PIPE+ - A Modeling Tool for High Level Petri Nets

Su Liu, Reng Zeng, Xudong He
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
{sliu002, rzeng001, hex}@cs.fiu.edu

***Abstract*— Petri nets are a formal, graphical and executable modeling technique for the specification and analysis of concurrent systems and have been widely applied in computer science and many other engineering disciplines. Low level Petri nets are simple and useful for modeling control flows; however, they are not powerful to define data and system functionality. High level Petri nets were developed to support data and functionality definitions [1]. To support the practical applications of Petri nets formalism, tools for designing and executing Petri nets are necessary. Although there are many existing tools for supporting low level Petri nets [5], few tools are available for high level Petri nets. There is especially a lack of tools to support high level Petri net notation proposed in the international standard [1]. In this paper, we present a tool, called PIPE+*, to support a subset of high level Petri nets proposed in [1]. PIPE+ is built upon an existing low level Petri net tool PIPE (Platform Independent Petri Net Editor) [2]. This paper describes the functionality of PIPE+ as well as illustrates the process of extending PIPE, which provides helpful insights for others to create Petri net tools suit their own needs. Furthermore, PIPE+ is an open source tool and thus is available for various enhancements from worldwide research community.**

Keywords: Petri Net; Modeling Tools

I. INTRODUCTION

Petri nets have been used to describe a wide range of systems since their invention in 1962 [3]. They are a graphical and formal method for describing and studying concurrent and distributed systems [4]. Low level Petri nets are suitable to model control flows and are applicable to work flow systems; however are not adequate to describe complex systems. High level Petri nets were developed to support the definition of data and functional processing [1]. To support the practical applications of Petri net formalism, tools for creating and executing Petri nets are needed. In [5], a Petri net tool database listed the tools developed in the past several decades. Unfortunately, many of the tools described in the database as well as in literature are no longer maintained or available and few of them support high level Petri nets, especially the high level Petri net definitions and notations proposed in the 2001 international standard [1]. Table 1 lists representative tools supporting some forms of high level Petri nets.

The high level Petri nets proposed in the international standards draws concepts from predicate transition nets, colored Petri nets, and algebraic Petri nets; and provides abstract and general definitions and notions. It is desirable to create a tool for editing and simulating the high level Petri nets; unfortunately no such tool exists yet. A realistic attempt is to develop a tool to support a restricted and concrete realization of high level Petri nets. The most critical components of the high level Petri net definitions are the net annotations with regard to transitions, which are algebraic terms of Boolean type. In [15], we viewed first order logic formulas as the algebraic terms associated with transitions and thus adopted the predicate transition net view as a concrete realization of high level Petri nets.

Name	High level Net Type	Graphical Editor	Simulator
AlPiNA	Algebraic Petri Nets	Yes	No
CoopnBuilder	CO-OPN language	Yes	No
CPN Tools	Colored Petri Nets	Yes	Yes
HISIm	Hybrid Petri Nets	Yes	Yes
Renew	Object Oriented Petri nets	Yes	Yes
PIPE+	High level Petri nets	Yes	Yes

Table 1 A List of High Level Petri net Tools

It requires tremendous effort to build a high level Petri net tool from scratch and is especially difficult to assure the quality of the initial design. It is desirable to leverage successful results and extend mature and existing tools. It is of tremendous value to build upon open source tools so that the resulting new tool can be shared and improved by the worldwide research community. PIPE+* is a tool to support high level Petri nets where transition conditions are defined in terms of first order logic formulas [14]. PIPE+ extends a well developed open source low level Petri net tool called PIPE (Platform Independent Petri net Editor) [6]. Furthermore PIPE+ retains the original low level Petri net editing and executing features, which allows user to choose appropriate net levels to model target systems.

In this paper, we first briefly review the background of basic and high level Petri net concepts. We introduce the chosen low level Petri net tool PIPE that PIPE+ built on, and then present implementation details of the extension according to the high level Petri net concepts. We discuss limitations and applications of the tool, and our contributions and perspectives.

* PIPE+ can be downloaded at http://users.cis.fiu.edu/~sliu002/

II. PETRI NETS AND HIGH LEVEL PETRI NETS

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

According to the international standard [1], a high level Petri net graph comprises: a net graph, place types, place marking, arc annotations, transition condition and declarations. The net graph is the net structure; place types are non-empty sets, restrict the data structure of tokens in the place; place markings are collection of elements (data items) associated with places, called tokens; arc annotations are inscribed with expressions which may comprise constants variables (e.g., x, y) and function images (e.g., f(x)); transition conditions are Boolean expressions inscribed in; declarations comprising definitions of place types, typing of variables and function definitions. For net execution, the most important is transition enabling. Enabling a transition involves the marking of its input places. When an enabled transition occurs, the enabling tokens from input place's are subtracted and the resulting tokens of the transition Boolean expression are added to the output places.

III. AN OVERVIEW OF PIPE

PIPE [2] is a Platform Independent Petri net Editor to edit, animate and analyze low level Petri nets, which has clear design and incorporates the latest XML Petri net standards of storing format, the Petri Net Markup Language (PNML). It is implemented in Java and can be logically divided into three major components [6], shown in Figure 1: the graphical user interface (GUI), a layer managing the interactions between the GUI and the modules (DataLayer), and analysis modules.

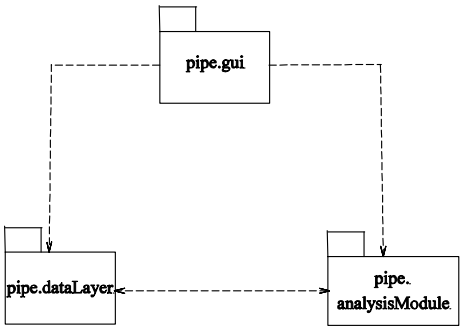

Figure 1 Package Diagram for PIPE

A. Graphical User Interface

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

- ***Editor***: Users are able to edit a low level Petri net by clicking and drawing Petri net graphical elements through the menu bar, toolbar. On the toolbar, it lists all the Petri net element thumbnails, such as place, transition and arc, which can be selected and added to the white canvas (tabbed pane) of the editor. Besides, these added elements' annotations and attributes can be defined by selecting one of the elements and pop up an editing dialog box.
- ***Simulator***: There is a switcher button between editor mode and simulation mode. Using the simulator, a user is able to fire a random transition or fire a number of transitions randomly selected among enabled ones. The simulation process includes subtracting tokens from input places and adding them to output places while firing a transition. Besides, the animation history is displayed on the left bottom of the interface frame by listing transition's label orderly.

B. Internal Architecture of PIPE—The DataLayer

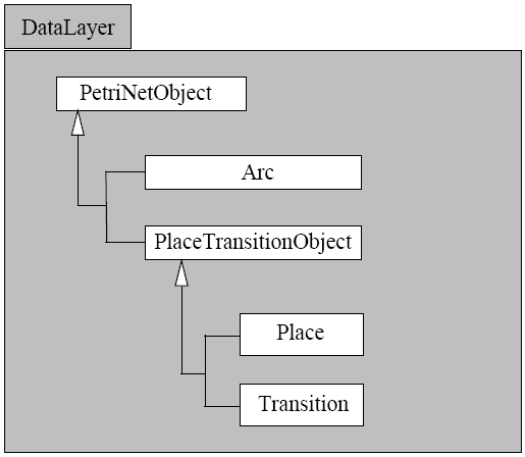

Figure 2 The Hierarchy of PetriNetOjbect Classes

The core component of PIPE is the data layer, which maintains states and contains all the classes used to represent a Petri net. Figure 2 shows the hierarchy of important Petri net object classes [6], including Arc, Place and Transition classes inherited from PetriNetObject because they have common variables and methods, such as id, name, location, etc.

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

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

C. Saving and Loading

PIPE is capable of saving and loading nets and writing the Petri net data layer into a Petri net Markup Language (PNML). An Extensible Stylesheet Language Transformation (XSLT) is used to transform it between PNML and XML files.

IV. PIPE+

A. Overview of the Extension

Similar to PIPE, PIPE+ is also an editor and a simulator. The editor is to model a system visually through a graphical interface. The goal is to utilize all the benefits that a high level Petri net provided with convenience. The details are presented below according to the high level Petri net concept's six elements in reference [1]. The simulator is no longer a simple black dot token animation game but to manage the movement of meaningful data. We developed a mandatory compiler with an interpreter to process token data inside transition conditions, which are defined using restricted first-order logic. Besides, a simulation algorithm is applied to ensure its fairness and improve its performance.

B. A Net Graph

Since the graphical elements of a high level Petri net are the same as low level ones, the PIPE's graphical editor is retained.

C. Place Type and Place Marking

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

To implement the concept that tokens with data structure, a data storage system is needed. Based on PIPE, the data layer package is modified by adding three classes: DataType, Token, abToken (Figure 3).

- DataType: The main data structure in class DataType is a Vector storing a list of basic types' name, which is used to show what data structure the token or place holds. The data structure consists of an array of basic types, such as string, integer, etc. For our tool, basic types are limited to strings and integers for the simplicity but are adequate for most of applications. For the convenience of extension on basic types, we introduce a new structure BasicType to data layer. The structure BasicType (see Figure 4) includes a flag data field "Kind" to indicate which type it is (in PIPE+, 0 represents integer, 1 represents string). Space is allocated to both integer and string since it is undecided before the "Kind" is defined. Further extension on basic types needs to enhance the class BasicType by allocating extra space and redefine "Kind".

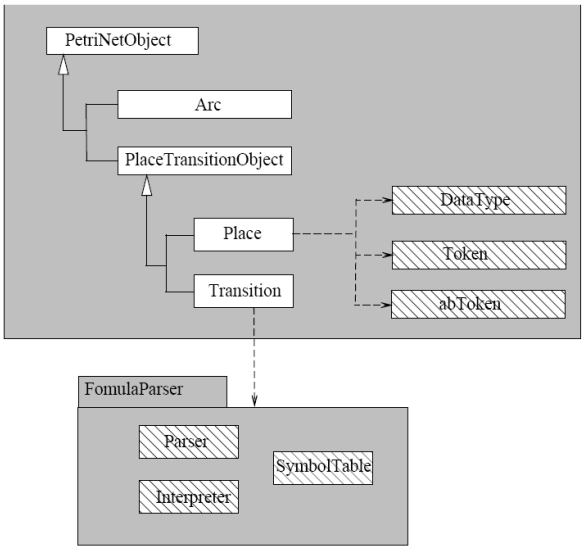

Figure 3 Extensions on DataLayer for PIPE+

- Token: Class Token is added to the data layer to maintain data value. The important field is a Vector storing a list of instances of value with type of the BasicType, see Figure 4. Token is a basic data storage element in the places and its value is calculated by the transitions. The simulation process is fetching data value from the token's BasicType and fill the calculated result value to another token's BasicType.
- Abstract Token: Since first-order logic covers quantification, the whole collection of tokens in a place need to be checked by transition condition expressions. For example, if an expression includes "$\exists x \in X$", all the tokens in "X" needs to be checked to see whether a "x" exists, so the whole collection of tokens is fetched while checking enabledness of a transition. The tokens in this type of place are defined as a power set. A new class abToken (abstract token) is added into the data layer to store the power set. It has a field storing a list of regular tokens with the same data type, so it also has a data type to restrict the tokens data structure. We flatten the nested power sets by duplicating some fields. For example, in a library system, one user may borrow a list of books, so that the database (power set) in library system is {username, password, books_borrowed{book1, book2,...} } is converted into {username, password, book1}, {username, password, book2}. This design sacrifices the space for the convenience of implementation, which can be further improved.

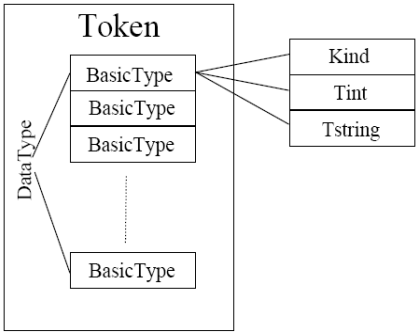

Figure 4 Structure of Class Token

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

In PIPE+, a place stores tokens by List container, the place's capacity is built as unbounded (remember it has nothing to do with the number of different tokens that may appear in a particular place). However, in the discussion of [7], bounded and unbounded places have the same expressive power. A bounded place is preferable for the reason of visualization and redundancy.

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

D. Transition Conditions and Arc Annotations

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

In PIPE+, arc annotations are variables to assist transition expression calculation by mapping token values to expression's predicate variables. Arc variables are restricted to be appeared in the connected transition expression's variables for the mapping. Since a transition is connected by input and output arcs and arcs are connected to places, the predicate variables in the transition expressions are classed into input variables and output variables. For example, in Figure 5, a and b are input variables while c is output variable.

In a transition calculation process shown in Figure 5: In step (1), each token in the connected place is firstly bounded to the connected arc variable; as a pair, {variable, token}, they are fetched into a symbol table of the transition (note the pair with output variable's token value is temporarily empty and to be filled by the result of the expression calculation). In step (2), the input variables in the transition expression can locate token value through the pair's arc variables by looking up symbol table. In step (3), after transition expression calculation, the output variables are assigned with result value and the symbol table's output variable pairs are filled by the value. In step (4), the output pairs' token are added to the connected output places according the arc's variables. For example, as c is on the output arc, c's token in symbol table [bob] is added to the output place.

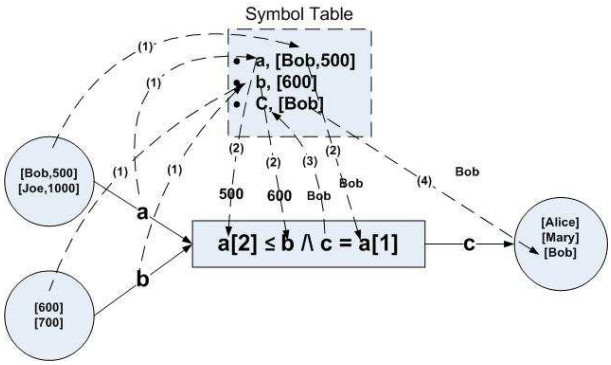

Figure 5 An Enabled Transition Formula Calculation Process

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

$$(a = b) \wedge \exists c \in C((c[1] > a[2]) \wedge (C' = C - \{b\} \cup \{[a[1], c[2]]\})). \quad (1)$$

In (1), lower case letters represent regular tokens, upper case represent power set; *C*' by convention represents output variables and also is a power set (upper letters); it further indicates the clause is a post-condition because output variables at the left side of the equation means assignment; $c[n]$ means the n^{th} element value in *c*'s data structure.

- *Parser and Interpreter*: Because logical formulas need to be parsed and interpreted, we build a compiler with a parser and an interpreter for the restricted first order logic formula. The parser includes a scanner, which is built by a lex file and generated by jflex 1.4.3 [16]. A BNF grammar is built in cup file and generated by leveraging the tool jcup v11 [17]. Since the transition

formula does not explicitly separate pre and post conditions, but only pre-conditions need to be calculated when checking whether the transition is enabled to fire or not, the interpreter has to differentiate pre and post conditions. A trick is found that in the post-condition, it usually starts with an output variable equals a subformula, for example, in (1), $(C' = C - \{b\} \cup \{[a[1], c[2]\})$ is a post-condition because C' is output variable. Therefore in the interpreter, when checking a clause with an "=" operator, the left hand side of the "=" variable is checked. If it is input variable, this clause is a pre-condition and the "=" is interpreted as a logic operator equal, which results in a Boolean true or false; on the other hand, if it is an output variable, the clause becomes a post-condition and the operator now is an assignment clause that assign the result value of right hand side formula to the left hand side output variable.

- Symbol Table: In PIPE+ each transition maintains a temporary symbol table to facilitate the interpreter. It does not use one big table for all the transitions, because it may cause name conflict and is hard to manage. The symbol table contains a list of elements that are structured by a pair of key and object. The pair of key and object is obtained from the transition's connected arc annotation and place. The reason we maintain the pair of key and object instead of key and token is because besides regular token type, the key may pair with a power set (abstract token type). Moreover, the symbol table is initiated each time before a logical formula is checked and cleared after the firing process.
- Declarations: In the standard[1], it comprising definitions of place types, typing of variables and function definitions. In PIPE+, the declarations are already in the modeling process by defining place data types, transition condition formulas and arc annotations.

E. Extensions On GUI

The GUI package in PIPE mainly consists of a GUIFrame, a GUIView, and some supporting classes. The GUIFrame is the PIPE's graphical frame includes a menu, a toolbar and a statusbar. The GUIView is the panel to draw Petri net graphical elements. Since requirements and concepts for high level Petri nets are token storage and flow, our modification to the PIPE's GUI is focused on Petri net elements places, transitions and arcs. The common procedure to extend PIPE's GUI is adding new selections on graphical elements' property setting menu for new features. In PIPE+, after modifying the gui.handler package for each Petri net element class, the new selections are shown in a popup menu by right clicking a Petri net element. The places now have the choices of defining data type and editing tokens; the transitions can contain logical formulas; the arcs can be labeled by variable key. These new features are triggered by additional selections on GUI and used through customized panels or dialogs.

F. Simulator

The simulator not only needs to execute the net model visually, but also has to ensure correctness, fairness and good performance. In PIPE+, the high level Petri net simulator designs as follows:

1) Graphical Simulation: Since in a low level Petri net, tokens are just black dots flowing from one place to another and the animation is visible to the user. In contrast, tokens in high level Petri nets are complex structured data, and especially when the number of tokens is large, which are inappropriate to be displayed upon graphical net; otherwise the graphical annotations are unreadable. Since the execution procedure is invisible to a user, the result can only be checked by looking into the contents of Places. In PIPE+, to view the tokens in the Places, user can open the Place edit panel and the value of tokens are displayed under the text area of Token List. Besides, the firing history is retained from PIPE by listing the fired transition name orderly and updated instantly after a transition fires, thus the user clearly knows a transition is fired.

2) Transition Occurrence Scheduling Algorithm: A scheduler is needed to coordinate the simulator's token flow strategy efficiently. Since the performance of the simulator mostly affected by the times of transition condition calculation, the PIPE+ chooses the scheduling algorithm from [9] to minimize the recalculation of transition condition checking. The idea is to keep track of disabled transitions discovered during the search of enabled transitions, and use the locality principle, that is an occurring transition only affects the marking on immediate neighboring places, and hence the enabling of a limited set of neighbor transitions. For the implementation, we maintain an unknown list and a disabled list. All transitions initialized as unknowns will be randomly picked and checked for enabling status. If the status is disabled, the transition will be moved to disabled list. Upon occurrence of a transition, we update the status of neighboring transitions to the unknown list if they are in the disabled list. The neighboring transitions can be found through occurred transitions' output places. Therefore, the disabled transition avoid recalculation if the tokens of its input places are not changed.

3) Enabling a Single Transition: In the high level Petri net concpets, tokens are meaningful data, when a selected transition start to check its expression, the expression's variables are to be instantiated. Since a transition may connect to a number of input places, where each place contains a list of tokens, to see whether the transition is enabled or disabled, it has to check all the possible combinations of instantiation tokens from its input places. For example, if there are three input places and each place has 3 tokens, the number of their combinations is $3 \times 3 \times 3 = 27$. If one of the three input places is a power set, no matter how many regular tokens inside the abstract token, it only counts as one abstract token. So the combinations reduces to $3 \times 3 \times 1 = 9$ combinations.

4) An Summarization of the Complete Internal Simulation Process:

a) All transitions in the net graph are initially stored in an unknown list; a disabled list is initialized to be empty;

b) A transition is randomly selected from the unknown list, and is checked for enabledness;

c) During the checking process of the selected transition, all the connected arcs and places of the transition are found;

d) Combinations of tokens from the transition's input places are orderly choosen to fill in its symbol table. Since symbols in symbol table are pairs of [key, object]. The keys are from arcs label; the objects are regular tokens. If the input place is a power set, the whole abstract token is sent as an object, otherwise only its first token is sent and the remaining tokens are still in place. For the symbol's key from output arcs, the object is empty because it is to be filled during transition firing action (after interpreting the post condition of transition formula);

e) The formula expression in the transition is checked utilizing a parser. A Boolean value is returned: if it is true, the transition is enabled and is fired immediately; if it is false, the transition is not enabled with the current input tokens, the tokens in symbol table will go back to the input places; if all the combination of input tokens cannot enable the transition, the transition is moved into a disabled list Both checking and firing a transition formula needs to parse and interpret the formula; however, the checking process only affects a formula's pre-condition while the firing process only affects a formula's post-condition.

f) After firing the transition, the tokens in the symbol table are sent to the output places according to the variables of arcs annotation and added to the tail of output places' token list. Since it changes the place marking of the output places, according to the scheduler algorithm's locality principle, if the dependent transitions are in the disabled list, it can now be moved back to the unknown list. Then go back to step b).

g) In step b), when unknown list is empty, the simulation process ended.

V. SOME ISSUES OF PIPE+

A. Limitations of PIPE+

1) Limited Basic Types: As we mentioned above, currently, the place data type of the PIPE+ only supports two basic types, string and integer. Since PIPE+ using a structure to define basic types, the structure can be extended to accommodate more types.

2) Flat Tokens: For the convinience of implementation, the place data type of the PIPE+ does not support nested powerset, such as {Bob, {book1, book2}}, but instead, it stores two flat tokens {Bob, book1}, {Bob, book2}.

3) Restricted First-order Logic for Transition Formula: A new grammar is built for the convenience of interpretation and to avoid ambiguity.

4) No True Concurrency: The PIPE+ only supports interleaving semantics. Besides, it does not support timed Petri nets.

5) Analysis Module: Lack of an integrated tool to analyze the properties of a net model;

6) Bugs and Errors: Since this is the first version of the

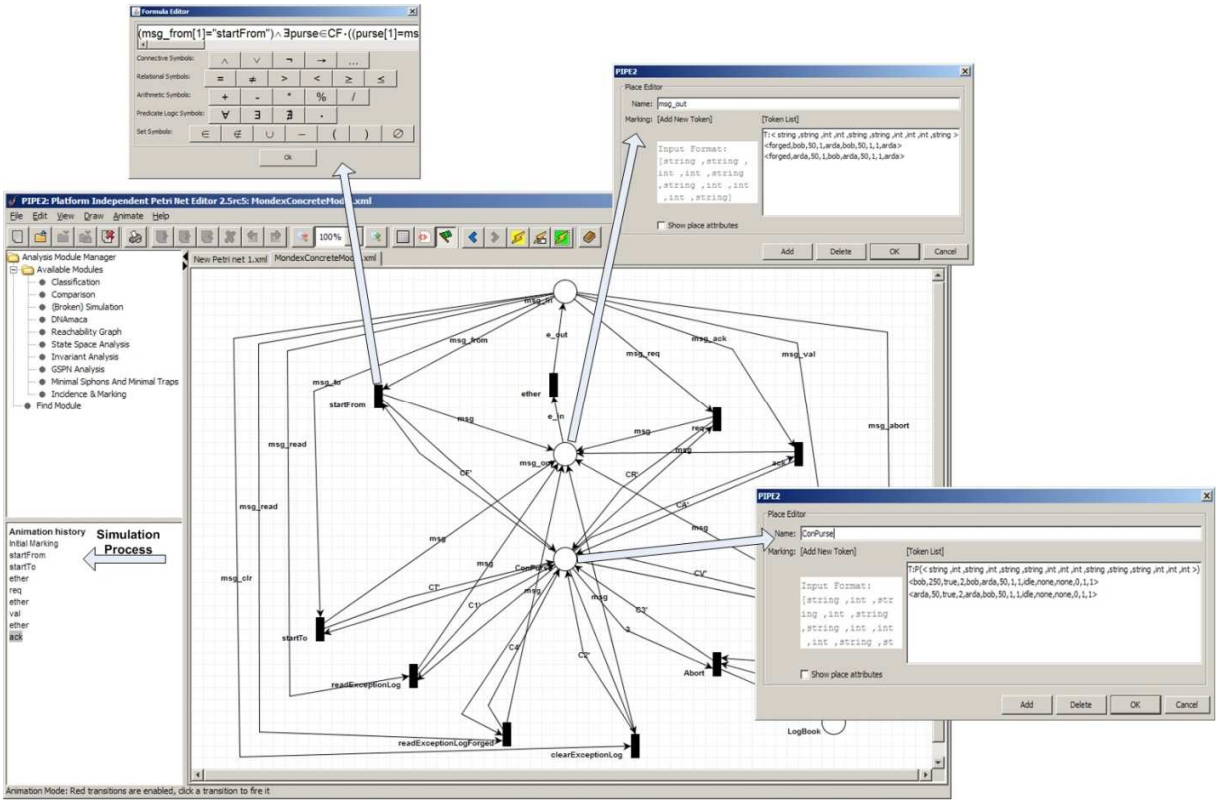

Figure 6 Screenshot of Mondex in PIPE+

PIPE+ and our main purpose is to introduce the new tool, bugs are unavoidable.

B. Testing the PIPE+

The most important part of testing is the transition condition formula. As the new parser and the interpreter were built for the restricted first-order logic formula, its correctness has to be assured. Our test cases are designed mainly on complex formulas including quantifier, relation expressions, arithmetic expressions and set expressions.

C. Using the PIPE+

The PIPE+ has been applied to a Mondex[8] smart card system, which is an electronic purse payment system based on smart card technology. The model for a concrete transaction between two purses has eight operations (including abort) and four statuses, and we translated into PIPE+ model with ten transitions and four places. Figure 6 is a screenshot of Mondex in PIPE+, in which the simulation of a transition firing sequence is shown at the left bottom of the interface's frame. After no more transition is available to fire, the result of the simulation is a final marking that can be read by opening the places, msg_out and ConPurse, to view contents which are tokens' data.

VI. CONCLUSIONS

In this paper, we present a tool PIPE+ supporting high level Petri nets editing and simulation. We believe PIPE+ can be a valuable tool for concurrent and distributed system modeling and simulation. PIPE+ is built upon an open source tool PIPE for low level Petri nets. We illustrated the process of extending PIPE, and discussed our design strategies, which provide helpful insights for others to create Petri net tools suit their own needs. Furthermore, PIPE+ is an open source tool and thus is available for sharing and continuous enhancements from worldwide research community.

Acknowledgements This work was partially supported by NSF grants HRD-0833093.

REFERENCES

[1] High-level Petri Nets-Concepts, Definitions and Graphical Notation, Version 4.7.1, 2000

[2] Pere Bonet, Catalina M. Llado, Ramon Puigjaner, "PIPE v2.5: a Petri Net tool for performance modeling," Proc. 23rd Latin American Conference on Informatics (CLEI 2007), San Jose, Costa Rica, October 2007

[3] Reisig, Wolfgang, "Petri nets: an introduction," Springer-Verlag New York, Inc.NY, 1985

[4] Tadao Murata, "Petri Nets: Properties, Analysis and Applications," Proceedings of IEEE, vol. 77 No.4, Chicago, IL, April 1988

[5] Petri Net Tool Database. http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html

[6] James Bloom, Clare Clark, Camilla Clifford, Alex Duncan, Haroun Khan, Manos Papantoniou, "Platform Independent Petri-net Editor: Final Report," London, March 2003

[7] Carlos A. Heuser, Gernot Richter, "Constructs for Modeling Information Systems with Petri Nets," 13th International Conference on Application and Theory of Petri Nets, 1992, Sheffield, UK

[8] Reng Zeng, Xudong He, "A Formal Specification of Mondex Using SAM," The Fourth IEEE International Symposium on Service-Oriented System Engineering, 2008

[9] Kjeld H. Mortensen, "Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator," 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus University, August 2001

[10] Didier Buchs, Steve Hostettler, Alexis Marechal, and Matteo Risoldi, "AlPiNA: An Algebraic Petri Net Analyzer," J. Esparza and R. Majumdar (Eds.): TACAS 2010, LNCS 6015, pp. 349–352, 2010

[11] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets," Proc. of 24th International Conference on Applications and Theory of Petri Nets , 2003

[12] R. Milner, M. Tofte, R. Harper, and D. MacQueen, "The definition of Standard ML," MIT Press, Cambridge, MA, 1997

[13] CPN ML Reference, http://www.daimi.au.dk/designCPN/man/Reference/Reference.Main3.CPN.ML.pdf

[14] Andrews, Peter, "An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof," 2nd ed. Kluwer Academic Publishers, 2002.

[15] X. He and T. Murata: "High-Level Petri Nets – Extensions, Analysis, and Applications", Electrical Engineering Handbook (ed. Wai-Kai Chen), Elsevier Academic Press, 2005, 459-476.

[16] JFlex Lexical Analyzer Generator. http://jflex.de/index.html

[17] JCUP Parser Generator. http://www2.cs.tum.edu/projects/cup/