. The data access module and the controller module are built using JavaSE and some third-party open-source libraries. The presentation module is built using JavaFX. A comprehensive API layer is built on top of the core layer and provides user-facing features and functionalities implemented in the application layer. Figure 6.3 shows two partitions of the application layer for two broad categories of functionalities provided by the tool - modeling and analysis. These layers are designed to be highly configurable and extendable.

6.2.1.1 Data Access Module

The Petri net data model is the necessary data structure to represent a Petri net model. This module provides functionalities to create, list, modify and delete the Petri net model elements. This module also provides the serialization techniques and transformation to other forms, such as file storing and retrieving methods. This module is designed to provide a generic way to represent Petri net models and provides a concrete implementation of HPrTN.

Configurations provide parameters to customize the environment for model analysis. For example, the simulation environment needs an initial seed for random number generation, initial logical timestamp, step size, number of steps. This information is stored when the simulation mode is chosen. The user does not need to provide the same information multiple times. These configurations are also considered as a part of the domain of the tool. The data access module facilitates manipulations of these configurations.

6.2.1.2 Controller Module

This module works as a mediator between the data access module and the presentation module and serves as a base for other layers. This module provides application-wide generic functionalities such as managing and handling system-generated events, as well as (a) domain objects manipulation, (b) structural and semantic consistency assurance, (c) static analysis and insights generation for result visualization, and (d) concurrency management.

6.2.1.3 Presentation Module

This module provides the foundation to build the GUI of the tool. The functionalities provided by this module include (a) the overall layouts of the application

windows, (b) application-wide components like menus and menu bars, tools and toolbars, status bar, etc, (c) composite and generic UI elements for manipulating Petri net elements and displaying analysis results, (d) UI related events definition, (e) uniform methods to publish and register events, and (f) the coherent way of event propagation.

6.2.1.4 API Layer

The API layer provides uniform high-level access to the functionalities provided by the core layer. This layer helps reduce the learning curve for other developers to adapt and extend PIPE+ Redesigned.

6.2.1.5 Application Layer

The application layer contains components to implement the user-facing features of the tool. This layer makes use of the API layer's functionalities to provide the solutions to the user requirements. The application layer captures the user action and delegates to the appropriate function in the API. To provide the invocation point, a GUI can be implemented utilizing the presentation module's functionalities via the API-Presentation component of the API layer.

PIPE+ Re-designed implements several components in the application layer to create and analyze HPrTN models, as shown in Fig.6.3. Graphical Editor and Formula Editor provide functionalities to create and modify hybrid predicate transition net models. The Simulator, Translator, and Result Publisher are components to support model analysis.

The component-based layer architecture, along with the publisher-consumer pattern of communication among the components, gives us the flexibility to reduce the

coupling among the components. This encourages the independent evolution of different components.

6.2.2 Implementation

6.2.2.1 Latest Technology Stack

The redesigned PIPE+ uses Java 8 as the base technology. All the dependent libraries are upgraded as well to make them compatible with Java 8. Also, some libraries are replaced with different ones as suggested by the developer community, to comply with the state of the norm.

6.2.2.2 Code Quality

To maintain the code quality, the static analyzers are run frequently to observe that the number of violations is in control. Mainly it is ensured that there are no critical violations. Apart from the rigorous checking during implementation, some guidelines and code styles are identified. These will be published with the codebase when it will be made available for others to collaborate on.

6.2.2.3 Functional Quality

All the identified performance bottlenecks in the previous version of PIPE+ are addressed in the redesigned PIPE+. Several new strategies are incorporated, and new algorithms are introduced for the simulation engine. Some of them are already discussed in chapter 3.

6.2.3 Build/Release Process

Previously, the only way to distribute the tool was by sharing the source code. As part of the redesigning process, a proper build and release process is also established. Along with source code, generated binary also will be distributed for others to try.

The new tool uses the *Maven* build tool. Maven is one of the most used build process orchestration tools for Java projects. Almost 80% of all Java projects in the world are being built using Maven. The source code is hosted on GitHub. GitHub *GitFlow* will be followed to generate the releases. Maven will produce the release build, and GitHub will host and announce the releases. The source code of this project can be found at https://github.com/dalam004/pipeplus.

CHAPTER 7

CONCLUSION

This thesis focuses on a specific approach to model and analyze discrete event systems, continuous systems, hybrid systems, and to provide a basis for modeling and analyzing cyber-physical systems.

The first contribution is enhancing the modeling capability of discrete event systems using PrTNs. The algebraic specification of PrTNs are fully realized in PIPE+ tool, which supports many new functionalities including full-first order logic formulas with quantifiers and set operations, the new type of real numbers, new mathematical functions, and a new clock variable. Model composition is also supported in PIPE+ to build larger systems by reusing existing models.

The second contribution is providing a more robust translation-based model checking technique. A significant contribution is to make the model translation more functionally complete. Several other alternative ways of model translation are also provided. Additionally, the simulation environment in PIPE+ tool is improved significantly, including different modes of stimulation, internal simulation result visualization, and exporting results for external analysis.

The third contribution is a new definition of HPrTNs, where continuous places are introduced to capture continuous states. Tokens in continuous places represent continuous states, and their evolution is defined by differential equations. Bounds are introduced to specify the invariant of continuous variables. Discrete transitions are used to change modes of continuous variables. HPrTNs are fully realized in PIPE+, including support for utilizing both static (syntactic) and dynamic semantics to model a wide range of dynamical systems.

The final contribution is providing analysis techniques for hybrid systems. Two complementary analysis methods are provided. Simulation based on the dynamic

semantics of HPrTNs is used to analyze non-linear hybrid systems, and is implemented in PIPE+. Reachability analysis is used to analyze piece-wise linear hybrid systems and is implemented through a translator in PIPE+ and leveraging the state of the art external tool SpaceEx.

Additional improvements can be done, including a better scheduler to handle discrete state transitions and continuous state evolution seamlessly, more options for model translation, and the completion of all the functionality in the proposed redesign of PIPE+.

BIBLIOGRAPHY

[1] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Heidelberg, Berlin: Springer-Verlag, 2010.

[2] P. J. Antsaklis, "Special issue on hybrid systems: theory and applications a brief introduction to the theory and applications of hybrid systems," *Proceedings of the IEEE*, vol. 88, pp. 879–887, July 2000.

[3] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pp. 278–292, 1996.

[4] Hassane Alla and René David, "A modelling and analysis tool for discrete events systems: continuous petri net," *Performance Evaluation*, vol. 33, no. 3, pp. 175 – 199, 1998.

[5] R. David and H. Alla, *Discrete, Continuous, and Hybrid Petri Nets*. Springer Berlin Heidelberg, Springer, 2010.

[6] H. J. Genrich and K. Lautenbach, "System modelling with high-level petri nets," *Theor. Comput. Sci.*, vol. 13, pp. 109–136, 1981.

[7] K. Jensen, "Coloured petri nets: A high level language for system design and analysis," in *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]* (G. Rozenberg, ed.), vol. 483 of *Lecture Notes in Computer Science*, pp. 342–416, Springer, 1989.

[8] W. Reisig, "Petri nets and algebraic specifications," *Theor. Comput. Sci.*, vol. 80, no. 1, pp. 1–34, 1991.

[9] M. Herajy, F. Liu, and C. Rohr, "Coloured hybrid petri nets for systems biology," in *Proceedings of the 5th International Workshop on Biological Processes & Petri Nets*, pp. 60–76, 2014.

[10] D. Bera, K. M. van Hee, and H. Nijmeijer, "Modeling hybrid systems with petri nets," in *Simulation and Modeling Methodologies, Technologies and Applications - International Conference, SIMULTECH 2014 Vienna, Austria, August 28-30, 2014 Revisaled Selected Papers*, pp. 17–42, 2014.

[11] R. Wieting, "Modeling and simulation of hybrid systems using hybrid high-level nets," in *Proceedings of the 8th European Simulation Symposium (ESS'96)*, vol. II, (Genoa, Italy), pp. 158–162, October 1996.

[12] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *CAV*, vol. 6806 of *Lecture Notes in Computer Science*, pp. 379–395, Springer, 2011.

[13] S. Liu, R. Zeng, and X. He, "Pipe+ – a modeling tool for high level petri nets," in *Proc. of International Conference on Software Engineering and Knowledge Engineering (SEKE11)*, (Miami), pp. 115–121, 2011.

[14] S. Liu, Z. Sun, R. Zeng, and X. He, "Samat - a tool for software architecture modeling and analysis," in *Proc. of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE12)*, (San Francisco, CA), pp. 352–358, 2012.

[15] L. Chang and X. He, "A methodology to analyze multi-agent systems modeled in high level petri nets," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 7, pp. 1199–1235, 2015.

[16] X. He, "A formal definition of hierarchical predicate transition nets," in *Application and Theory of Petri Nets 1996, 17th International Conference, Osaka, Japan, June 24-28, 1996, Proceedings* (J. Billington and W. Reisig, eds.), vol. 1091 of *Lecture Notes in Computer Science*, pp. 212–229, Springer, 1996.

[17] D. M. M. Alam and X. He, "A method to analyze high level petri nets using SPIN model checker," in *The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017* (X. He, ed.), pp. 161–166, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2017.

[18] D. M. M. Alam and X. He, "A method to analyze predicate transition nets using SPIN model checker," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 9-10, pp. 1455–1482, 2017.

[19] D. M. M. Alam, X. He, and W. C. Chu, "Modeling and analyzing hybrid systems using hybrid predicate transition nets (S)," in *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018* (Ó. M. Pereira, ed.),

pp. 397–396, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018.

[20] X. He and D. M. M. Alam, "Hybrid predicate transition nets - A formal method for modeling and analyzing cyber-physical systems," in *19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*, pp. 216–227, IEEE, 2019.

[21] S. Liu and X. He, "Pipe+verifier - A tool for analyzing high level petri nets," in *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015* (H. Xu, ed.), pp. 575–580, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015.

[22] P. Bonet, C. M. Llado, and R. Puigjaner, "Pipe v2.5: a petri net tool for performance modeling," in *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, (San Jose, Costa Rica), October 2007.

[23] I. F. D. L. Mota, A. Guasch, M. M. Mota, and M. A. Piera, *Robust Modelling and Simulation*. Springer, Cham, 2017.

[24] H. J. Genrich and K. Lautenbach, "The analysis of distributed systems by means of predicate ? transition-nets," in *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979* (G. Kahn, ed.), vol. 70 of *Lecture Notes in Computer Science*, pp. 123–147, Springer, 1979.

[25] H. J. Genrich, "Predicate/transition nets," in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986* (W. Brauer, W. Reisig, and G. Rozenberg, eds.), vol. 254 of *Lecture Notes in Computer Science*, pp. 207–247, Springer, 1986.

[26] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, and K. Wolf, "Complete Results for the 2015 Edition of the Model Checking Contest." http://mcc.lip6.fr/2015/results.php, 2015.

[27] P. M. Merlin and D. J. Farber, "Recoverability of communication protocols," *IEEE Trans. Communications*, vol. 24, no. 4, pp. 1036–1043, 1976.

[28] S. M. C. Ghezzi, D. Mandrioli and M. Pezzi, "A unified high-level petri net formalism for time-critical systems," *IEEE Trans. Software Engineering*, vol. 17, no. 2, pp. 160–172, 1991.

[29] X. He, "Modeling and analyzing cyber physical systems using high level petri nets," in *Proc. 18th IEEE International Conference on Software Quality, Reliability, and Security*, (Lisbon, Portugal), July 2018.

[30] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[31] X. He, "A comprehensive survey of petri net modeling in software engineering," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 5, pp. 589–626, 2013.

[32] *CPNTool*, 2020 (accessed July 14, 2020). `http://cpntools.org/`.

[33] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[34] S. Liu, R. Zeng, Z. Sun, and X. He, "Bounded model checking high level petri nets in pipe+verifier," in *Proc. of International Conference on Formal Engineering Methods (ICFEM 14)*, vol. 8829 of *LNCS*, (Luxembourg), pp. 348–363, 2014.

[35] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[36] G. Argote-Garcia, P. J. Clarke, X. He, Y. Fu, and L. Shi, "A formal approach for translating a SAM architecture to PROMELA," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 3, 2008*, pp. 440–447, 2008.

[37] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Berlin, Heidelberg: Springer-Verlag, 2007.

[38] X. He, R. Zeng, S. Liu, Z. Sun, and K. Bae, "A term rewriting approach to analyze high level petri nets," in *Proc. of the 10th Theoretical Aspects of Software Engineering Conference (TASE 16)*, (Shanghai, China), July 2016.

[39] R. Gerth, *Concise Promela Reference*, 1997 (accessed July 14, 2020). `http://spinroot.com/spin/Man/Quick.html`.

[40] "SPIN – Success Stories." http://spinroot.com/spin/what.html, 2013.

[41] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pp. 46–57, IEEE Computer Society, 1977.

[42] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, "Formally analyzing software architectural specifications using SAM," *Journal of Systems and Software*, vol. 71, no. 1-2, pp. 11–29, 2004.

[43] S. Rajan, N. Shankar, and M. K. Srivas, "An integration of model checking with automated proof checking," in *Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings* (P. Wolper, ed.), vol. 939 of *Lecture Notes in Computer Science*, pp. 84–97, Springer, 1995.

[44] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings* (D. Kapur, ed.), vol. 607 of *Lecture Notes in Computer Science*, pp. 748–752, Springer, 1992.

[45] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[46] K. L. McMillan, *Symbolic model checking*. Kluwer, 1993.

[47] M. Baldamus and J. Schröder-Babo, "p2b: A translation utility for linking promela and symbolic model checking (tool paper)," in *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings* (M. B. Dwyer, ed.), vol. 2057 of *Lecture Notes in Computer Science*, pp. 183–191, Springer, 2001.

[48] O. Grumberg and S. Katz, "Veritech: a framework for translating among model description notations," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 2, pp. 119–132, 2007.

[49] C. Schröter, S. Schwoon, and J. Esparza, "The model-checking kit," in *Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings* (W. M. P. van der Aalst and E. Best, eds.), vol. 2679 of *Lecture Notes in Computer Science*, pp. 463–472, Springer, 2003.

[50] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods Syst. Des.*, vol. 9, no. 1/2, pp. 41–75, 1996.

[51] Z. Manna, N. Bjørner, A. Browne, E. Y. Chang, M. Colón, L. de Alfaro, H. Devarajan, A. Kapur, J. Lee, H. Sipma, and T. E. Uribe, "Step: The stanford temporal prover," in *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings* (P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds.), vol. 915 of *Lecture Notes in Computer Science*, pp. 793–794, Springer, 1995.

[52] K. Havelund, "Java pathfinder, A translator from java to promela," in *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings* (D. Dams, R. Gerth, S. Leue, and M. Massink, eds.), vol. 1680 of *Lecture Notes in Computer Science*, p. 152, Springer, 1999.

[53] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000* (C. Ghezzi, M. Jazayeri, and A. L. Wolf, eds.), pp. 439–448, ACM, 2000.

[54] K. Jiang and B. Jonsson, "Using spin to model check concurrent algorithms, using a translation from c to promela," in *2nd Swedish Workshop on Multi-Core Computing, Uppsala, Sweden: Department of Information Technology, Uppsala University*, pp. 67–60, 2009.

[55] G. J. Holzmann, "Logic verification of ANSI-C code with SPIN," in *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings* (K. Havelund, J. Penix, and W. Visser, eds.), vol. 1885 of *Lecture Notes in Computer Science*, pp. 131–147, Springer, 2000.

[56] A. Zaks and R. Joshi, "Verifying multi-threaded C programs with SPIN," in *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings* (K. Havelund, R. Majumdar, and

J. Palsberg, eds.), vol. 5156 of *Lecture Notes in Computer Science*, pp. 325–342, Springer, 2008.

[57] G. C. Gannod and S. Gupta, "An automated tool for analyzing petri nets using SPIN," in *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pp. 404–407, IEEE Computer Society, 2001.

[58] P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.

[59] C. L. Talcott, "Cyber-physical systems and events," in *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions* (M. Wirsing, J. Banâtre, M. M. Hölzl, and A. Rauschmayer, eds.), vol. 5380 of *Lecture Notes in Computer Science*, pp. 101–115, Springer, 2008.

[60] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2015.

[61] R. David and H. Alla, *Discrete, Continuous, and Hybrid Petri Nets*. Berlin, Heidelberg: Springer, 2010.

[62] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[63] R. Alur, *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[64] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*, pp. 209–229, 1992.

[65] J. Eker, J. W. Janneck, E. A. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127–144, Jan 2003.

[66] A. Fehnker, F. W. Vaandrager, and M. Zhang, "Modeling and verifying a lego car using hybrid I/O automata," in *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, pp. 280–289, 2003.

[67] A. Platzer, "Differential dynamic logic for hybrid systems," *J. Autom. Reasoning*, vol. 41, no. 2, pp. 143–189, 2008.

[68] R. David and H. Alla, "Continuous petri nets," in *8th European Workshop on Application and Theory of Petri Nets*, (Zaragoza, Spain), 1987.

[69] R. David and H. Alla, "On hybrid petri nets," *Discrete Event Dynamic Systems*, vol. 11, no. 1-2, pp. 9–40, 2001.

[70] I. Demongodin and N. T. Koussoulas, "Differential petri nets: representing continuous systems in a discrete-event world," *IEEE Transactions on Automatic Control*, vol. 43, pp. 573–579, April 1998.

[71] K. S. Trivedi and V. G. Kulkarni, "Fspns: Fluid stochastic petri nets," in *Application and Theory of Petri Nets*, vol. 691 of *Lecture Notes in Computer Science*, pp. 24–31, Springer, 1993.

[72] M. Heiner, M. Herajy, F. Liu, C. Rohr, and M. Schwarick, "Snoopy - A unifying petri net tool," in *Petri Nets*, vol. 7347 of *Lecture Notes in Computer Science*, pp. 398–407, Springer, 2012.

[73] F. Sessego, A. Giua, and C. Seatzu, "Simulation and analysis of hybrid petri nets using the matlab tool hypens," *IEEE Int. Conf. on Systems, Man, and Cybernetics*, October 2008.

[74] J. Júlvez, C. Mahulea, and C. R. Vázquez, "Simhpn: A matlab toolbox for simulation, analysis, and design with hybrid petri nets," *Nonlinear Analysis: Hybrid Systems*, vol. 6, pp. 806–817, March 2012.

[75] A. Amengual, "A specification of a hybrid petri net semantics for the hisim simulator." http://www.icsi.berkeley.edu/pubs/techreports/TR-09-003.pdf, 2009.

[76] M. Herajy, F. Liu, C. Rohr, and M. Heiner, "Snoopy's hybrid simulator: a tool to construct and simulate hybrid biological models," *BMC Systems Biology*, vol. 11, July 2017.

[77] T. A. Henzinger, "The theory of hybrid automata," in *Verification of Digital and Hybrid Systems.* (I. M.K. and K. R.P., eds.), vol. 170 of *NATO ASI Series (Series F: Computer and Systems Sciences)*, (Berlin, Heidelberg), Springer, 2000.

[78] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[79] C. L. Guernic and A. Girard, "Reachability analysis of hybrid systems using support functions," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings* (A. Bouajjani and O. Maler, eds.), vol. 5643 of *Lecture Notes in Computer Science*, pp. 540–554, Springer, 2009.

[80] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech," in *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings* (M. Morari and L. Thiele, eds.), vol. 3414 of *Lecture Notes in Computer Science*, pp. 258–273, Springer, 2005.

[81] T. A. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," in *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pp. 460–463, 1997.

[82] A. Chutinan and B. H. Krogh, "Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations," in *HSCC*, vol. 1569 of *Lecture Notes in Computer Science*, pp. 76–90, Springer, 1999.

[83] E. Asarin, T. Dang, O. Maler, and O. Bournez, "Approximate reachability analysis of piecewise-linear dynamical systems," in *HSCC*, vol. 1790 of *Lecture Notes in Computer Science*, pp. 20–31, Springer, 2000.

[84] N. Fulton, S. Mitsch, J. Quesel, M. Völp, and A. Platzer, "Keymaera X: an axiomatic tactical theorem prover for hybrid systems," in *CADE*, vol. 9195 of *Lecture Notes in Computer Science*, pp. 527–538, Springer, 2015.

[85] "Circular dependency - Wikipedia." https://en.wikipedia.org/wiki/Circular_dependency, 2020.

[86] " Why are circular references considered harmful? - Stack Overflow." https://stackoverflow.com/questions/1897537/why-are-circular-references-considered-harmful, 2018.

VITA

DEWAN MOHAMMAD MOKSEDUL ALAM

2003-2008	B.S. in Computer Science Bangladesh University of Technology Dhaka, Bangladesh
2008-2009	Software Engineer AftiGIS Bangladesh Dhaka, Bangladesh
2009-2014	Senior Software Engineer Escenic AS Dhaka, Bangladesh
2014-2015	Senior Software Engineer Cefalo AS Dhaka, Bangladesh
2015-2017	Research Assistant Florida International University Miami, Florida
2018-2019	Teaching Assistant Florida International University Miami, Florida
2018-2020	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS

D. M. M. Alam and X. He, *"A method to analyze high level Petri nets using spin model checker,"* in Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering, pp. 161-–166, July 2017.

D. M. M. Alam and X. He, *"A method to analyze predicate transition nets using SPIN model checker,"* International Journal of Software Engineering and Knowledge Engineering, vol. 27, no. 9–10, pp. 1455-–1482, 2017.

D. M. M. Alam, X. He, and W. C. Chu, *"Modeling and analyzing hybridsystems using hybrid predicate transition nets (S),"* in The 30th International Conference on Software Engineering and Knowledge Engineering, (Redwood City, California, USA.), pp. 397—396, July 2018.

X. He and D. M. M. Alam, *"Hybrid predicate transition nets - A formal methodfor modeling and analyzing cyber-physical systems,"* in 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22–26, 2019, pp. 216—227, 2019.