Modeling and Analyzing the Software Architecture of a Communication Protocol Using SAM

Tianjun Shi, Xudong He
School of Computer Science, Florida International University, Miami, FL 33199

Abstract: SAM is a general framework for modeling and analyzing software architectures. In this paper, we apply SAM to model and analyze a popular alternating-bit communication protocol. To compare with other existing formalization of this communication protocol and to show the salient features of SAM, we provide two specifications of the communication protocol – one without a timer and one with a timer. Furthermore we explore two different translation approaches using the model checking language SMV, and compare their effectiveness. We provide some general rules, including rules to deal with timing issues, to translate predicate transition nets into SMV specifications so that automatic verification of systems properties through model checking can be done.

Key words: Software architecture, formal specification and verification, model checking, predicate transition nets, temporal logic

1. INTRODUCTION

There has been a large amount of research work on the representation and analysis of software architecture since software architecture appeared as a promising and important field in software engineering. After proposed as a general development model to specify and analyze software architecture in [WHD99], SAM has been further refined and extended in several ways. Originally, SAM used low-level Petri nets and propositional temporal logic as its underlying foundation, which is adequate to describe the static structure and simple control flows of software architectures, but is not capable to specify data aspect of software architecture. By using predicate transition nets (PrT nets) and first order temporal logic as its foundation, SAM is now capable to specify all aspects of software architecture [HD02].

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

One important feature of an *architecture description language* (ADL) [MT00] is to specify non-functional properties such as timing, security, and performance. Timing issues are extremely important in safety and mission critical systems. To specify timing related properties, a formal method explicitly dealing with time is often used, for example time Petri nets and real-time temporal logic. Previously, we used time Petri nets [BD91] and real-time computational tree logic [EMS92] in SAM to specify a control and command system [WHD99]. To improve the understandability and maintain the flexibility of SAM framework, we want to use non-timed versions of formal methods as the foundation of SAM. Researchers in both Petri net community [GMM91] and the temporal logic community [AL94] have shown how to deal with timing issues without explicitly introducing timing features into a formal method.

In this paper, we show how to use SAM to model and analyze the software architecture of a popular *alternating bit protocol* (ABP) with behavioral and non-functional properties. To compare with other existing formalization of ABP and to show the salient features of SAM, we provide two specifications of ABP – one without a timer and one with a timer. Furthermore we explore two different translation approaches using the model checking language SMV, and compare their effectiveness. We provide some general rules, including rules to deal with timing issues, to translate PrT nets into SMV specifications so that automatic verification of systems properties through model checking can be done. This work has improved our previous results in several ways: (1) it provides an approach to model timing-related behavior using PrT net and to specify timing properties using first order temporal logic of a software architecture using SAM, (2) it shows a technique to analyze a first order temporal logic specification under a PrT model using symbolic model checking.

2. AN OVERVIEW OF SAM

SAM is a general formal framework for specifying and analyzing software architectures [WHD99], and has been developed in the past five years at the Florida International University. The development philosophy of SAM is to introduce a simple, flexible, and formal software architecture specification and analysis model without the need to invent another ADL. In SAM, a software architecture is defined by a hierarchical set of compositions, in which each composition consists of a set of components, a set of connectors, and a set of constraints to be satisfied by the interacting components. SAM uses as its foundation a dual formalism combining a Petri net model and a temporal logic. Petri nets are used to define the behavior

models of components and connectors while temporal logic is used to specify properties of components and connectors. The correctness of a software architecture specification is assured when all the system properties hold in the behavior models. SAM does not fix a particular Petri net model and a temporal logic as its underlying formal foundation. Different Petri net models and temporal logics could be chosen based on a particular application. For example, real-time Petri net and real-time computational tree logic were used to study software architectures of real-time systems [WHD99], predicate transition nets and first order linear time temporal logic were used to specify and verify software architectures explicitly dealing with data abstraction [HD00]. In this paper, we use the combination of PrT nets and first order linear time temporal. For more detailed information of SAM, please refer to [WHD99, HD00]. The detailed introduction to PrT nets and temporal logic can be found in [HD02].

3. MODELING OF ALTERNATING BIT PROTOCOL

In this section, we first take a look at the alternating bit protocol, and then choose two versions of ABPs as our examples to explain the modeling in SAM. The conventions for modeling and property specification conform to those in [HD02].

3.1 Introduction to ABP

Alternating bit protocol is a simple yet effective protocol for reliable transmission over lossy channels that may lose or corrupt, but not duplicate, messages. The protocol consists of a sender, a receiver, and two channels, and main goal of ABP is to ensure that the receiver will eventually deliver an accepted message in the sender. There are several variant ABPs, among which the main differences focus on whether the channel can detect the lost or corrupted messages or not. If yes, a message is resent when a lost or corrupted message is detected. Otherwise, a timeout mechanism is introduced to send message periodically when no desired acknowledgement is received during a certain time. The protocol guarantees that (1) an accepted message will eventually be delivered, (2) the accepted messages are delivered in order, and (3) an accepted message is delivered only once.

We choose two various versions of ABPs as our examples — ABP without timer and ABP with timer. The former assumes that the channels may lose or corrupt, but not duplicate, messages, and the lost or corrupted messages are detectable; the latter assumes that channels may lose, but not corrupt or duplicate, messages, and the lost messages is not detectable. Since

a corrupted message can be discarded simply and deemed as lost message, it is reasonable to assume the channel does not corrupt message.

3.2 Model of ABP without A Timer

It is straightforward to construct the architecture of this ABP from its requirements. As shown in *Figure 1*, there are two components, *Sender* and *Receiver*, and two connectors, *DataChannel* and *AckChannel*. ABP has two interface ports. One is the *Accept* port, which is the interface accepting messages from the environment; the other is the *Deliver* port, which is the interface delivering the accepted messages out to the environment. All other ports are internal to the ABP system.

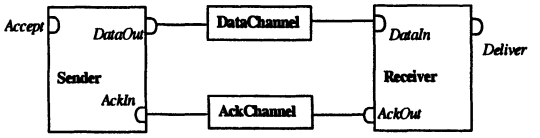

Figure 1. Architecture of ABP without a timer

The architecture property specifications are derived directly from the requirements of the protocol. In fact, they are formal expressions of the protocol requirements in temporal logic. The property specifications include:

P(1) Liveness property: accepted message will eventually delivered.
$\Box ((Accept(x) \rightarrow \Diamond Deliver(x))$

P(2) Safety property: messages will be delivered in order.
$\Box ((Accept(x1) \mu (Accept(x2) \wedge \neg Accept(x1))$
$\rightarrow \Diamond (\neg Deliver(x2) \mu Deliver(x1)))$

P(3) Safety property: Each messages is delivered only once.

It is a bit complex to express this property since the properties in SAM are described by states rather than by actions. To guarantee this property, what we need is to ensure that the action sending a message by sender and the action delivering a message by receiver take place in turn. That is, one sending action results in exact one delivering action. Therefore it is straightforward to describe this property by actions as follows:

$(sendData(x) \rightarrow O (\neg sendData(y) \mu deliverData(z)))$
$\wedge (deliverData(x) \rightarrow O (\neg deliverData(y) \mu sendData(z)))$

Where the first subformula means that, after the sender sent a message, it does not send another message until the receiver delivers a message; and the second subformula means that, after the receiver delivered a message, it does not deliver another message until the sender sends a message.

State changes result from actions. For example, the sending action results in the changes of state from $Accept(x)$ to $\neg Accept(x)$. Thus the action

sendData(*x*) can be described by state as *Accept*(*x*) ∧ O¬*Accept*(*x*). Therefore, we can describe property P3 in SAM as follows:

((*Accept*(*x*) ∧ O¬*Accept*(*x*))
→ O (¬(*Accept*(*y*) ∧ O¬*Accept*(*y*)) μ (¬*Deliver*(z) ∧ O *Deliver*(z))))
∧ ((¬*Deliver*(*x*) ∧ O*Deliver*(*x*))
→ O(¬(¬(*Deliver*(*y*) ∧ O*Deliver*(*y*)) μ (*Accept*(z) ∧ O¬*Accept*(z))))

Following the process of ABP and its requirements, we can develop the property specification and corresponding behavior model of each element in the architecture, as shown in figures 2-4 respectively.

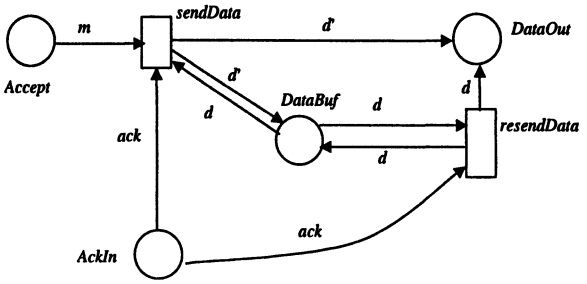

Figure 2. Behavior model of Sender

The net inscription of Sender model is as follows:

φ(*Accept*) = MESSAGE, where MESSAGE is the type of string.
φ(*AckIn*) = BIT ∪ {*lost*, *corrupted*}, where BIT = {0, 1}.
φ(*DataOut*) = φ(*DataBuf*) = BIT× MESSAGE
R(*sendData*) = (*ack*∈ BIT ∧ *ack* = *d*[1] ∧ *d*'[1] = 1- *ack* ∧ *d*'[2] = *m*)
R(*resendData*) = (*ack* = *lost* ∨ *ack* = *corrupted*)

The initial marking of Accept is dependent on the environment, and the other initial markings of the other three Predicates are as follows:

M_0(*AckIn*) = {1}, M_0(*DataBuf*) = {<1,""> }, M_0(*DataOut*) = { }

And the corresponding property specification is:

□ ((*AckIn*(*x*) ∧ (*x* = *lost* ∨ *x* = *corrupt*) ∧ *DataBuf*(*y*)) → ◇*DataOut*(*y*))
□ ((*AckIn*(x) ∧ *Accept*(*m*) ∧ *DataBuf*(*y*) ∧ *x* = *y*[1])
→ ◇ *DataOut*(<1-*x*, *m*>))

The net inscription of Receiver model is as follows:

φ(*Deliver*) = MESSAGE, φ(*AckOut*) = φ(*AckBuf*) =BIT
φ(*DataIn*) = (BIT × MESSAGE) ∪ {*lost*, *corrupted*}
R(*resendAck*) = (*d* = *lost* ∨ *d* = *corrupted*)
R(*deliverData*) = (*d* ∈BIT×MESSAGE ∧ *d*[1] = 1-*b* ∧ *b*'= *d*[1] ∧ *m* = *d*[2])
M_0(*AckBuf*) = {1}, M_0(*DataIn*) = M_0(*AckOut*) = M_0(*Deliver*) = { }

The corresponding property specification is:

□ ((*DataIn*(*x*) ∧ (*x* = *lost* ∨ *x* = *corrupt*) ∧ *AckBuf*(*y*)) → ◇*AckOut*(*y*))
□ ((*DataIn*(*x*) ∧ *AckBuf*(*y*) ∧ *x*[1] = 1 - *y*) → ◇*AckOut*(*x*[1]))

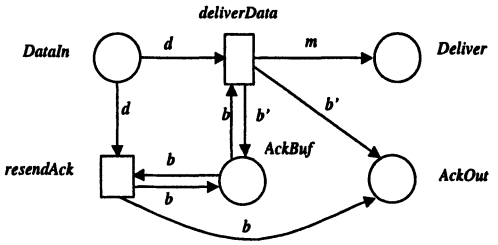

Figure 3. Behavior model of Receiver

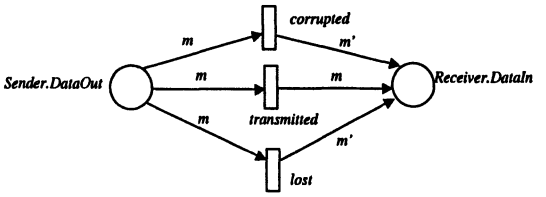

Figure 4. Behavior model of the channel

The behavior models for *DataChannel* and *AckChannel* have the same net structure, and can be deemed as two instances of the same net structure. For simplicity, we only describe the model for the *DataChannel* connector.

The net inscription of the *DataChannel* model is as follows:

R(*lost*) = (*m'* = *lost*)

R(*corrupt*) = (*m'* = *corrupted*)

R(*transmitted*) = *true*

The corresponding property specification is:

□ (*Sender.DataOut*(*x*)

→ ◇(*Receiver.DataIn*(*y*) ∧ (*y* = *x* ∨ *y* = *lost* ∨ *y* = *corrupted*))

The verification of architecture specification is to validate that the behavior models satisfy the property specifications of the system. We will describe the verification approach in section 4.

3.3 Model of ABP with A Timer

When timeout mechanism is introduced into the ABP, there exists a timing issue for the architecture model. In our approach, the basic idea to express the timing aspect of the data is to associate each token in the PrT net with a timestamp. The timestamp of a token indicates when this token is generated. The time constraint on the firing of a transition is described in the constraints of this transition. Consider the PrT net in *Figure 5*. When T1 fires, a message in P1 is transmitted to P2. If the T1 must fire when it is enabled for a certain time, say 2 time units minimum and 4 time units

maximum, then the constraints on T1 can be expressed as R(*T1*)=($t1 + 2 \leq t2 \leq t1 + 4$). That is, T1 may fire after it has been enabled for 2 time units and must fire if it has been enabled for 4 time units.

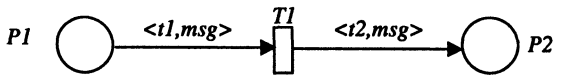

Figure 5. Time constraint on transition

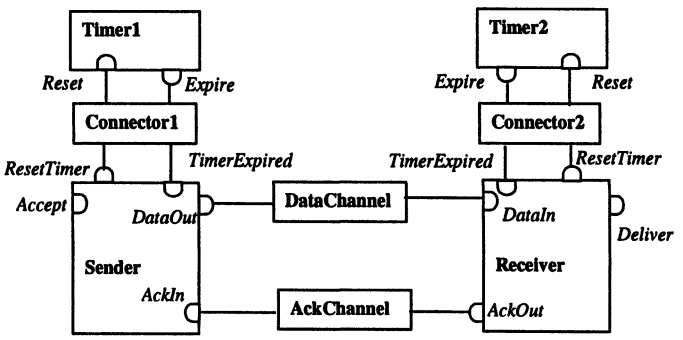

Figure 6. Architecture model of ABP with timer

The architecture model, as shown in *Figure 6*, is similar to the one without timer, except for the additional timers. Instead of resending message when lost ACK is detected, the message is resent when there occurs a timeout signal in this model. The architecture property specifications are the same except for a few differences in the concrete temporal formulas because of the additional timestamp associated to each token. The specifications are as follows:

P(1) $\Box((Accept(<t1, x>) \rightarrow \exists t2.\ \Diamond Deliver(<t2, x>))$

P(2) $\Box((Accept(<t1, x1>) \wedge Accept(<t2, x2>) \wedge t2 > t1)$
$\rightarrow (\neg Deliver(<t4, x2>)\ \mu\ \exists\ t3.Deliver(<t3, x1>)))$

P(3) $((Accept(<t1, x>) \wedge O\neg Accept(<t1, x>)) \rightarrow O\ (\neg(Accept(<t2, y>)$
$\wedge O\neg Accept(<t2, y>))\ \mu\ (\neg Deliver(<t3, z>) \wedge O\ Deliver(<t3, z>))))$
$\wedge((\neg Deliver(<t4, x>) \wedge ODeliver(<t4, x>)) \rightarrow O\ (\neg(\neg Deliver(t5, y)$
$\wedge ODeliver(<t5, y>))\ \mu\ Accept(<t6, z>) \wedge O\neg Accept(<t6, z>))))$

Due to the space limit, we only show the behavioural model of the *Sender* without the net inscription. When an accepted message is allowed to send in sender, it should be the earliest accepted one among all the accepted messages ready to send. The constraint on timestamp of token m' enforces that transition *sendData* will fire in a certain time unit when it is enabled.

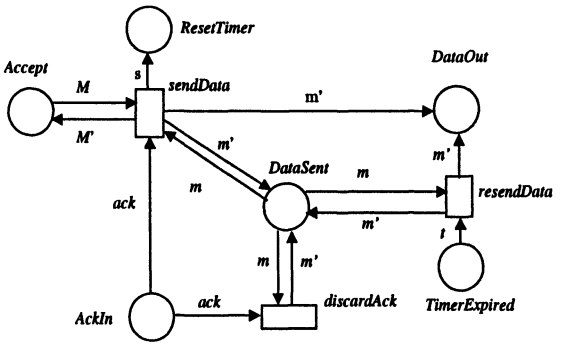

Figure 7. Model of the Sender

The corresponding property specification is:

□ ((*TimeExpired*(*t1*) ∧ *DataBuf*(<*t2*, *x*>))→ ◇*DataOut*(<*t3*, *x*>))

□ ((*AckIn*(<*t1*, *b*>) ∧ *DataBuf*(<*t2*, *b*, x>) ∧*Accept*(<*t3*, *y*>))
→ ◇*DataOut*(<*t4*, 1-*b*, *y*>))

□ ((*AckIn*(<*t1*, *b*>) ∧ *DataBuf*(<*t2*, 1-*b*, *x*>)) → ◇¬*AckIn*(<*t1*, *b*>)

4. ANALYSIS OF ABP MODEL

In this paper, we apply Symbolic Model Checking technique to analyze SAM specification by using SMV (Symbolic Model Verifier) tool. In the following sections, we first take a glance at the SMV tool, and then describe how to translate SAM specification into an SMV program, finally show the whole process by verify the two architecture models described above.

4.1 Rules for Translating Architecture Specifications

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. It uses the OBDD-based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied. To verify a finite system in SMV, it has to be described in SMV input language. Since it is intended to describe finite state systems, the only data types in the language are finite ones –Booleans, scalars and fixed arrays. The complete syntax of the SMV language is described in the SMV documentation [McM93].

An SMV program consists of a definition of a finite state transition systems and a list of properties written in CTL formula. A transition system is defined in terms of a state space, a set of transition relations, and initial states. A SAM architecture specification consists of behavior models defined by PrT nets, and properties specified in linear temporal logic. Therefore, to

translate a SAM specification into an SMV program, we should translate behavior models in PrT nets into the finite transition system in SMV, and properties in linear temporal logic to CTL. The general procedure and rules for translation are as follows:

Step 1: Connecting the behavior models of individual elements

It is straightforward to get the composition level behavior model from the individual behavior models and the architecture.

Step 2: Defining state variables

Declare a state SMV variable for each place in the composition behavior model. For each place p, the declare statement could be:

$$p : 0..(|\varphi(p)|^{bound})$$

where $|\varphi(p)|$ is the number of all distinct value allowed in p, and *bound* is maximum tokens in p. $|\varphi(p)|^{bound}$ indicates the minimum number of values the state variable p should hold. In fact, it is not necessary that the array begin at subscript 0. Instead of using array as the type of state variable, we could also use a list of enclosed values so that the value is more meaningful.

It should be noted that, the type of a place p can only have finite values and each p should be bounded. Otherwise the states for the behavior model will be infinite. As we have known, SMV is applicable only to finite transition system. It is impossible for us to translate a behavior model with infinite states into SMV. However, in most cases, some specific properties still hold after reducing a behavior model with infinite states to one with finite states. Therefore, to verify a specific property, we can reduce some infinite places to finite places as long as the reduction does not affect the verification of this property. For example, the type of place *Accept* in *Figure 2* is infinite, and the message can be any string. When we want to verify the accepted message will be eventually delivered, it does not matter what the message means. In this case, we can reduce the type of *Accept* to a finite type, even a type containing only one value.

Step 3: Defining initial state

Initialize each SMV state variable to the value, which is corresponding to the initial marking of the place in the behavior model, using the INIT statement.

$$INIT = \bigwedge_{p \in P} (p = f(M_0(p)))$$

Where f is the mapping from the initial marking of place p to the value of state variable p.

Step 4: Defining transition relations

Firstly, for each transition in the behavior model, define the transition name as its enabling conditions in DEFINE statement. For a transition t, its corresponding define statement is $t := \bigwedge_{p_i \in {}^\bullet t} (p_i > 0) \wedge R(t)$.

Secondly, for each SMV state variable p, declare a *next* statement as follows:

```
next(p) :=  case
                t_i : f(M(p));      // for each t_i ∈ •p
                t_i : 0;            // for each t_i ∈ p•
                1: p;
            esac;
```

where f is the mapping from the marking of place p to the value of state variable p, after t_i fires.

An alternate method to define the transition relations is to use TRANS instead of ASSIGN statement. Rather than describe transition relations from the perspective of places, we can also describe transition relations in terms of transitions. Similar to the method proposed by Wimmel in [Wim97], the TRANS statement consists of one subformula $TRANS_t$ for each transition t:

$$TRANS_t = t \wedge (\bigwedge_{p \in t^\bullet} next(p) = f(M(p))) \wedge (\bigwedge_{p \in {}^\bullet t - t^\bullet} next(p) = 0) \wedge (\bigwedge_{p \in P - ({}^\bullet t \cup t^\bullet)} next(p) = p)$$

Where f is the mapping from the marking of predicate p to the value of state variable p, after t fires.

In addition, to make in possible to verify a behavior model containing deadlocks, which means there is no enabled transition, a subformula has to be added to the transition relation that allows the system to stay in its current state if there is a deadlock [Wim97]. The symbol *deadlock* is defined as:

$$deadlock := \neg \bigvee_{t \in T} t$$

Thus, the TRANS statement is finally defined as follows:

$$TRANS = \bigvee_{t \in T} TRANS_t \vee (deadlock \wedge \bigwedge_{p \in P} next(p) = p)$$

Step 5: Defining the specifications to be verified

What we need to do here is to translate the properties to be verified in SAM to the SPEC part in SMV. Since specification properties in SAM are described in linear temporal logic, and specifications in SMV are described in CTL, it is also straightforward.

Step 6: Defining fairness constraints if necessary

When fairness assumptions should be made in the behavior model, add fairness constraints in the FAIRNESS part in SMV. For example, there should be a fairness constraint for the channel in ABP so that the ABP system can work successfully.

Step 7: Dealing with Timing constraint when necessary

When timing constraint is applied in the behavior model in SAM, additional codes should be added to the SMV program. Generally, time grows without bound in behavior model, so the underlying state transition system has infinite states and SMV becomes inapplicable. In most cases, however, we don't care about absolute time but relative time. In the ABP model with timer, for example, all the time constraints are about the time difference between tokens. This will make it possible to deal with timing constraint in SMV. What we need to do is to keep track of time lapses with a state variable whose range is bounded. We call this state variable *clock* in our context. The clock does not increase unless there is no enabled transition. When the clock reaches its upper bound, its next time value will be the lower bound. The example for dealing with timing constraint is shown in section 4.3.

4.2 Analysis of ABP without A Timer

Following the general steps and rules described in previous section, it is quite easy to translate the architecture specification to SMV program. Our goal is to verify that the compositional architecture specification satisfies the three properties, P1, P2 and P3, described in section 3.2. To make the underlying transition system to be finite and simplify the analysis, we assume that, there are 8 distinct messages accepted initially. Namely, there are 8 distinct tokens in Place Accept. The *id*s for the messages range from 1 to 8. The messages will be sent by the *Sender* in the order from 8 to 1. No any other message is accepted. We verify that all the 8 messages and only the 8 messages are delivered, and the 8 messages are delivered in the order they are sent by the *Sender*.

With the assumption above, we translate the architecture specification to the SMV program as follows. The figure for connected behavior model is omitted here.

Since $|\varphi(\text{AckBuf})| = 2$ and it is 1-bounded, we can define the state variable *AckBuf* as:

AckBuf: 0..2;

However, to make it more meaningful, we define it as follows instead.

AckBuf: {0, bit0, bit1};

Where state 0 means there is no token in Place *AckBuf*. Similarly, we can define *AckOut* and *AckIn*.

AckOut: {0, bit0, bit1}; AckIn: {0, bit0, bit1, lost, corrupted};

We have noted from the assumption that $|\varphi(AckBuf)|=8$ and it is 8-bounded, so it seems that we should define *Accept* to range between 0 and

8^8. In fact, *Accept* can be presented by 8 values (in addition to 0) since messages are sent in order. So we define *Accept* as:

Accept: 0..8;

Where state *0* means there is no token in Place *Accept*, *i* (*i*>0) means that there are *i* tokens in *Accept* and the id for each message is 1 to *i* respectively. Similarly, Deliver can be defined as:

Deliver: 0..8;

To make it more convenient to extract message id and associated bit tag, we define the remaining two variables as follows:

DataBuf: 0..17; //0: empty, 1: undefined, 2-17: message_id *2 + *bit*

DataOut: 0..17;

DataIn: 0..19; //18-lost, 19-corrupted

After the state variables are defined, the initial state and transition relations are straightforward. After the state variables are defined, the initial state and transition relations are straightforward. In the SPEC part we use formula *AF Deliver = 8* to verify property P1, and formula *AG (deliverData & !next(deliverData) → DataIn/2 = Deliver + 1)* to verify P2 and P3.

If we run the SMV program now, the first formula is false. This is because we have not considered the fairness constraints. ABP works successfully only when the channels don't always lose or corrupt messages. Therefore we should add fairness constraints on the SMV program to make sure that a message will get through the channel when it is sent infinitely often. The fairness constraints to guarantee this are as follows:

AF (DataIn >0 & DataIn <18)

AF (AckIn = bit0 | AckIn = bit1)

After the fairness constraints are added, the specification is evaluated to be true.

4.3 Analysis of ABP with Timer

The analysis of ABP with timer is similar to the analysis without timer shown in the previous section, except for the timing constraints. Therefore, we focus mainly on the timing constraints in this section. Instead of verifying all the three properties shown in section 3.3, we just verify here that an accepted message will eventually delivered. We assume that no new message can be accepted until the sender receives proper ACK and ready to accept new message.

To deal with the timing constraints, a clock is defined as follows:

clock: 1..20;

The clock advances only when there is no mature transition and the Sender is not ready to accept new message. That is, the clock stops while the

Sender is waiting to accept new message. The initial value of clock is 1, and its transition relation is as follows:

```
next(clock) := case
    noMatureTrans & !waitingforAccept & clock < 20: clock + 1;
    noMatureTrans & !waitingforAccept & clock = 20; 1;
    1: clcok;
  esac;
```

Where *noMatureTrans* and *waitingforAccept* are defined as:

```
NoMarureTrans := ! (sendData | resendData | discardData | discardAck |
                    dataTransit | AckTransit | deliverData | resendAck |
                    R.beginTiming | R.clearTiming | R.TimeExpires |
                    S.beginTiming | S.clearTiming | S.TimeExpires );
waitingforAccept := AckIn > 0 & Accep t= 0 & DataBuf > 0
                    & clock != AckIn/2 & clock != DataBuf/2
                    & (AckIn mod 2) = (DataBuf mod 2)
```

Again, fairness constraints need to be added to make the model work successfully.

$AF\ (DataIn > 0)$

$AF\ (AckIn > 0)$

And the SPEC to verify is:

$AG\ (Accept \rightarrow AF\ Deliver)$

4.4 Summary of Running Results

Our SMV programs were executed on Sun-Blade-1000 running SunOS 5.8, using the SMV Release 2.5.4. SMV reported the resources needed to analyze the properties. For the first SMV program, which was a quite simple one, 10,412 BDD nodes and about 1.3M memories were allocated, and it was finished in 0.06 second. The size of the state space is about 2.3×10^7, and the reachable states are 129. For the second SMV program, about 1.8×10^6 BDD nodes and 300M memory were allocated, and it was finished in 315 seconds. The size of the state space is 7.8×10^{24} and the number of reachable states is 2.1×10^4.

It should be noted that when the properties are evaluated false, for example, when we remove the fairness constraints in the SMV program, it takes SMV a little more time to find a counterexample.

5. RELATED WORK

Several papers studied the analysis of alternating bit protocol. In [Suz90], an ABP without timer was analyzed using temporal Petri nets, which are low-level Petri nets with certain restriction on the firings of transitions

represented by temporal formulas. However, the analysis technique using ω-regular expressions may not always be possible or straightforward, and may require additional techniques when a given formula is complex. In addition, using low-level Petri nets makes it difficult in representing values of data items. An ABP with timer was represented in terms of labeled transition systems (LTS) in [GKM98], and its properties were analyzed by an exhaustive search of state space of the LTS model. But the paper did not model timeout mechanism and simply treated it as external event.

In [Wim97], several methods to represent a low-level Petri net using SMV were proposed and compared, which were helpful for developing our approach to translate the architecture specifications into SMV programs. W. Chan et al. applied model checking to the analysis of software specification in [CAB98]. The main difference between their work and ours was the language used for representing software architecture specifications. They used RSML, a state-machine language based on statecharts.

6. CONCLUSION

We have presented how to model timing issues in SAM and applied symbolic model checking technique to validate the architecture specification in SAM. General rules to translate architecture specification in SAM into the SMV language were identified and proved to be powerful. The satisfying results indicate symbolic model checking is an effective technique in the process of analyzing the architecture specifications in SAM.

In this paper, we connected the behavior models of components and connectors into a single composition level behavior model, before applying model checking. This approach works in most situations due to the high-level abstraction of a software architecture specification. However, the connected composition level behavior model can be quite large in some situations to prevent the effective use of symbolic model checking technique. Future work could focus on compositional model checking techniques.

Acknowledgements

This work is support in part by the *NSF* under grant HDR-9707076 and by the *NASA* under grant NAG 2-1440.

References

[AAG95] G. Abowd, R. Allen, and D. Garlan: "Formalizing style to understand descriptions of software architecture", *ACM Trans. on Software Engineering and Methodology*, vol.4, 1995, 319-364.

[AG97] R. Allen, and D. Garlan: "A formal basis for architectural connection", *ACM Trans. on Software Engineering and Methodology*, vol.6, 1997, 213-249.

[AL94] M. Abadi and L. Lamport: "An Old-Fashioned Recipe for Real Time", *ACM Transactions on Programming Languages and Systems*, vol.16, no.5, 1994, 1543-1571.

[BD91] B. Berthomieu and M. Diaz: "Modeling and verification of time dependent systems using time Petri nets", *IEEE Trans. on Software Eng.*, vol.17, no.3, 1991.

[CAB98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. Reese, "Model Checking Large Software Specification," IEEE Transactions on Software Engineering, vol. 24, no. 7, July 1998, 498-520,

[CW96] E. Clarke and J. Wing: "Formal methods: state of the art and future", *ACM Computing Surveys*, vol.28, 1996, 626-643.

[EMS92] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasian: "Quantitative temporal reasoning", *Real-Time Systems*, 4:331-352, 1992.

[GKM98] D. Giannakopoulou, J. Kramer and J. Magee, "Behaviour Analysis based on Software Architecture," International Workshop on the Role of Software Architecture in Testing and Analysis, Sicily, Italy, 1998.

[GMM91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze: "A unified high-level Petri net formalism for time-critical systems", *IEEE Trans. On Software Engineering*, vol.17, no.2, 1991, 160-172.

[HD00] X. He and Y. Deng: "Specifying software architectural connectors in SAM", *Intl. Journal of Software Engineering and Knowledge Engineering*, vol.10, 2000, 411-432.

[HD02] X. He, Y. Deng, "A Framework for Developing and Analyzing Software Architecture Specifications in SAM," *The Computer Journal*, vol.45, no.1, 2002, 111-128.

[HL90] X. He and J.A.N. Lee: "Integrating predicate transition nets and first order temporal logic in the specification of concurrent systems", *Formal Aspects of Computing*, vol.2, 1990, 226-246.

[Hoa85] C. Hoare: *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[IW95] P. Inverardi and A. Wolf: "Formal specification and analysis of software architectures using the chemical abstract machine model", *IEEE Transaction on Software Engineering*, vol.21, 1995, 373-386.

[McM93] K. McMillan, Symbolic Model Checking, Kluwer Academic Publisher, 1993.

[MP92] Z. Manna and A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems – Specification*, Springer-Verlag, Berlin, 1992.

[MT00] N. Medvidovic and R. Taylor: "A classification and comparison framework for software architecture description languages", *IEEE Transaction on Software Engineering*, vol. 26, 2000, 70-93.

[Spi92] J. Spivey: Z Reference Manual (2nd ed.), Cambridge University Press, U.K, 1992.

[Suz90] Ichiro Suzuki, "Formal Analysis of the Alternating Bit Protocol by Temporal Petri Nets," IEEE Transactions on Software Engineering, vol. 16, no. 11, 1990, 1273-1281.

[WHD99] J. Wang, X. He and Y. Deng: "Introducing software architecture specification and analysis in SAM through an example", *Information and Software Technology*, vol. 41, 1999, 451-467.

[Wim97] G. Wimmel, "A BBD-based Model Checker for the PEP tool," Major Individual Project, Department of Computer Science, University of Newcastle, 1997.