

# Compositional Approach for Modeling and Verification of Component-Based Software Systems

Jeffrey J.P. Tsai and Eric Y.T. Juan  
Department of Electrical Engineering and Computer Science  
University of Illinois, Chicago  
851 S. Morgan St., Chicago, IL 60607  
Email: tsai@eecs.uic.edu

## Abstract

With the rapid growth of networking and high-computing power, the demand of larger and more complex software systems has increased dramatically. To deal with the complexity in designing large-scale complex software systems, the concept of component-based software design has gained popularity recently. However, in pursuing a component-based approach there are obstacles to be overcome. One of them is the state-explosion problem in the formal verification of large-scale component-based systems. In this paper, we introduce a modeling technique and two condensation theories to model and verify component-based software systems. Our condensation theories are much weaker than current theories useful for the compositional verification. More significantly, our new condensation theories can eliminate the interleaved behaviors caused by asynchronously sending actions. Therefore, our technique provides a much more powerful means for the compositional verification of asynchronous processes. Our technique can efficiently analyze several state-based properties: deadlock state and reachable state. The experimental results show a significant improvement in the analysis of large-scale component-based systems.

## 1 INTRODUCTION

With the rapid growth of networking and high-computing power, the demand of larger and more complex software systems has increased dramatically. Examples include web-based systems, multimedia systems, telecommunication systems, intelligent agents systems, flight control systems, patient monitoring systems, robotics, virtual reality systems, and so on. However, the development of large-scale and complex software systems is much more difficult and error prone. This is due to the fact that techniques and tools for assuring the correctness and reliability of software systems lag far behind the increasing growth of size and complexity of software systems. The results are unreliable and poorly performing applications, delayed projects, and considerable cost overruns. In order to improve the usability and reliability of large-scale systems, the supporting techniques and development tools need to be greatly enhanced.

Formal verification is rapidly becoming accepted as a promising and automated method to verify the correctness of software systems. Despite many works in the area of formal verification, one of main bottlenecks so far is the state explosion problem. When the size of a software

system increases linearly, the analysis complexity of the system could grow exponentially. The capability and performance of current techniques still can not efficiently verify typical large-scale software systems in practice.

A technique “*compositional verification*”, which is considered more suitable for analyzing well-defined subsystems, such as component-based systems, has been proposed by researchers to deal with the state-explosion problem of large-scale systems. However, current compositional verification techniques are efficient only for verifying event-based properties of synchronous processes. In this paper, we introduce a modeling technique and two condensation theories to model and verify component-based software systems. Our condensation theories are much weaker than current theories useful for the compositional verification. More significantly, our new condensation theories can eliminate the interleaved behaviors caused by asynchronously sending actions. Therefore, our technique provides a much more powerful means for the compositional verification of asynchronous processes. Our technique can efficiently analyze several state-based properties: deadlock state and reachable state. The experimental results show a significant improvement in the analysis of large-scale component-based systems.

## 2 THE MODEL

This section introduces a model, namely *multiset labeled transition systems* (MLTSs in the sequel). MLTSs are closely related to *labeled transition systems* (LTSs for short) which are intensively used as a state-space model in the family of process algebras. There are three distinguishing features between MLTSs and traditional LTSs. First, the label of a transition is a multiset of actions in MLTSs instead of one action in LTSs. Second, we make a clear distinction between synchronously communicating actions and asynchronously communicating actions. Third, the composition of traditional LTSs is for synchronous processes only, while the composition of both synchronous and asynchronous processes can be achieved in MLTSs. These features of MLTSs promise the development of a new mechanism for compositional verification. With the use of the new mechanism, the high analysis complexity of large-scale systems can significantly be reduced.



Figure 1: MLTS specification of a server.

The behavior of a process can be modeled by an MLTS. An MLTS consists of *states*, *transitions*, *actions*, and one *initial state*. In graphical representation of MLTSs, a *state* is denoted by a shaded circle, and a *transition* by a solid arrow labeled with *actions*. The *initial state* is pointed out by a dotted arrow.

A *state* in MLTSs could be interpreted as a condition. The states of an MLTS describe the possible conditions of a process. For instance, Figure 1 gives an MLTS which specifies a server. The condition of *SERVER* is either ready to accept a request (state *Ready*) or busy in retrieving data (state *Retrieve-data*). The state of an MLTS is changed by the execution of *actions*. Once *SERVER* performs the action of receiving request (*Receive-Req*), the condition of *SERVER* is

changed from state *Ready* to state *Retrieve-data*. The condition of *SERVER* returns to state *Ready* after *SERVER* sends out data (action *Send-data*). An *initial state* is the condition at the beginning. The initial state of *SERVER* is *Ready*.

The relationship of *states* and *actions* is represented by *transitions*. A *transition* contains a *starting state*, one or more *actions*, and an *ending state*. For example, transition (*Ready*, *Receive-Req*, *Retrieve-data*) indicates that state *Retrieve-data* is *reachable* from state *Ready* by the execution of action *Receive-Req*. In the following, we first give the formal definition of MLTSs.

### Definition 2.1 Multiset Labeled Transition Systems (MLTSs)

- A **multi-set**  $MS$  consists of a set  $D_{MS}$  and a mapping  $MS: D_{MS} \rightarrow N$ , where  $N = \{1,2,3,\dots\}$  is the set of positive integers.  $MS$  is said to be a **multi-set** of a set  $S$  iff (if and only if)  $D_{MS} \subseteq S$ .
- A **MLTS** is a quadruple  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$ , where
  - 1)  $\mathbf{S}$  is a set of **states**;  $\mathbf{s}_{in}$  is the **initial state**;
  - 2)  $\Sigma_\tau$ , which is a set of **actions** (transition labels), comprises **invisible** (or **internal**) **action** ( $\tau$ ) and **communicating actions**  $\Sigma$ , where  $\Sigma$  consists of  $\Sigma_{syn}$  (**synchronously communicating actions**),  $\Sigma_{as}$  (**asynchronously sending actions**), and  $\Sigma_{ar}$  (**asynchronously receiving actions**); and
  - 3)  $\mathbf{T} \subseteq \mathbf{S} \times \mathbf{M}_{\Sigma_\tau} \times \mathbf{S}$  is a set of **transitions** such that  $\forall (s, m_s, s') \in \mathbf{T}$ :  $m_s$  is a non-empty *multiset* of  $\Sigma_\tau$ .

In MLTSs, a transition is labeled with a multiset of actions. A multiset consists of countable objects. This means that an action can have multiple instances in a transition label. We use *synchronous communication* and/or *asynchronous communication* as the primitive means of communication between processes. *Synchronously communicating actions* ( $\Sigma_{syn}$ ) in MLTSs are used to specify unbuffered mode of synchronous communication which is usually referred to as handshaking or rendezvous communication. Synchronous communication between processes is performed through a simultaneous execution of actions which read from or write to a shared channel. In other words, actions which read from or write to a shared channel, have to take place at the same time.

The semantics of asynchronous interaction of MLTSs is essentially the same as asynchronous message passing used in NIL [5] and PLITS [2]. In asynchronous communication, a sending process is not blocked to wait for its communicating partners. Once a message is ready, the process is free to perform its asynchronously sending action. Messages which have not been received by receivers are stored in the buffers of channels. In MLTSs, the channel buffers of asynchronous communication have unbounded capability. An asynchronously receiving process, as in the case of synchronous communication, may be blocked in order to wait for messages. Note that channels for synchronous communication have no buffer for the storage of messages, since synchronous communication between processes has to take place at the same time.

## 3 DEADLOCK STATE AND REACHABLE STATE

In this section, we focus on the properties of deadlock states and reachable states. A state is said to be *reachable* in an MLTS if the state can be reached from the initial state via directed edges. Recall that a *state* in MLTSs could be interpreted as a condition. Thus, a *reachable state* can be considered as a possible condition that a process or system can reach.

A system is said to be *deadlocking* if the system has reached a condition (state) such that the system cannot do anything. In practice, a *deadlock state* reflects a failure or successful termination of a system. In order to distinguish failure from successful termination, we preserve the conditions of deadlocking systems, i.e., *deadlock states*. From the inspection of *deadlock states*, we can easily determine whether a deadlocking system fails or successfully terminates. In addition, a *deadlock state*, which provides detailed conditions of the whole system, is useful for debugging and modifying an improper system design. The following defines *reachable states* and *deadlock states* in MLTSs.

**Definition 3.1 (Reachable States)**

Let  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$  be an MLTS. A state  $s$  is *reachable* in  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$  iff

- 1)  $s = \mathbf{s}_{in}$  or
- 2)  $\exists (s_0, m_1, s_1) \dots (s_{n-1}, m_n, s_n)$  such that
  - i)  $n \geq 1$ , ii)  $s_0 = \mathbf{s}_{in}$ , iii)  $s_n = s$ , and iv)  $\forall 1 \leq j \leq n: (s_{j-1}, m_j, s_j) \in \mathbf{T}$ .

**Definition 3.2 (Deadlock States)**

Let  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$  be an MLTS. A state  $s$  is a *deadlock state* of  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$  iff

- 1)  $s$  is *reachable* in  $(\mathbf{S}, \Sigma_\tau, \mathbf{T}, \mathbf{s}_{in})$  and 2)  $\nexists (s, m, s') \in \mathbf{T}$  ( $s$  has no outgoing transition).

## 4 CONDENSATION THEORIES FOR MLTSs

This section presents our newly derived *congruence theories* for the compositional verification of deadlock states and reachable states. We call our congruence theories *IOT-failure equivalence* and *IOT-state equivalence*. *IOT-failure equivalence* preserves the property of deadlock states while *IOT-state equivalence* preserves the property of reachable states. We will explain these two *congruence theories* using simple examples.

### 4.1 Paths and Input/Output-Traces (IO-Traces)

The computation of an MLTS can be described in terms of paths. A path is an alternating sequence of states and transitions in MLTSs. For example, MLTS  $P_7$  in Figure 2 has a path

$$\sigma_7 = \{s_0, (s_0, Ch_1, s_1), s_1, (s_1, Ch_2 !, s_2), s_2, (s_2, Ch_3 !, s_3), s_3, (s_3, Ch_4, s_4), s_4\}.$$

For simplicity, we also write path  $\sigma_7$  as

$$\{s_0, Ch_1, s_1, Ch_2 !, s_2, Ch_3 !, s_3, Ch_4, s_4\}.$$

Similarly, MLTS  $P_{7-C}$  in Figure 2 has a path

$$\sigma_{7-C} = \{s_0, (Ch_1, Ch_2 !, Ch_3 !), s_3, Ch_4, s_4\}.$$

Path  $\sigma_7$  means that if the local condition of MLTS  $P_7$  is state  $s_0$ , then  $P_7$  is ready to sequentially execute actions  $\{Ch_1, Ch_2 !, Ch_3 !, Ch_4\}$  and to sequentially reach states  $\{s_1, s_2, s_3, s_4\}$ . However, the execution along path  $\sigma_7$  may fail due to some condition outside MLTS  $P_7$ , i.e., the environment condition of  $P_7$ . In other words, for the occurrence of a path to be successful, we need to consider the global condition which consists of a local condition and an environment condition.

We use *IO-traces* in order to deduce and compare the global conditions required for the occurrences of paths (actions). *IO-traces* are derived from paths by removing some details which are irrelevant to the success of paths' occurrences. Based on *IO-traces*, we have developed *IOT-failure equivalence* and *IOT-state equivalence* as presented in the following sections.

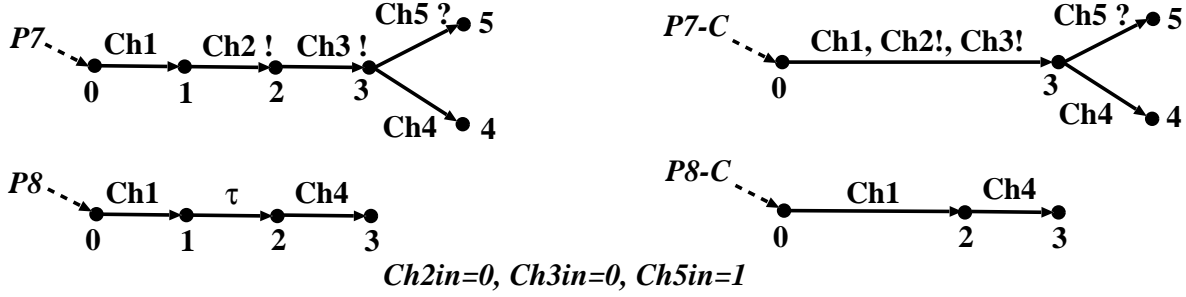


Figure 2: Example of paths and IO-traces.

An *IO-trace* consists of i) a starting state, ii) an ending state, and iii) a sequence of multisets of actions. Informally speaking, an *IO-trace* is derived from a path by

- 1) removing intermediate states,
- 2) replacing a transition with two ordered elements (multisets), i.e., i) its *environment pre-condition* and then ii) its *environment post-condition*,
- 3) removing empty multisets of actions, and
- 4) summing up adjacent multisets of *environment post-conditions*.

The *environment pre-condition* of a transition consists of *synchronously communicating actions* and *asynchronously receiving actions* which are labeled on the transition. The *environment post-condition* of a transition is represented by *asynchronously sending actions* labeled on the transition. The execution of *invisible actions* does not interact with the environment. Therefore, *invisible actions* are not included in *IO-traces*. Let us consider MLTSs in Figure 2 as an example. Assume that  $Ch_1$  and  $Ch_4$  are synchronous communication channels between processes  $P_7$  and  $P_8$ . It is clear that the *environment pre-condition* of transition  $(s_0, Ch_1, s_1)$  is  $\{Ch_1\}$ ; the *environment post-condition* of transition  $(s_0, Ch_1, s_1)$  is null. Now let us derive the *IO-trace* for the path  $\sigma_7 = \{s_0, Ch_1, s_1, Ch_2 !, s_2, Ch_3 !, s_3, Ch_4, s_4\}$  in MLTS  $P_7$  above. After 1) removing intermediate states  $s_1, s_2$  and  $s_3$ , and 2) replacing a transition with its *environment pre-condition* and then its *environment post-condition*, we get

$$s_0, \{Ch_1\}, \{\}, \{\}, \{Ch_2 !\}, \{\}, \{Ch_3 !\}, \{Ch_4\}, \{\}, s_4.$$

After that, we remove empty multisets of actions and sum up adjacent multisets of *environment post-conditions* (or *asynchronously sending actions*). As a result, the *IO-trace* of path  $\sigma_7$  is

$$IOT\sigma_7 = s_0, \{Ch_1\}, \{Ch_2 !, Ch_3 !\}, \{Ch_4\}, s_4.$$

Similarly, we can derive the *IO-trace* of path  $\sigma_{7-C} = \{s_0, (Ch_1, Ch_2 !, Ch_3 !), s_3, Ch_4, s_4\}$  in MLTS  $P_{7-C}$  above as

$$IOT\sigma_{7-C} = s_0, \{Ch_1\}, \{Ch_2 !, Ch_3 !\}, \{Ch_4\}, s_4.$$

From paths  $\sigma_7$  and  $\sigma_{7-C}$ , we can see that two different paths might have an identical *IO-trace*. *IO-traces* are useful for predicting the successful occurrences of paths (or actions). Based on *IO-traces*, *IOT-failure equivalence* and *IOT-state equivalence* will be presented in the following sections.

## 4.2 IO-Trace Failures (IOT-Failures)

In order to efficiently analyze a large-scale system, it is desirable to eliminate data which are irrelevant to the verification of interesting properties. Our *IOT-failure equivalence* is developed for the compositional verification of deadlock states. This means that *IOT-failure equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any deadlock state. *IOT-failure equivalence* is very useful for reducing the size (complexity) of MLTSs when we focus on the property of deadlock states.

Informally speaking, an *IOT-failure* for an MLTS is a pair consisting of i) an *IO-trace*  $t$  starting from the initial state and ii) the set of *environment pre-conditions* for the outgoing transitions of the ending state of  $t$ . In addition, the ending state of the *IO-trace*  $t$  above should be *stable*.

A state  $s$  is said to be *stable* iff  $s$  does not have any out-transition whose *environment pre-condition* is empty; otherwise  $s$  is *non-stable*. In other words, only *stable states* are candidates for the construction of deadlock states, because a *non-stable* state has at least one out-transition which is guaranteed to be executable in any condition of the environment.

Let us consider MLTS  $P_7$  in Figure 2 as an example. In MLTS  $P_7$ , both transitions  $(s_1, Ch_2 !, s_2)$  and  $(s_2, Ch_3 !, s_3)$  have an empty *environment pre-condition* because they execute neither *synchronously communicating action* nor *asynchronously receiving action*. Therefore, states  $s_1$  and  $s_2$  are *non-stable*. In contrast, states  $s_0, s_3, s_4,$  and  $s_5$  are *stable*. These *stable states* are reached from the initial state  $s_0$  via the following *IO-traces* respectively:

$$\begin{aligned} IOT\sigma_0 &= s_0, \phi, s_0, \\ IOT\sigma_3 &= s_0, \{Ch_1\}, \{Ch_2 !, Ch_3 !\}, s_3, \\ IOT\sigma_4 &= s_0, \{Ch_1\}, \{Ch_2 !, Ch_3 !\}, \{Ch_4\}, s_4, \text{ and} \\ IOT\sigma_5 &= s_0, \{Ch_1\}, \{Ch_2 !, Ch_3 !\}, \{Ch_5 ?\}, s_5. \end{aligned}$$

From these *stable states* and *IO-traces*, we get four *IOT-failures* in MLTS  $P_7$ :

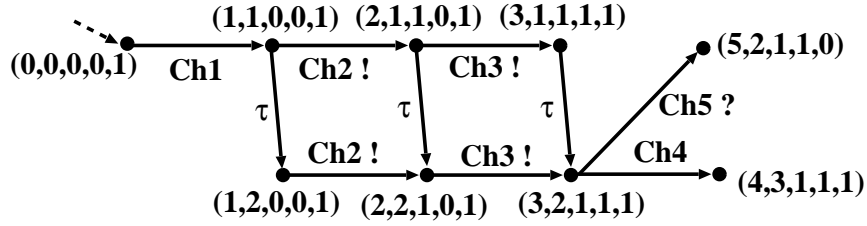
$$\begin{aligned} IOTF_0 &= (IOT\sigma_0, \{Ch_1\}), \\ IOTF_3 &= (IOT\sigma_3, \{\{Ch_4\}, \{Ch_5 ?\}\}), \\ IOTF_4 &= (IOT\sigma_4, \phi), \text{ and} \\ IOTF_5 &= (IOT\sigma_5, \phi). \end{aligned}$$

*IOT-failures*  $IOTF_4$  and  $IOTF_5$  have an empty set of *environment pre-conditions* since states  $s_4$  and  $s_5$  have no out-transition. State  $s_3$  has two out-transitions  $(s_3, Ch_4, s_4)$  and  $(s_3, Ch_5?, s_5)$ . Transition  $(s_3, Ch_4, s_4)$  has an *environment pre-condition*  $\{Ch_4\}$ , and transition  $(s_3, Ch_5?, s_5)$  has an *environment pre-condition*  $\{Ch_5 ?\}$ . Therefore, the set of *environment pre-conditions* of *IOT-failure*  $IOTF_3$  is  $\{\{Ch_4\}, \{Ch_5 ?\}\}$ .

Two MLTSs  $L_1$  and  $L_2$  are said to be *IOT-failure equivalent* iff  $L_1$  and  $L_2$  have the same set of *IOT-failures*. For instance,  $P_{7-C}$  in Figure 2 also has four *IOT-failures*  $IOTF_0, IOTF_3, IOTF_4,$  and  $IOTF_5$ . Thus, MLTSs  $P_7$  and  $P_{7-C}$  in Figure 2 are *IOT-failure equivalent*. Similarly, MLTSs  $P_8$  and  $P_{8-C}$  in Figure 2 are *IOT-failure equivalent* as well.

*IOT-failure equivalence* is a congruence in terms of deadlock states. Therefore, *IOT-failure equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any deadlock state. For example, from the MLTSs in Figure 2, we can compose two MLTSs as shown in Figure 3. Without verifying these two MLTSs in Figure 3, we can guarantee that they have the same set of deadlock states, i.e.,  $s_{(5,2,1,1,0)}$  and  $s_{(4,3,1,1,1)}$ .

$P7 \parallel P8$  ( $Ch2in=0, Ch3in=0, Ch5in=1$ )



$P7-C \parallel P8-C$  ( $Ch2in=0, Ch3in=0, Ch5in=1$ )

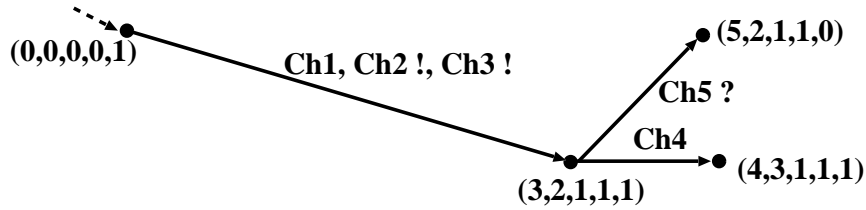


Figure 3: Deadlock-state equivalent MLTSs.

### 4.3 IO-Trace States (IOT-States)

*IOT-state equivalence* is developed for the compositional verification of reachable states. An *IO-trace state (IOT-state)* is an *IO-trace* starting from the initial state. Two MLTSs are said to be *IOT-state equivalent* if they have the same set of *IOT-states*. *IOT-state equivalence* is a *congruence* in terms of reachable states. This means that *IOT-state equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any reachable state.



Figure 4: Example of *IOT-states* and *IOT-state equivalent* MLTSs.

As a simple example, let us consider MLTSs  $P9$  and  $P9-C$  in Figure 4. Assume that  $Ch_1$  and  $Ch_2$  are synchronous communication channels. States  $s_1$ ,  $s_2$ , and  $s_3$  in MLTS  $P9$  are concisely represented by a macro state in MLTS  $P9-C$ . We can see that MLTSs  $P9$  and  $P9-C$  are *IOT-state equivalent* because they have the same set of *IOT-states*:

$$\begin{aligned}
 IOTS_0 &= s_0, \phi, s_0, \\
 IOTS_1 &= s_0, \{Ch_1\}, s_1, \\
 IOTS_2 &= s_0, \{Ch_1\}, s_2, \\
 IOTS_3 &= s_0, \{Ch_1\}, s_3, \text{ and} \\
 IOTS_4 &= s_0, \{Ch_1\}, \{Ch_2\}, s_4.
 \end{aligned}$$

## 5 CONCLUSION

This paper presents a new modeling technique and two new condensation theories to reduce the state explosion problem of asynchronous processes as well as synchronous processes in component-based software systems. Our condensation technique has reasonable complexity (polynomial in the numbers of states and transitions). From the experimental results, our technique promises a much more efficient analysis, especially for asynchronous processes in distributed systems. The condensation theories can be applied to Petri nets model too [6]. The current version of our technique focuses on the analysis of deadlock states and reachable states. Nevertheless, we believe that a more elaborated extension can be used to verify many other important safety and liveness properties of distributed systems, such as accessibility and event sequences. An extension of our work is currently under study.

## 6 ACKNOWLEDGMENTS

This research was supported in part by NSF and DARPA under Grant CCR-9633536.

## References

- [1] S. Brookes, C. Hoare, and A. Roscode, "A theory of communicating sequential processes," *ACM* 31, 3, pp. 560-599, 1984.
- [2] J.A. Feldman, "A programming methodology for distributed computing (among other things)," *Communication ACM* 22, pp 353-368, 1979.
- [3] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
- [4] R. Milner, "Operational and algebraic semantics of concurrent processes," *Handbook of theoretical computer science*, ed. J. van Leeuwen, Elsevier Science Publisher B.B., 1990.
- [5] R.E. Strom and N. Halim, "A new programming methodology for long-lived software systems," *IBM J. Res. Devel.* 28, pp. 52-59, 1984.
- [6] Y. T. Juan, J. J. P. Tsai, and T. Murata, "Compositional Verification of Concurrent Systems Using Petri-Nets-Based Condensation Rules," *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 5, pp. 917-979, Sept. 1998.