SAMAT - A Tool for Software Architecture Modeling and Analysis

Su Liu, Reng Zeng, Zhuo Sun, Xudong He
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
{sliu002, zsun003, rzeng001, hex}cis.fiu.edu

***Abstract*—A software architecture specification plays a critical role in software development process. SAM is a general framework for developing and analyzing software architecture specifications. SAM supports the scalability of architectural descriptions through hierarchical decomposition and the dependability analysis of architectural descriptions using a dual formalism based on Petri nets and temporal logic. In this paper, we present SAMAT (Software Architecture Modeling and Analysis Tool), a tool to support the hierarchical modeling and analyzing of software architecture specifications in SAM. SAMAT nicely integrates two external tools PIPE+ for behavioral modeling using high-level Petri nets and SPIN for model checking system properties.**

I. Introduction

Since late 1980s, software architecture has become an active research area within software engineering for studying the structure and behavior of large software systems [16]. A rigorous approach towards architecture system design can help to detect and eliminate design errors early in the development cycle, to avoid costly fixes at the implementation stage and thus to reduce overall development cost and increase the quality of the systems. SAM [17], [8], [9], [10] is a general framework for systematically modeling and analyzing software architecture specifications. Its foundation is a dual formalism combining a Petri net model for behavioral modeling and a temporal logic [9] for property specification. To support the application of SAM framework, we are developing a tool set, called SAMAT.

SAMAT has the following features:

1) supporting software architecture modeling through hierarchical decomposition;
2) modeling software component and connector behaviors using high-level Petri nets [2];
3) specifying model constraints (system properties) using first-order linear time temporal logic [15];
4) analyzing the SAM's behavior model through model translation and model checking using SPIN [11].

In the following sections, we discuss our development of SAMAT and its main features. Section 2 gives an overview of the SAM framework as well as its foundation. Section 3 presents the components and functionality of SAMAT and the design of SAMAT. Section 4 provides the model translation process from SAM to PROMELA for model checking in SPIN. Section 5 shows an example to illustrate the use of SAMAT. Section 6 compares the SAM famework and SAMAT with related software architecture framework and tools. Section 7 contains a conclusion.

II. The SAM Framework

SAM [17] is a general formal framework for specifying and analyzing software architecture. SAM supports the hierarchical modeling of software components and connectors.

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

Figure 1 shows a hierarchical SAM specification model. The boxes, such as "A1" and "A2", are called compositions, in which "A1" is component and "A2" is connector. Each composition may contain other compositions, for example "A1" wrap up three compositions: "B1", "B2" and "B3". Each bottom-level composition is either a component or a connector and has a property specification (a temporal logic formula). The behavior model of each component or connector is defined using a Petri net. Thus, composing all the bottom-level behavior models of components and connectors implicitly derives the behavior of an overall software architecture. The intersection among relevant components and connectors such as "P1" and "P2" are called ports. The ports form the interface of a behavior model and consist of a subset of Petri net places.

A. The Foundation of SAM

The foundation of SAM is based on two complementary formal notations: predicate transition nets [6], [7] (a class of high-level Petri nets) and a first-order linear-time temporal logic [15].

1) Predicate transition nets (PrT nets), a class of high-level Petri nets, are used to define the behavior models of components and connectors. A PrT net comprises a net structure, an underlying specification and a net

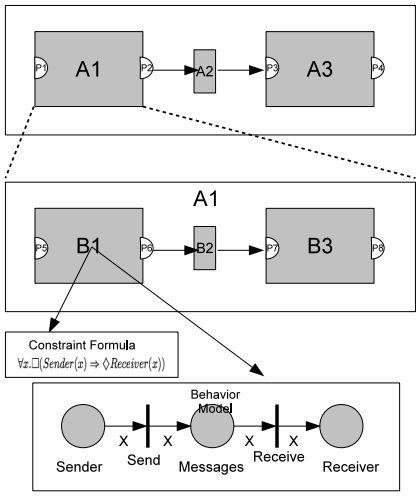

Figure 1. Hierarchical SAM Specification Model

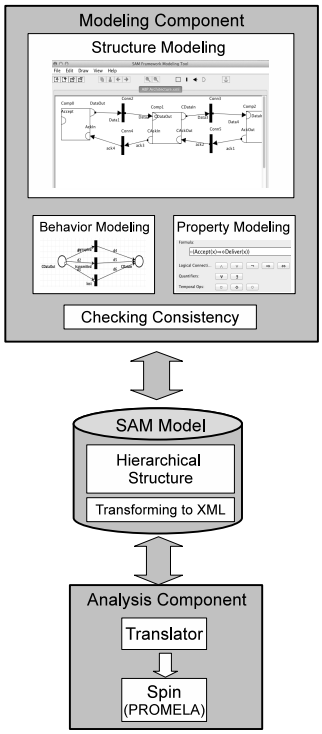

Figure 2. The Functional View of SAMAT

inscription [7]. A token in PrT nets contains typed data and the transitions are inscribed with expression as guard to move or stop the tokens.

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

SAM supports the behavior modeling using PrT nets [14] and property specification using the FOLTL. SAM supports structural as well as behavioral analysis of software architectures. Structural analysis is achieved by enforcing the completeness requirement imposed on the components and connectors within the same composition, and the consistency requirement imposed on a component and its refinement. Behavioral analysis requires the checking of system properties in FOLTL satisfied in the behavioral models in PrT nets. In this paper, we analyze SAM behavior model by leveraging SPIN [11], which is a popular formal verification tool used worldwide.

III. SAMAT

In this section, we present the functional view and the design view of SAMAT.

A. Functional View of SAMAT

SAMAT is comprised of a modeling component, a SAM model, and an analysis component (Figure 2). The modeling component has three functions: structure modeling creates hierarchical compositions, behavior modeling specifies behaviors of software components/connectors using Petri nets, and property modeling defines property specifications using temporal logic. The SAM specification is a hierarchical structure integrating the results of structure, behavior, and property modeling, which can be transformed into XML format. The analysis component contains a translator to generate a model suitable for model checking.

B. Design View of SAMAT

SAMAT is a platform independent (implemented in Java) and visual software architecture modeling and analysis tool. As shown in Figure 3, SAMAT is designed using the Model-Vew-Control pattern.

1) The model of SAMAT includes a hierarchical layer of SAM compositions that builds the SAM model in Figure 2. It also include the functionalities of generating flat Petri net model and conjunctions of FOLTL formulas for analysis purpose.
2) The graphical interface of SAMAT is developed using Java Swing API as it provides full GUI functionalities and mimics the platform it runs on. It consists of a SAM composition editor, a PIPE+ editor, a FOLTL editor and an analysis displayer. The composition editor is used for modeling the SAM compositions into a hierarchical structure; the PIPE+ editor is used for modeling the behaviors of SAM model via PrT nets; the FOLTL editor is used for defining the properties into FOLTL formulas; the analysis displayer is used for showing the analyzing result generated by SPIN [11].
3) The controller is comprised of composition controllers, a XML transformer and a PROMELA translator. The composition controllers provide options to specify detailed

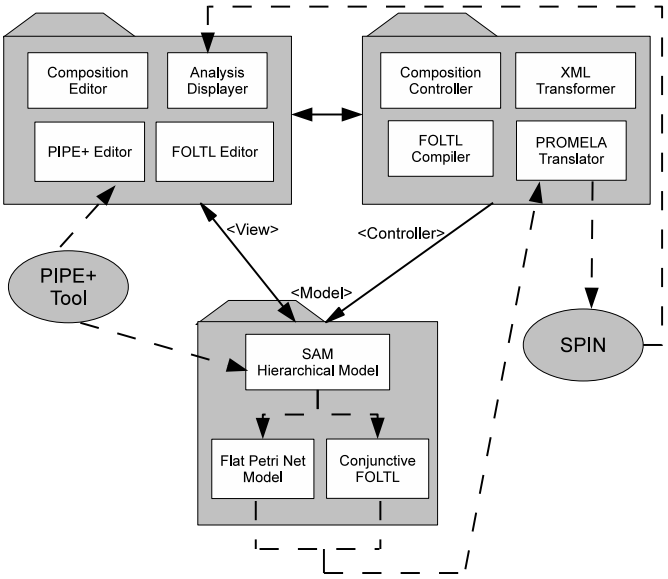

Figure 3. The Design View of SAMAT

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

SAMAT integrates two external tools: PIPE+ [14] for behavior modeling and SPIN [11] for model analysis.

C. SAM Hierarchical Model in SAMAT

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

The Petri nets layer in the bottom of Figure 4 is the behavior model of its parent composition. In this case, it is a high-level Petri net formalism modeled in PIPE+ editor. Once a Petri net model is created, it is transformed and saved in XML format and is appended to its parent SAM composition.

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

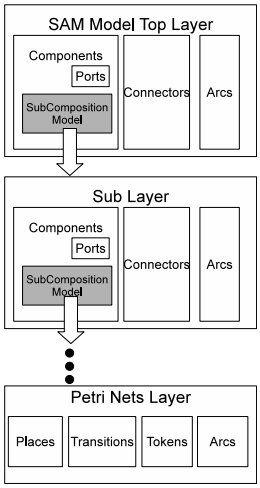

Figure 4. The SAM Hierarchical Model

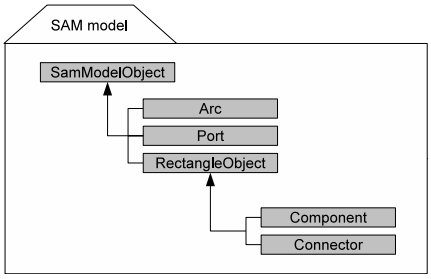

Figure 5. The Architecture of SAM Model Package

the sub composition can be further decomposed, another new tab will be built. If the behavior model option is selected, PIPE+ is triggered for the user to build a behavior model using Petri nets. Therefore, the top-down decomposition process is straightforward.

D. Inheritance Class Design in SAMAT

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

E. FOLTL Editor

One of the underlying formalism in SAMAT is FOLTL. The vocabulary and models of FOLTL used in SAMAT are based on the high-level Petri net formalism and follow the approach defined in [13]. An example FOLTL formula is $\Box((x>y) \Rightarrow \Diamond (b=1))$, where variables are restricted to the underlying behavior models' arc variables. Since in each composition, SAMAT integrates a FOLTL formula editor where a user can specify system properties, the composition-level property specification

is obtained by conjoining the property specifications of all components and connectors. The FOLTL compiler checks the syntax of a FOLTL formula and the translator generates constraint code in PROMELA.

F. PIPE+

The other formalism in SAMAT is PrT nets, which are a class of high-level Petri nets. We integrate an existing open source high-level Petri net tool PIPE+ [14] to specify the behavior model of the SAM architecture. PIPE+ is capable of specifying and simulating high-level Petri nets proposed in [2]. SAMAT leverages PIPE+'s editing mode in which a high-level Petri net behavior model can be developed graphically with dragging and dropping actions. The high-level Petri net model is comprised of:

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

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

G. XML Transformer

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

IV. Verifying SAM Specifications

To ensure the correctness of a software architecture specification in SAM, we have to check all the constraints are satisfied by the corresponding behavior models. To automate the verification process in SAMAT, we leverage an existing linear time temporal logic model checking tool SPIN [11].

A. SPIN and PROMELA

1) The SPIN Model Checker: SPIN [11] is a model checker for automatically analyzing finite state concurrent systems. It has been used to check logical design errors in distributed systems, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. A concurrent system is modeled in the PROMELA (Process or Protocol Meta Language) modeling language [11] and properties are defined as linear temporal logic formulas. SPIN can automatically examine all program behaviors to decide whether the PROMELA model satisfies the stated properties. In case a property is not satisfied, an error trace generated, which illustrates the sequence of executed statements from the initial state. Besides, SPIN works on-the-fly, which means that it avoids the need to preconstruct a global state graph or Kripke structure, as a prerequisite for the verification of system properties.

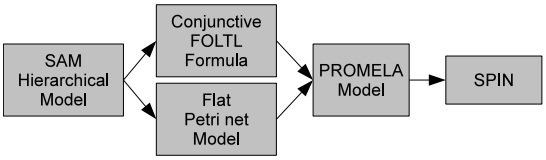

Figure 6. Verifying SAM Specifications

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

B. From SAM Model to PROMELA Model

As shown in Figure 6, SAMAT starts by generating a flat (high-level) Petri net model from its hierarchical SAM model. Then, SAMAT automatically translates the flat Petri net model into a PROMELA model. Combined with FOLTL constraint formulas, SPIN can check the PROMELA model and output a verification result to SAMAT.

1) Generating An Integrated Flat Petri Net Model: Because a SAM model is hierarchically specified and each component in a different layer has its own behavior model, direct translation of a hierarchical SAM specification into PROMELA could result in a complex model not preserving the original semantics. Thus, SAMAT preprocesses the model by flattening the hierarchical structure.

In this phase, all the individual Petri net models created in different components of a SAM model need to be connected by directed arcs in both horizontal and vertical dimensions. Therefore, selecting interfaces among all the Petri net models are important. Because each Petri net model has input places (places without input arc, e.g. Sender in Figure 1) and output places (places without output arc, e.g. Receiver in Figure 1), which are used to communicate with other models, these input and output places are chosen as candidate interface places heuristically. Similarly, each SAM component has its input ports (P1 in Figure 1) and output ports (P4 in Figure 1) for the communication with other components, these input and output ports form the interface of the component.

The connection strategies are :

- **Horizontally**: Each SAM component has its input ports and output ports specified by one of the interface places of the underlying Petri net model (e.g. in Figure 7,

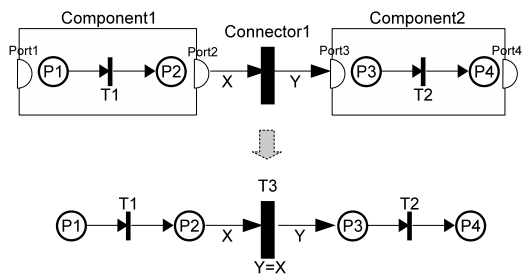

Figure 7. Generating Analysis Model by Horizontal Connection

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

Table I
MAP FROM HIGH-LEVEL PETRI NET TO PROMELA

Port 1 specified by P1 and Port 2 specified by P2). Integrating Petri net models from different components in the same hierarchical layer is by connecting the interface places. Moreover, the components in the same layer are connected by SAM connectors and arcs, so that SAMAT transforms them into Petri net transitions and arcs respectively. A new transition is created for each connector during the transformation (e.g. in Figure 7 is T3). The pre-condition of such transition is true by default; however a post-condition may be added. In the example, a post-condition "Y=X" is added. The variables in the new transition formula match the connector's input and output arc variables. The sort of the variables is exactly the sort of the interface places, specified in ports, through connected arcs. Corresponding new arcs are added to reserve the flow relationships, which are connected with the interface places in ports and related transitions. For example, a new arc between place P2 in Port2 and T3 in Connector1.

- **Vertically**: The input or output ports not connected with any arcs in a component are mapped to the corresponding input and output ports in the parent component. For example in Figure1, ports P1 and P2 in top layer component A1 map to the second layer's input port P5 and output port P8.

Thus, the Petri net models are connected and flattened into an integrated flat Petri net model that is ready to be translated into PROMELA model.

2) Translating Behavior Models to PROMELA: The translation process maps a high-level Petri net model to a PROMELA model. The resulting PROMELA model should catches the concept of high-level Petri nets defined in [2] and preserves the semantics. The PROMELA program's major parts are definitions of places and place types, transition enabling and firing inline functions, a main process and an init process that defines the initial marking.

The translation map is shown in Table 1:

- **Translating places**: We predefined each message type (place type) into a new structure. Places and place types are mapped to PROMELA's buffered channels and predefined message types. Besides, structured tokens are mapped to typed messages in PROMELA. Because SPIN verifies a model by exhaustive state searching, we set bounds to limit the number of tokens in places. The bounds are then mapped to the length of the channels. A sample PROMELA program resulted from place translation is shown below:

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

- **Translating transitions**: In high-level Petri nets, a transition expression consists of a precondition and a postcondition. The precondition defines the enabling condition of the transition, and the postcondition defines the result of the transition firing. Each precondition and postcondition are translated into two inline functions, is_enabled() and fire(), respectively. fire() is triggered by the Boolean variable in is_enabled() evaluated to be true, otherwise it is skipped as the transition is not enabled. To check the precondition of a transition expression, we first consider a default condition that whether each of the input place has at least one token by checking the emptiness of each mapped channel. Because the expression is usually defined by conjunct clauses, we then evaluate each clause (the postcondition clauses are not evaluated this time). The evaluation process includes non-deterministically receiving a message from an input channel to a local variable, instantiating the Boolean expression, and evaluating it. A sample PROMELA program from transition translation is shown below:

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

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

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

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

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

- **Using basic data types**: Since the basic data types supported in PIPE+ are integer and string, which are mapped to "int" and "short" in PROMELA respectively.
- **Handling non-determinism**: In high-level Petri nets, tokens are meaningful data and usually different from each other and thus different firing orders result in different markings. Therefore, a non-deterministic inline function is defined and is called to non-deterministically pick a token from an input place each time a precondition is evaluated.
- **Supporting power set**: Because PIPE+ supports quantifiers in restricted first order logic formulas in transition expression, the domain of each quantified variable is a list of tokens as a power set contained in a place. For this kind of places, we are not dealing with one message but all the messages in the channel, we do not put all received messages into a local variable but directly manipulate the channel. The strategy is when the first message is received from the channel, it is used and then is sent back immediately.

3) FOLTL Formula: Since the FOLTL constraint formula extracted from the SAM composition model is conjoined into one integrated conjunctive FOLTL formula, the translation process is straightforward. We only need to wrap the formula by following PROMELA's syntax:

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

4) Translation Correctness: The translation correctness is ensured by the following completeness and consistency criteria [18], [3]

Let *N* be a given Petri net and P_N be the resulting PROMELA program from the translation.

- **Completeness** Each element in *N* is mapped to P_N according to the mapping rules described in Section 4.2.2.
- **Consistency** The dynamic behavior of *N* is preserved by P_N as follows:
  - A marking of *N* defines the current state of *N* in terms of tokens in places, our place translating rule correctly maps each marking into a corresponding state in P_N;
  - The initial marking of *N* is correctly captured by the initial values of variables in the init{} process of P_N;
  - The enabling condition and firing result of each transition *t* in *N* is correctly inline functions "is_enabled_Transition_i" and "fire_Transition_i" respectively;
  - The atomicity of enabling and firing a transition in *N* is preserved in P_N by language feature "atomic{}".
  - An execution of *N* is firing sequence $\tau = M_0 t_0 M_1 t_1 ... M_n t_n ...$, where $M_i (i \in nat)$ is a marking and $t_i (i \in nat)$ denotes the firing of transition t_i. Each execution is correctly captured by the construct "do ... od" in the "Main" Promela function, which produces an equivalent execution sequence $\sigma = S_0 T_0 S_1 T_1 ... S_n T_n$, where $S_i (i \in nat)$ is a state and T_i denotes the execution of inline function "Transition_i".

The proofs of the completeness and consistency are straightforward and can be found in [18], [3] .

C. Verification using SPIN

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

V. Using SAMAT

The alternating bit protocol (ABP) is a simple yet effective protocol for reliable transmission over lossy channels that may lose or corrupt, but not duplicate messages. It has been modeled and analyzed in [10]. In this section, we show some snapshots of using SAMAT in modeling and verifying this protocol.

In Figure 8, the top layer of ABP model in SAMAT consists of three components and four connectors. The first component "Sender" has a behavior model shown in Petri net. On the right, it shows the FOLTL editor to editing formula <>(Deliver(m) = 5). After the modeling process, SAMAT automatically generates PROMELA code as an input for SPIN and displays the model checking result after SPIN finished model checking. In this case an error is found, the replayed simulation on the error path is shown below the model checking result. The error indicates the ABP specification model in [10] is incorrect. A deadlock state (a none final state such that none of the transitions are enabled) can be reached when an acknowledgement message was corrupted in the channel and a resend message successfully reached the receiver's DataIn place. This discovery highlights the great benefits and usefulness of SAMAT.

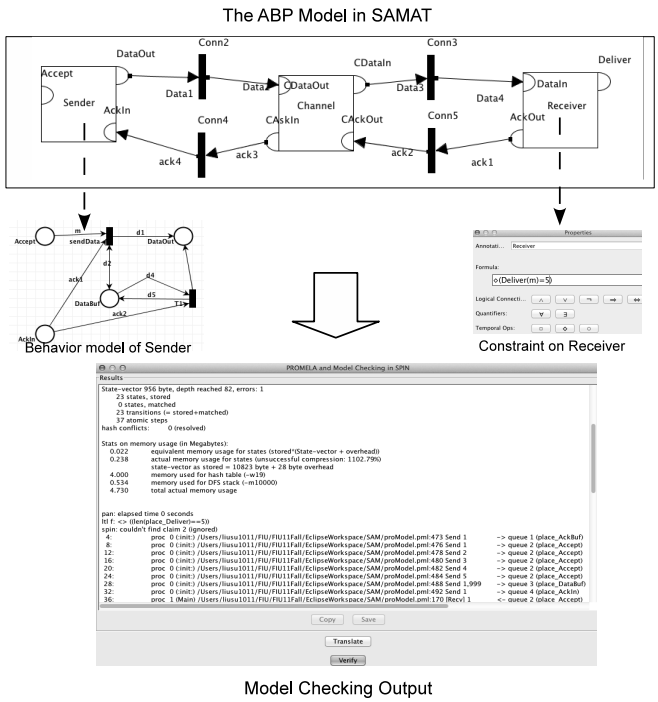

Figure 8. Model the ABP in SAM Tool

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

Table II
SOFTWARE ARCHITECTURE FRAMEWORKS

VI. RELATED WORK

In the past decades, many software architecture modeling and analysis framework were proposed and their supporting tools were built. Several comparative studies[19], [4], [5] on these frameworks were published. Table 2 presents a comparison of several architecture modeling and analysis frameworks and their supporting tools.

Besides, CPN Tools [1] is a widely used Coloured Petri nets [12] tool that can also be used for modeling and analyzing software architecture. In terms of software architectural modeling, CPN Tools do not directly support architecture level concepts and features and thus the resulting models do not match user's abstraction well and can be difficult to understand. In terms of analysis, CPN Tools generates a full state space during verification and thus is often limited by the memory size while SPIN can perform verification on-the-fly to avoid full state space preconstruction so that it can handle complex behavior models.

VII. CONCLUSION

In this paper, we present a tool SAMAT for modeling and analyzing software architecture specifications in SAM. SAMAT leverages two existing tools, PIPE+ for building Petri net models and SPIN for analyzing system properties. SAMAT can be a valuable tool for complex concurrent and distributed system modeling and analysis. SAMAT is an open source tool and is available for sharing and continuous enhancements from worldwide research community.

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

REFERENCES

[1] Cpn tools. http://cpntools.org.
[2] *High-level Petri Nets - Concepts, Definitions and Graphical Notation*, 2000.
[3] Gonzalo Argote-Garcia, Peter J. Clarke, Xudong He, Yujian Fu, and Leyuan Shi. A formal approach for translating a sam architecture to promela. In *SEKE*, pages 440–447, 2008.
[4] Paul Clements and Mary Shaw. The golden age of software architecture: A comprehensive survey. Technical report, 2006.
[5] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *Software Engineering, IEEE Transactions on*, 28(7):638 – 653, jul 2002.
[6] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13(1):109 – 135, 1981.
[7] Xudong He. A formal definition of hierarchical predicate transition nets. In *Application and Theory of Petri Nets*, pages 212–229, 1996.
[8] Xudong He and Yi Deng. Specifying software architectural connectors in sam. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):411–431, 2000.
[9] Xudong He and Yi Deng. A framework for developing and analyzing software architecture specifications in sam. *Comput. J.*, 45(1):111–128, 2002.
[10] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using sam. *Journal of Systems and Software*, 71:1–2, 2004.
[11] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
[12] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER*, page 2007, 2007.
[13] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.
[14] Su Liu, Reng Zeng, and Xudong He. Pipe+ - a modeling tool for high level petri nets. *International Conference on Software Engineering and Knowledge Engineering (SEKE11)*, pages 115–121, 2011.
[15] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
[16] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23:31–39, March 2006.
[17] Jiacun Wang, Xudong He, and Yi Deng. Introducing software architecture specification and analysis in sam through an example. *Information & Software Technology*, 41(7):451–467, 1999.
[18] Reng Zeng and Xudong He. Analyzing a formal specification of mondex using model checking. In *ICTAC*, pages 214–229, 2010.
[19] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723 – 744, 2010.