Incremental Architectural Modeling and Verification of Real-Time Concurrent Systems[1]

Yi Deng, Jiacun Wang and Rakesh Sinha
School of Computer Science
Florida International University
Miami, FL 33199
e-mail: {deng, wangji, sinha}@cs.fiu.edu

Abstract

An incremental approach for architectural modeling and analysis of real-time concurrent systems is presented. The approach integrates existing formal methods, more specifically time Petri nets and real-time computational tree logic, and leverages their complementary strengths in a way that allows us to systematically enforce that architectural design meets the system's timing requirements, and to incrementally verify the conformance. Consequently, our approach is able to provide better assurance to system design and yet reduce the complexity of analysis. Our approach is based on a Real-time Architectural Specification (RAS) model, which provides a formal basis to systematically maintain a correlation between the (timing) requirements of a system and its architectural design. Based on RAS, we further present a method to verify timing properties of a system design. This method helps conquer the complexity of analysis in two dimensions. Horizontally at each design level, incremental verification is achieved by introducing TPN reduction rules that allow us to compose analysis results on individual system components. Vertically across design levels, incremental verification is achieved by propagating higher-level constraints to lower-level designs so that we can safely plug a component design into a high-level architecture without having to re-verify the entire model. A naval command and control (C2) system is used throughout the paper to demonstrate the concept and usability of our approach.

1. This work was supported in part by the NSF under Grant No. HDR-9707076, by Air Force Office of Scientific Research under Grant No. F49620-96-1-0221, and by Army Research Office under grant No. DAAG55-98-1-0428. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements either expressed or implied by the above named Agencies.

1. Introduction

Real-time systems must not only behave functionally correct, but also satisfy real-time constraints. Examples of such systems include air traffic control systems, flexible manufacturing systems, and military command and control systems. Due to the complexity and stringent time dependent requirements, these systems are extremely difficult and costly to develop.

To ensure the correctness of real-time systems, many formal methods have been developed over the past two decades, which help remove ambiguity and inconsistency in the real-time system development process. The majority of the known formalisms can be classified into two broad classes, namely, operational (or model-based) and descriptive (or property-based). The former defines a system through explicitly representing and manipulating its states or behavior, while the latter describes what properties the system must satisfy without referring to the system's states explicitly. These two classes of formalisms have relative strengths and merits against each other. Operational formalisms are better suited to describe systems as abstract programs, which can be formulated as parallel composition of subsystems. They are cumbersome for describing rules and constraints. Descriptive formalisms are more appropriate for specifying requirements and rules, but does not reflect the component interaction view relevant for design level specification of distributed systems [17]. Petri nets [14] are a well-known example of operational formalisms and temporal logic [13] is a well-known example of descriptive formalisms.

Although formal methods have been extensively studied in the past two decades, their applications to real-world systems only have occurred in recent years [3] and their wide acceptance as a practical and effective software development approach still remains to be seen. Among many reasons, formal models are difficult to create, change and understand. The majority of published results on formal methods only used very small toy problems to demonstrate the intrinsic features of the formal models and the associated approaches and techniques, but failed in showing how to scale the techniques to realistic sized problems. Comparatively speaking, there have been far more research efforts and papers addressing theoretical issues than how to incorporate theories into software engineering methods. There is a lack of systematic ways to show how these techniques can work together in complex projects. Although many of the above problems are well-documented in [3], research efforts towards representing and formalizing system development processes have only emerged in the recent years.

The paper aims to address some of the above problems with a formal systematic approach (a Real-time Architectural Specification model – RAS in the sequel) to support real-time system development. RAS is built on complementary dual formalisms – time Petri nets [2] (TPN in the sequel) for specifying system behavior and real-time computation tree logic [6] (RTCTL in the sequel) for defining system constraints in a way that facilitates the compositional analysis of system properties. RAS supports a hierarchical component-based real-time system development approach that provides system scalability and maintainability, and ensures the consistency of system constraints among related components at the same abstraction level as well as different abstraction levels. By integrating TPN and RTCTL into a coherent architectural model, RAS provides a formal approach to systematically maintain time-critical system constraints during architecture refinements. Therefore, RAS not only facilitates the design traceability (allows us to determine the acceptable behavior of a system component with regard to the time constraint at any design level), but also supports a flexible and evolutionary design process such that system components can be refined and analyzed incrementally based on the constraints imposed on them.

The advantages of using complementary dual formal methods have been known for a long time. Many research efforts exploring approaches in integrating complementary dual methods have been published, especially those integrating Petri nets and temporal logic ([8], [12]). However most of the approaches do not support hierarchical component-based development. On the other hand, there are research efforts specifically dealing with hierarchical and compositional formal specifications. The approaches most closely related to our work are those based on Petri nets: including hierarchical colored Petri nets [10] and hierarchical predicate transition nets [9], and several object-oriented Petri net models ([1], [11], [7]). However none of the above approaches provides the refinement and/or composition mechanisms as general as our approach. More specifically, (1) the refinement unit is a single net element (either a place or a transition) in the above hierarchical Petri net models, but can be a complete Petri net in our model; and (2) the composition unit is a class in all of the above object-oriented Petri net models, but can be multiple related classes in our model.

2. Overview of RAS Model

An RAS architectural model consists of three basic elements: component (operational) models, connections, and architectural constraints (constraints). The component models specify the real-time behavior and communication interface of the components or subsystems; the connections specify how the components interact with each other; and the constraints define real-time system requirements imposed on the components and connections. All connections are defined using only communication interfaces. In this way, the architecture can be viewed as a template for system composition, that is, a specific design (or sub-architecture) of a component can be "plugged-into" the place of the corresponding specification to form an instantiated, recursively defined multi-level architectural model.

The framework of RAS is illustrated at a more detailed level in Figure 1. In the RAS architectural model, a component (e.g., C) model is described as a time Petri net (TPN). The model has two parts: (1) *communication ports* (denoted graphically by half circles), including *input ports* (e.g., PORT6) and *output ports* (e.g., PORT7), and (2) a net that describes the time-dependent, operational behavior of C, that is, it defines the semantics associated with the ports. The communication between C and its environment is solely through the ports. Such an interface specification not only promotes information hiding, but more importantly, allows any sub-architecture that conforms to the component specification to be "plugged into" the place of the specification to form a multi-leveled, and more detailed system architecture.

The components (either specification or sub-architecture) form the building blocks of RAS architecture, but the main focus in RAS is on the composition of components. A connection represents a channel of interaction between two components. A connection, modeled by timed transitions connecting the components' communication ports, defines the direction of message flow and the delay in the channel. A connection may be uni-directional or bi-directional. For instance, the connection between component A and B describes a *request-reply* relationship and is bi-directional, whereas the connection from C to A is uni-directional. The basic communication model supported by RAS is asynchronous message (token) passing; that is, the sending component does not suspend itself if there is a concurrent activity available.

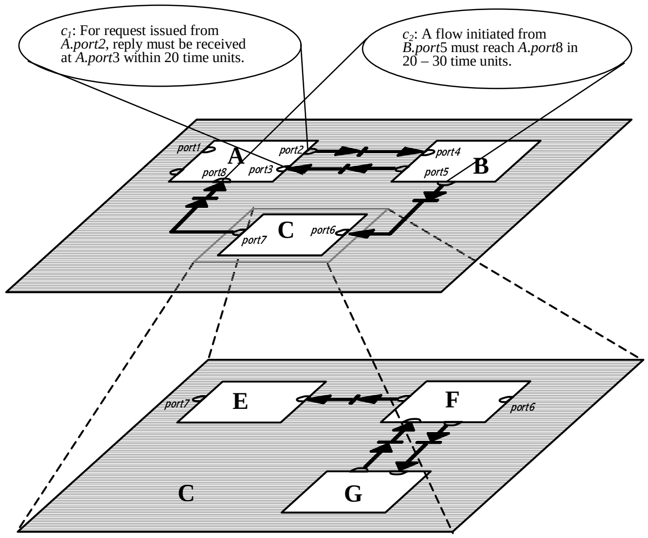


Fig. 1. Framework of the RAS model

In addition to serving as a component's communication interface, RAS communication ports also provide the linkage between the operational design (components and connections) and the descriptive architectural constraints. Each port represents an atomic proposition, which is satisfied by an arriving token (an incoming message at the port). These ports constitute the alphabet (atomic propositions) of the RTCTL formulae that define the constraints. That is, the architectural constraints are specified without using any internal information (or state) of the components, which gives us the flexibility to experiment with different component designs under the same architecture.

The set of constraints can be logically divided into three classes: *component constraints*, *environmental constraints*, and *path constraints*. A component constraint describes the timing properties of a component that its environment expects from it, an environmental constraint describes the timing properties that a component expects from its environment, and a path constraint describes the timing constraint for message transmission among components.

Examples of environmental constraints and path constraints are shown in Figure 1. Formally, environmental constraint C1 is described as: PORT2 $\rightarrow AF^{\leq 20}$ PORT3, and path constraint C2 as: PORT5 $\rightarrow A(G^{\leq 20} \neg$ PORT8 $\wedge$ $F^{\leq 30}$PORT8).

3. A Motivating Example

In this section, we first introduce a practical design example—the hierarchical design of a generic naval Command and Control (C2) system and then use the example to demonstrate our modeling principle and incremental verification techniques.

A C2 system is a typical real-time distributed system representing complex integration of multi-software and multi-hardware components. A naval command and control battle group (BG) is an example of C2 systems. It consists of at least one carrier along with several escorting platforms (ships, submarines, and aircraft). The carriers and their platforms contain a wide variety of sensors and weapon systems, which allow BG to carry out defensive and offensive missions as prescribed by a controlling authority. The threat to BG is multi-warfare in character consisting of enemy submarines, surface ships, and aircrafts. As a consequence, the defense of BG involves anti-submarine warfare (ASW), anti-surface warfare (ASUW), and anti-air warfare (AAW). The enemy platforms must be detected using information from generic BG sensors; they must also be identified, tracked, and engaged before they launch their offensive weapons against the BG assets.

Due to space limitation, we only consider the ASUW subsystems. (The study of the entire system with three types of warfare involved is included in [5]) Figure 3(a) shows ASUW system's structure. It has three components: (1) C2 Center, which reflects the function of compositional warfare organization; (2) ASUW Center, which reflects the function of ASUW organization; and (3) ASUW Fire Unit, which reflects the function of fire unit. The third component, ASUW Fire Unit, can be further refined into two low level components, Fire Control and Damage Assessment, as shown in Figure 3(b). Tables I, II and III show the meanings of ports and transitions in Figure 3(a) and (b).

The first concern of the designer and the user of a C2 system should be the reaction time of the system. Consequently, we have a path constraint

C2.SUR $\rightarrow AF^{\leq 40}$ C2.ASU,

which states that a fire command must be issued within 40 time units after receiving an enemy intelligence message. Since a C2 system is a closed-loop system, a constraint reflecting the time limitation for the feedback of damage assessment results should be included. Thus we have the following environmental constraint:

ASUC.SM $\rightarrow AF^{\leq 22}$ C2.FSU,

Moreover, since the bottleneck for information processing is usually located in component C2 Center, the component is always asked to respond as soon as possible. This is captured by the component constraint:

C2C.R$\rightarrow AF^{\leq 18}$ C2C.S,

We also have an environmental constraint about the inter-arrival time of radar data:

C2.SUR$\rightarrow AG^{\leq 50}\neg$C2.SUR, which states that the time interval between two successive radar messages entering the system is at least 50 units. Figure 4, 5, and 6 show the internal TPN representation of the three components. (The description of nodes of TPN models is given in [6].)

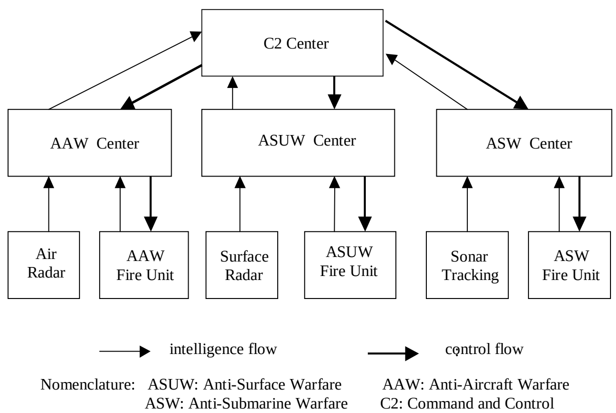


Fig. 2. A generic naval command and control system.

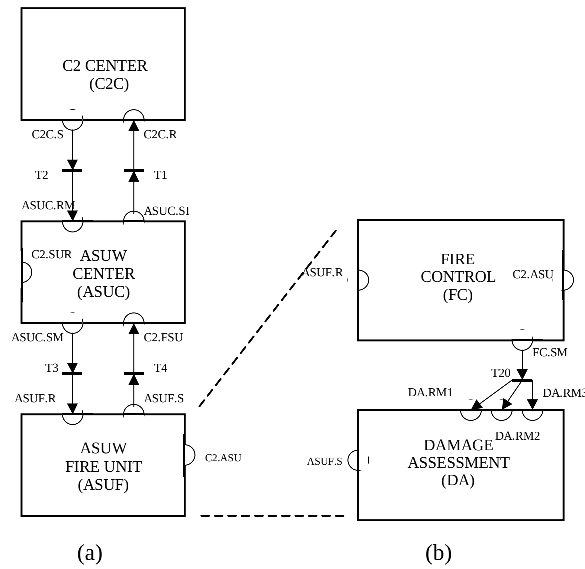


Fig. 3. (a) RAS Architecture of the ASUW system; (b) Refinement of ASUW Fire Unit.

TABEL I LEGEND OF PORTS IN FIGURE 3(a).

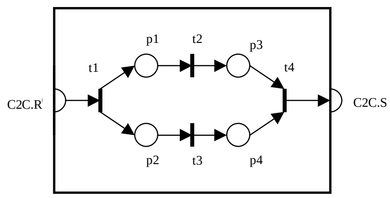
PORT	DESRIPTION	TYPE
ASUC.SI	ASUW center ready to send intelligence to C2 center	output
ASUC.RM	ASUW center received command from C2 center	input
ASUC.SM	ASUW center ready to send command to ASUW fire unit	output
ASUF.R	ASUW fire unit received command from ASUW center	input
ASUF.S	ASUW fire unit ready to send damage assessment results to ASUW center	output
C2C.R	C2 center received messages from centers	input
C2C.S	C2 center ready to send commands to centers	output
C2.SUR	A message from surface radar arrived	input
C2.FSU	ASUW center received damage assessment results	input
C2.ASU	A combat command from ASUW fire unit is sent	output

TABEL II LEGENDS OF TRANSITIONS IN FIGURE 3(a).

TRANSITION	DESCRIPTION	FIRING INTERVAL
T1	ASUW center sends intelligence to C2 center	[1, 1]
T2	C2 center sends command to ASUW center	[1, 1]
T3	ASUW center sends command to ASUW fire unit	[1, 1]
T4	ASUW fire unit sends damage assessment results to ASUW center	[1, 1]

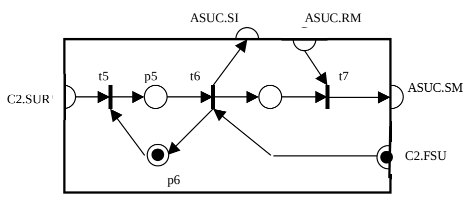
TABEL III LEGENDS OF PORTS AND TRANSITIONS IN FIGURE 3(b).

PORT	DESCRIPTION	TYPE
FC.SM	Ready to send command for collecting firing results	output
DA.RM1,2,3	Waiting for damage assessment	input
TRANSITION	**DESCRIPTION**	**FIRING INTERVAL**
T20	ASUW fire unit sends command of collecting firing results	[1, 2]



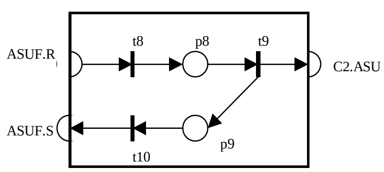
Static firing time intervals:
t1: [1, 2], t2: [3, 5], t3: [3, 5], t4: [5, 6].

Fig.4. The internal representation of C2 Center.



Static firing time intervals:
t5: [2, 3], t6: [1, 2], t7: [4, 6].

Fig. 5. The internal representation for ASUW Center.



Static firing time intervals:
t8: [4, 6], t9: [1, 2], t10: [5, 7]

Fig. 6. The internal representation of ASUW Fire Unit.

Once these component operational models are constructed, we have a complete system architectural model (at this design level) both syntactically and semantically, which is also executable. To enforce the integrity of the design against system requirements, it is important to verify and/or validate that the components and their interactions satisfy their respectively constraints.

4. An Incremental Verification Method for Timing Analysis

In this section, we present an incremental technique for timing verification of RAS design models. Our method works by verifying designs independently at each level – starting from the very top level. Within a given level, we analyze one component at a time. The analysis consists of verifying all constraints related to this component and then reducing it to a simpler TPN of constant size, which, in turn, is used to verify subsequent components.

4.1 Incremental Verification at a Given Design Level

Our algorithm works in two stages. In the first stage, we order the components so that the timing behavior of any component depends only on the components which are lower in this order. This stage assumes that the underlying dependency-graph of components is acyclic. In the next stage, we consider components one at a time in the order computed in the previous stage. Each component is verified against its component constraints and then reduced to a simple TPN of constant size that preserves the ports and all the externally observable time-dependent properties of the component.

The following two definitions are needed for the description of our verification algorithm.

***Definition* 1** Let *PORT_IN* and *PORT_OUT* be the input port set and output port set of a component, respectively. A pair of ports, <*port_in*, *port_out*>, where *port_in* ∈ *PORT_IN* and *port_out* ∈ *PORT_OUT*, is a *calling pair* of the component if a reply token from the environment is expected at *port_in* whenever a token is sent to some other component from *port_out*.

***Definition* 2** Let *PORT_IN* and *PORT_OUT* be the input port set and output port set of a component, respectively. A pair of ports, <*port_in*, *port_out*>, where *port_in* ∈ *PORT_IN* and *port_out* ∈ *PORT_OUT*, is a *passing pair* if a token is expected to be sent to the environment from *port_out* whenever a token from some other component is received at *port_in*.

For example, in Figure 5, <ASUC.SI, ASUC.RM> and <ASUC.SM, C2.FSU> are calling pairs, whereas <C2.SUR, ASUC.SI> is a passing pair.

The basic idea of the algorithm is to *close* passing pairs by connecting them with timed transitions, thereby replacing each component by a very simple net of constant size that preserves all external observable time-critical properties of the component being replaced. In Figures 7-11, we introduce five *component-level reduction rules* to support such closure (The formal description of these rules is included in [16]). The rules work at a much coarser level than the reduction rules given in [15], which work at individual transition level. Consequently, we need fewer applications of our rules to reduce the size of the model being analyzed. In these rules, the *static firing time interval* $SI(t_{(1,2)})$ of transition $t_{(1,2)}$ is the time delay interval of the message transfer between the epoch of *port1* being marked and that of *port2* being marked, and $SI(t_{((1,2),3)})$ of $t_{((1,2),3)}$ is the time delay interval of the message transfer between the epoch of both *port1* and *port2* being marked and that of *port3* being marked, and so on. Without the loss of generality, in our reduction rules, we assume that for any component model to be reduced, no transitions in the model are enabled in the initial state of the system. Particularly, we assume:

***Assumption* 1** The TPN model of the system under design is safe.

This assumption implies that the system takes the same statistical property of time to deal with inputs arriving in different time. This enables us to consider only one set of inputs to verify the system constraints. In this case, for any component to be reduced, no transitions in the component will be enabled by the coming tokens at its input ports until the previous input tokens have reached its output ports, which makes our reduction rules applicable. The assumption is looser than the one made in [15], which asks the underlying Petri net of a TPN is safe. However, the assumption is still a little strict: the reduction rules are applicable to verify the constraints of such systems that no queue exists. Nevertheless, these rules help to show one of the natures of our model, that is, it supports incremental verification.

Of course, we need to make sure that these reduction rules preserve timing properties of nets. Let $L(N)$ be the set of all firing schedules of TPN N. For any set of transitions U, let $L(N)|U$ be the set of all schedules of N where transitions not from U have been deleted from each of schedules.

***Definition* 3** Let N_R be the result of applying some reduction rule to a time Petri net N, and let U be any set of transitions of N which are left completely unmodified by this transformation. Then we say that a reduction rule is *equivalence preserving* if $L(N)|U=L(N_R)|U$.

Theorem *Reduction rules 1 - 5 are equivalence preserving.* (A proof is included in [5].)

Because of dependencies among components we need to be careful in choosing the order in which components are considered. E.g., if a component has a calling pair, it needs to interact with other components. We say that component C_i *depends on* component C_j, denoted by $C_i \rightarrow C_j$, if after C_i sends a token from the output port of its calling pair, it can not receive a token at the input port if we remove C_j away from the system. If $C_i \rightarrow C_j$ then C_j has to be analyzed before C_i. If $C_i \rightarrow C_j$ and $C_j \rightarrow C_l$, then C_l has to be analyzed before C_j, which, in turn, has to be analyzed before C_i. We can build a *dependency graph* such that there is a directed edge from component C_i to component C_j iff C_i depends on C_j. If this dependency graph contains a cycle, then we can't pick an order, so we assume that

***Assumption* 2** The dependency graph does not contain a cycle.

In this case we can (linearly) order the component (e.g., by performing a topological sort [4]) in such a way that all dependency edges go from a component higher in this order to a component lower in the order. In other words, the first component does not depend on any other components so it can be verified and reduced to a simple

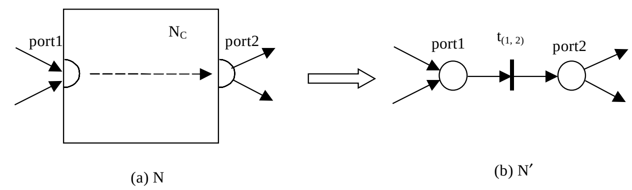


Fig. 7. Illustration of component-level reduction rule 1.

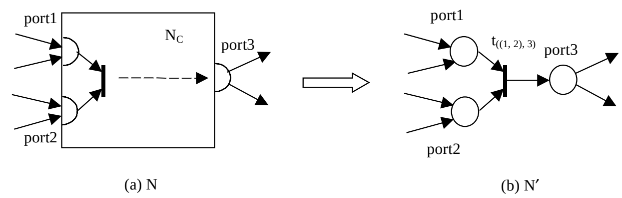


Fig. 8. Illustration of component-level reduction rule 2.

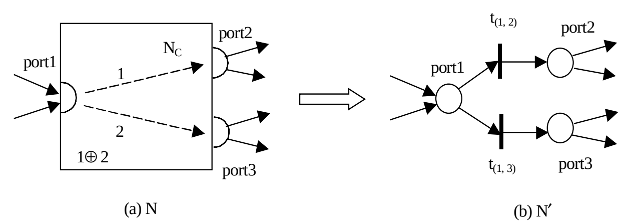


Fig. 9. Illustration of component-level reduction rule 3. The token reached *port*1 will either follows token flow path 1 to reach *port*2, or follow token flow path 2 to reach *port*3.

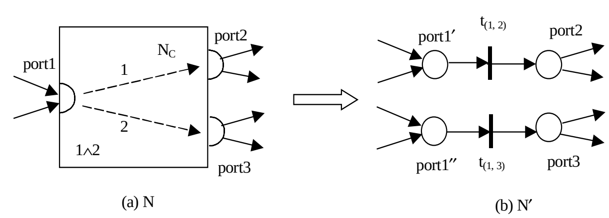


Fig. 10. Illustration of component-level reduction rule 4. The arrival of a token at *port*1 will result in a token reaches each of *port*2 and *port*3.

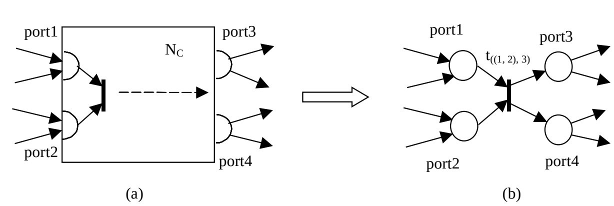


Fig. 11. Illustration of component reduction rule 5.

TPN. The second component depends only on the first one, which has already been reduced, and so on. Suppose that there are *k* components, which are linked together by inter-related connection structures to form the system or subsystem. Now we can describe our incremental verification method as follows:

Verification Algorithm

1. *For each component, list the reduction rule that can be applied to it*.
2. *Build the dependency graph on components and perform a topological sort. If the sorted order is C_1, C_2, …, C_k, then we know that $\forall i > j$, C_j doesn't depend on C_i. Set i = 1.*
3. *For C_i,*
   3.1 *Verify environmental constraints defined on it. If violation is found, quit the algorithm, report the violation of constraints and redesign the system or subsystem.Verify path constraints defined on it. If violation is found, quit the algorithm, report the violation of constraints.*
   3.2 *Calculate time delay interval of the message transfer between each passing pair of the component;*
   3.3 *Verify component constraints. If violation is found, quit the algorithm, report the violation of constraints.*
4. *If all constrains are verified, then the algorithm ends successfully; else set i = i+1, and go back to 3.*

We now illustrate the incremental verification on the C2 system given in Section 3. We can find that the four constraints ensure that the C2 system satisfies Assumption 1. Therefore, our verification method is applicable to this system. (Due to the space limitation, we skip some steps that are not crucial for understanding the algorithm.) The C2 system consists of three components: C2 Center, ASUW Center and ASUW Fire Unit. There are two dependency edges: ASUW Center $\rightarrow$ C2 Center, and ASUW Center $\rightarrow$ ASUW Fire Unit, so we can order the three components as

C2 Center, ASUW Fire Unit, and ASUW Center.

Applying reduction rule 1 to component C2 Center results in Figure 12(a), and applying reduction rule 4 to component ASUW Fire Unit results in Figure 12(b). Applying simple reachability analysis [2], we conclude

*D*_(C2C.R, C2C.S) = [9, 13],

*D*_(ASUF.R, C2.ASU) = [5, 8], and

*D*_(ASUF.R, ASUF.S) = [10, 15],

where *D*_(A, B) represents the message transferring time delay between A and B. This gives

*D*_(ASUC.SM, C2.FSU) = [12, 17].

So the command constraint C2C.R $\rightarrow AF^{\le 18}$C2C.S and the environmental constraint ASUC.SM $\rightarrow AF^{\le 22}$ C2.FSU are verified. Next, we consider component ASUW Center. Figure 12(c) shows its interaction with the other two components, which have been simplified. Based on Figure 12(c), we obtain

*D*_(C2.SUR, C2.ASU) = [24, 35].

Based on this value, we have verified the path constraint C2.SUR $\rightarrow AF^{\le 40}$ C2.ASU. Therefore, all three constraints have been verified.

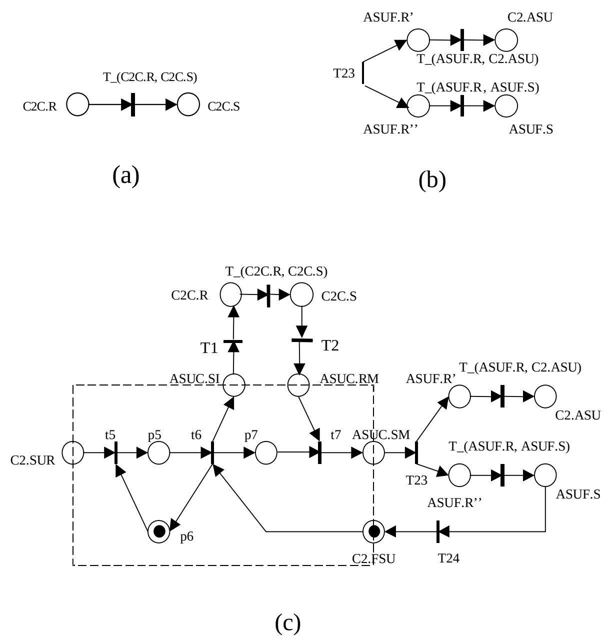


Fig. 12. (a) Reduction of component C2 Center.
(b) Reduction of component ASUW Fire Unit.
(c) The simple but equivalent model of the system.

4.2. Constraints Decomposition and Incremental Verification across Design Levels

In the last subsection we showed that we could use the compositional algorithm to verify the design at any given level. In this section, we briefly describe an incremental technique that ensures that, after each refinement step, we can safely plug the sub-architecture into its parent-level architecture without having to re-verify the entire model. For more details, refer to [DWS97]. This technique consists of three basic elements: 1) We automatically derive constraints for the lower-level design from the higher-level constraints. 2) The lower-level design is verified using the technique discussed in the previous section. 3) The sub-architecture is plugged into the parent-level architecture to form a more detailed architectural model.

When we refine an RAS component into a sub-architecture, the RAS model mandates that it inherit all the ports from the high-level component to ensure interface consistency. During the verification of the high-level design, we compute time delays for certain port-pairs. Correctness of the high-level design is contingent on the values of these time delays. So for the lower level design, we add these delays as constraints that must be satisfied. As long as these constraints are satisfied, we can ensure that the low-level designs is consistent with the high-level design.

Let's take the analysis of the C2 system as an example. In the last subsection, we finished the verification of its high-level design. Figure 13 shows the internal representation of a low-level design for ASUW Fire Unit. From this figure we know ports ASUF.R, C2.ASU, and ASUF.S are inherited from Figure 6, the high-level design model for ASUW Fire Unit (The legend of nodes in Figure 13 is included in [5]). During the high-level verification, we concluded that *D*_(ASUF.R, C2.ASU)=[5, 8], and *D*_(ASUF.R, ASUF.S) = [10, 15], which enabled us to prove the constraints for the high-level design. Hence, for the low-level model we have the following two constraints:

(1) ASUF.R $\rightarrow A(G^{\leq 5}\, \neg$C2.ASU $\wedge\, F^{\leq 8}$ C2.ASU), and

(2) ASUF.R $\rightarrow A(G^{\leq 10} \neg$ASUF.S $\wedge\, F^{\leq 15}$ ASUF.S).

Then we can use the compositional verification algorithm developed in the previous subsection to verify the design in Figure 13 as long as the static firing time interval for each transition is given.

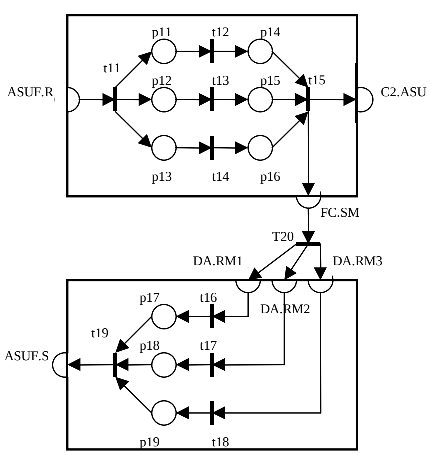


Fig. 13. Refinement of Figure 6.

5. Conclusion

We have presented an RAS-based incremental approach for architectural modeling and verification of real-time distributed systems. The contribution of this paper is twofold: First, it provides a systematic way to link real-time system constraints to the process of formal modeling and analysis to ensure that the constraints are met at any given design level. Second, our approach is scalable in both modeling and analysis. We have presented an incremental verification technique for timing analysis based on constraint propagation and component-level Petri net reduction. This technique significantly reduces the complexity of analysis by supporting both horizontal and vertical composition-ability. These two features together ensure that a complex system model can be built, changed and verified progressively both at a given design level and across different design levels.

Reference:

[1] E. Battiston, A. Chizzoni, and F.D. Cindio, Inheritance and concurrency in CLOWN, *Proc. of the 1st Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, Italy, 1995.

[2] B. Berthomieu, and M. Diaz, Modeling abd verification of time dependent systems using time Petri nets, *IEEE Transactions on Software Engineering*, vol.17, no.3, 1991, 259-273.

[3] D. Graigen, S. Gerhart, and T. Ralston, Formal methods reality check: industrial usage, *IEEE Transactions on Software Engineering*, vol.21, no.2, 1995, 90-98.

[4] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. The MIT Press, 1994.

[5] Y. Deng, J. Wang, R. Sinha. Compositional and incremental verification of real-time systems based on the RAS model. *Technical Report, School of Computer Science, Florida International University*, 1997.

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

[7] X. He and Y. Ding, Object orientation in hierarchical predicate transition nets, *Advances in Petri Nets, Lecture Notes in Computer Science* (ed. G. Ahga), in press.

[8] X. He, Specifying and verifying real-time systems using time Petri nets and real-time temporal logic, *Proc. of the 6th Annual Conference on Computer Assurance* (*COMPASS*'91), Maryland, U.S.A., 1991, 135-140.

[9] X. He, A formal definition of hierarchical predicate transition nets, *Proc. of the 17th International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Science*, vol.1091, Osaka, Japan, 1996, 212-229.

[10] K. Jensen, *Colored Petri Nets*, vol.1, Springer-Verlag, 1992.

[11] C. Lakos, From colored Petri nets to object Petri nets, *Proc. of the 16th International Conference on Application and Theory of Petri Nets*, Torino, Italy, 1995.

[12] D. Mandrioli, A. Morzenti, M. Pezze, P. Pietro, and S. Silva, A Petri net and logic approach to the specification and verification of real time systems. *Formal Methods for Real time Computing*, *1996.*

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

[14] T. Murata, Petri nets: properties, analysis and applications, *Proceedings of IEEE*, vol. 77, no.4, 1989.

[15] R. Sloan, U. Buy. Reduction rules for time Petri nets. *Acta Informatica*. 33:687-706, 1996.

[16] J. Wang and Y. Deng, Incremental modeling and verification of flexible manufacturing systems, *Journal of Intelligent Manufacturing*, 1998, to appear.

[17] J.M. Wing, A specifier's introduction to formal methods, *IEEE Computer*, vol.23, no.9, 1990, 8-24.

Appendix. Formulization of RAS Model

A RAS consists of a set of compositions C (a composition may correspond to a design level, or the concept of sub-architecture given in Section 2.1) and a hierarchical mapping h:

RAS = (C, h).

(1) $C = \{C_1, C_2, \ldots, C_k\}$, and $C_i = \{ Cm, Cn, Cs\}$ for each C_i, where

- Cm is a set of components. Each $Cm_j \in C_m$ is defined by a TPN. Let

$Cm_j.PORT\_IN = \{p \mid p \in Cm_j.P,\ {}^*p \cap Cm_j.T = \varnothing\}$;

$Cm_j.PORT\_OUT = \{p \mid p \in Cm_j.P,\ p^* \cap Cm_j.T = \varnothing\}$;

$Cm_j.PORT = Cm_j.PORT\_IN \cup Cm_j.PORT\_OUT$.

$Cm_j.PORT\_IN$ is called the set of input ports of component Cm_j, $Cm_j.PORT\_IN$ the set of output ports of Cm_j, and $Cm_j.PORT$ the set of ports of Cm_j. Moreover, For $\forall\ Cm_j, Cm_k \in C_i.Cm$,

$Cm_j.P \cap Cm_k.P = \varnothing$,

$Cm_j.T \cap Cm_k.T = \varnothing$.

- Cn is connection. $Cn = (P, T, I, O, M_0, SI)$ such that

$$Cn.P \cap \left(\bigcup_{Cm_j \in Cm} Cm_j.P \setminus Cm_j.PORT \right) = \varnothing$$

$$Cn.T \cap \left(\bigcup_{Cm_j \in Cm} Cm_j.T \right) = \varnothing .$$

Also, in the TPN of composition C_i,

$$C_i.P = \left(\bigcup_{Cm_j \in Cm} Cm_j.P \right) \cup Cn.P ,$$

$$C_i.T = \left(\bigcup_{Cm_j \in Cm} Cm_j.T \right) \cup Cn.T .$$

Let

$$C_i.PORT = \bigcup_{Cm_j \in Cm} Cm_j.PORT ,$$

$C_i.PORT\_INT = \{p \mid p \in C_i.PORT,\ {}^*p \cap C_i.T \neq \varnothing \wedge p^* \cap C_i.T \neq \varnothing\}$;

$C_i.PORT\_EXT = \{p \mid p \in Cm_j.PORT,\ {}^*p \cap C_i.T = \varnothing \vee p^* \cap C_i.T = \varnothing\}$.

$C_i.PORT\_INT$ is called the set of internal ports of composition C_i, $C_i.PORT\_EXT$ the set of external ports of C_i, and $C_i.PORT$ the set of ports of C_i.

- Cs is a set of constraints. Each $Cs_j \in Cs$ is an RTCTL formula and it only uses ports as its atomic propositions. The atomic proposition is true at the moment τ iff:

- marking transition happens at τ, and

- the port contains a token in the new marking.

In the temporal structure $\Sigma = (S, R, L)$, $S = (M, \varphi_M)$ where M is a TPN marking, and φ_M is the global time when the TPN enters M; R is a binary relation on S, which is indicated by firing transitions; and L is a mapping: $(M, \varphi_M) \rightarrow C_i.PORT$. In addition, the following condition is enforced:

$$\bigwedge_{Cs_i \in Cs} Cs_i \qquad (1)$$

(2) For $\forall\ Cs_j \in C_i.Cs$, denote by $Cs_j.PORT$ the set of ports which are used as atomic propositions of Cs_j. If

$Cs_k.PORT \cap Cm_j.PORT \neq \varnothing$,

we say Cs_k is defined on Cm_j; if

$Cs_k.PORT \subseteq Cm_j.PORT$,

we say Cs_k is defined only on Cm_j. Let

$Cm_j.Cs = \{Cs_k \mid Cs_k \in C_i.Cs \wedge Cs_k.PORT \subseteq Cm_j.PORT \}$;

$\forall\ C_i \in C$, $\forall\ Cm_l \in C_i.Cm$, h: $Cm_l \rightarrow C_j$, $j \neq i$, such that

$$Cm_l.PORT = C_j.PORT\_EXT, \qquad (2)$$

$$Cm_l.Cs \subseteq C_j.Cs. \qquad (3)$$

In the above definition, Exp. (1) gives the *horizontal constraint consistency condition*; Eqs. (2) and (3) together give the *vertical interface consistency conditions*.