International Journal of Software Engineering
and Knowledge Engineering
Vol. 14, No. 5 (2004) 471–496
© World Scientific Publishing Company

CONSTRAINT PROPAGATION AND PROGRESSIVE VERIFICATION FOR COMPONENT-BASED PROCESS MODEL

YI DENG*,†, JIACUN WANG§, XUDONG HE*,‡ and JEFFREY J. P. TSAI¶

*School of Computer Science,
Florida International University,
Miami, FL 33199, USA
†deng@cs.fiu.edu
‡hex@cs.fiu.edu
§Department of Software Engineering, Monmouth University,
West Long Branch, NJ 07764, USA
jwang@monmouth.edu
¶Department of Computer Science,
University of Illinois at Chicago,
Chicago, IL 60607-7053, USA
tsai@cs.uic.edu

Received 19 February 2003
Revised 1 May 2004
Accepted 11 June 2004

System assembly is one of the major issues in engineering complex component-based systems. This is especially true when heterogeneous, COTS and GOTS distributed systems, typical in industrial applications, are involved. The goal of system assembly is not only to make constituent components work together, but also to ensure that the components as a whole behave consistently and guarantee certain end-to-end properties. Despite recent advances, there is a lack of understanding about software composability, as well as theory and techniques for checking and verifying component-based systems. A theory of software system constraints about components, their environment and about system as a whole is the necessary foundation toward solid understanding of the composability of component-based systems. In this paper, we present a systematic approach for constraint specification and constraint propagation in concert with design refinement with a novel technique to ensure consistency between system-wide and component constraints in a design composition process of component-based systems. The consistent constraint propagation is used in our approach to drive progressive verification of the design. It allows us to verify overall design composition without interference of internal details of component designs. Verification is done separately at architectural and component levels without having to compose results of component analyses. A component can be safely replaced with alternative design without re-verifying the overall system composition so long as the replacement conforms to the corresponding interface and component constraint(s).

Keywords: System design; system composition; constraints verification; Petri nets; temporal logic.

1. Introduction

The concept of constraints has been widely used and is considered an important means for developing high quality software systems. Definitions, purposes and applications of constraints vary. However, what is common is that they represent certain conditions or properties that must be satisfied by system design and implementation. In other words, a constraint is a required property or assertion about a system, the violation of which will render the system unacceptable to one or more stakeholders. For example, system constraints have been defined in the forms of assertions, contracts, pre/post-conditions, invariants, etc. [4, 12, 16, 21, 28] to support OO analysis and design. In [25], an elaborate set of security constraints was presented for multilevel secure database management systems (MLS/DBMS), and a system architecture was proposed to handle security constraint processing in distributed MLS/DBMS environment. In [9], we introduced the concept of security constraint patterns, which formally specify the generic form of security policies that all implementations of the system architecture must enforce, and showed how to use these constraints to guide the design and analysis of security systems. System constraints can be described or specified in different forms ranging from natural languages, to IDL [18], UML [23], OCL [28], to formal languages and notations, e.g. temporal logic [10, 11] and Petri nets [20, 22]. To enable formal verification, rigorously defined mathematical formalism is required. However, given the large size of state space of software systems, even for relatively small systems, brutal force constraints verification is impractical without the support of an effective means to manage the complexity [8].

In this paper, we define a constraint as a system-wide property that must be satisfied by a design. Given a composition of the system by its components, the constraint defines the property that the components as a whole must hold under the particular composition. Consequently, the constraints serve as the basis to enforce traceability and consistency in design decomposition and refinement, and the basis for verifying whether a given design conforms to the system-wide requirement that it supposes to enforce. We are interested in how to formally assure the system-wide constraints in design decomposition and refinement. There are numerous techniques for formal verification in the literature. The key is how to apply these techniques systematically in concert with the design refinement process, and how to manage the verification process in such a way as to control the state explosion problem associated with formal analysis, which are the focus of this paper. We present a practical approach of verification driven by a systematic and consistent propagation of system-wide constraints of a software system onto its components in the process of design refinement. It should be noted that we do not consider the verification of individual components (i.e. verifying an implementation of a component against its design specification) in this paper since there have been numerous research efforts and methods dealing with this research issue; however, our proposed design level constraints propagation and progressive verification method will greatly facilitate

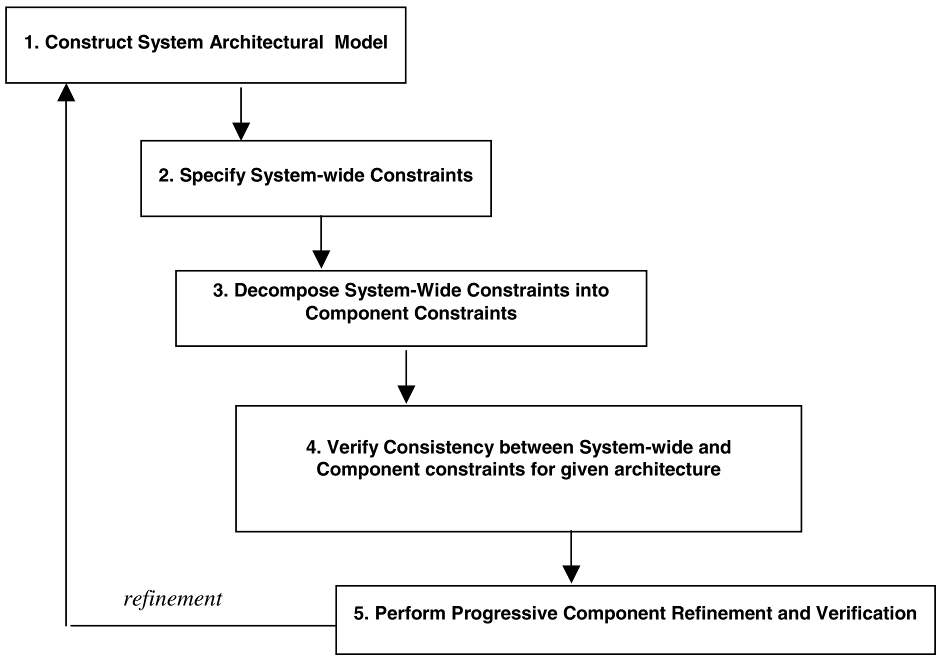

1. Construct System Architectural Model
2. Specify System-wide Constraints
3. Decompose System-Wide Constraints into Component Constraints
4. Verify Consistency between System-wide and Component constraints for given architecture
5. Perform Progressive Component Refinement and Verification
refinement

Fig. 1. Framework of constraint-driven progressive verification.

the above individual component verification task by reducing component sizes into manageable levels and thus provides an effective way to control the complexity of verifying individual components.

In a nutshell, our approach for constraint propagation and verification follows the following broad steps as shown in Fig. 1. First, we model a software system as a composition of components without considering internal details of the components. This model is called the system architectural model. Second, system-wide constraints are formulated and formally specified, which define one or more system-wide properties which the given composition of the system must satisfy. Third, the global constraints are decomposed into component constraints that each component must satisfy under the given system composition or architecture. How to decompose the system-wide constraints into component constraints is subject to problem-specific design decisions. It is an issue that should be addressed by a design method rather than a verification method. However, our approach provides a systematic way to ensure the consistency between system-wide and component constraints. By combining a component constraint with the interface (denoted by ports) of the component (defined in the architectural model), we can easily generate a simple component model, called component requirement model, which preserves the properties defined by the component constraints. These generated component models are small and constant sized. In Step 4, these small component models are then plugged into the overall system architecture. The resultant architectural model is verified against the system-wide constraint patterns using standard analysis techniques, e.g. reachability analysis [20] and model checking [7]. This verification shows the consistency between global and component constraint patterns under the given system architecture (Step 1). Once the consistency between system-wide and component constraints is verified, these component constraints serve as the basis for component design in Step 5. A more detailed architectural or behavior model for each component can be constructed, if necessary, and verified against the corresponding constraint pattern using the same process described above. Again, any available analysis technique can be used to verify the component model against its constraint model.

Under this constraint-driven approach of progressive verification, a component, which could be a composition of other components, in the design composition can be safely replaced (in terms of verified property) by an alternative design without re-verifying of the overall system design, so long as the replacement has the same interface and satisfies the corresponding component constraints. It allows us to analyze overall design composition by verifying individual components in the system composition. Verification is done separately at architectural and component levels. This significantly reduces the complexity. There is no need to compose the results of analysis (once the consistency between system-wide and component constraint patterns is established).

In this paper, two complementary formal notations, temporal logic [7] and Petri nets [20, 26], are employed in concert with the described methodology. The former, a popular descriptive formalism, is best suited for describing rules and constraints. By contrast, the latter is a well-known operational model well suited for modeling the control and composition of distributed systems. Temporal logic, more specifically Computational Tree Logic (CTL) [10, 11], is used to describe system constraint models. Petri nets, more specifically Place Transition Nets (P/T nets) [20], are used to describe design composition models. These two notations are seamlessly integrated [9, 27] in our methodology. It should be pointed out, however, the approach of the constraint-driven verification is general and independent of these two specific formalisms.

The rest of the paper is organized as follows: In Sec. 2 we discuss in more detail the methodology of constraint propagation and progressive verification. In Sec. 3, we present an automatic method to convert a component constraint to its corresponding component requirement model, which helps to significantly reduce the complexity of the verification of consistency between system-wide constraints and intermediate component constraints. In Sec. 4, we illustrate the constraint propagation and progressive verification method through a communication protocol example. Section 5 introduces our software tool for system modeling and verification. Section 6 gives a concluding remark.

2. A Methodology of Constraint-Driven Progressive Verification

In this section, we discuss in more detail the methodology of constraint-driven progressive verification outlined in Fig. 1. The discussion consists of two parts. First, we present the overall process of progressive verification driven by consistent constraint propagation. Second, we discuss techniques for verifying the consistency between system-wide constraints and component constraints.

2.1. *Process of constraint-propagation and progressive verification*

As shown in Fig. 1, the constraints propagation and verification process consists of the following broad steps.

Step 1. Construct a top-level structural model of the system

The purpose of this step is to build the model for the top-level system architecture, which describes the overall organization of the system, as well as the coordination between its components. The internal structure and behavior for the components are not included in this model. This model is constructed by decomposing the system into subsystems or components and connections between the components. In this paper, we use P/T nets [20] to describe a system's architectural model and component models. A brief introduction of the Petri net model is given in Appendix 1.

Figure 2 shows an example of the top-level structural model of a system with three components. As we can see, the interface that the component communicates with the rest of the system is specified as component communication ports (denoted graphically by half circles), including input ports (e.g., port8) and output ports (e.g., port9), represented by Petri net places. Simple Petri net pieces are used to define the connections among components, which represent channels of interaction between components. Notice that at this level no internal information about a component is revealed, which is to be established in a later design stage.

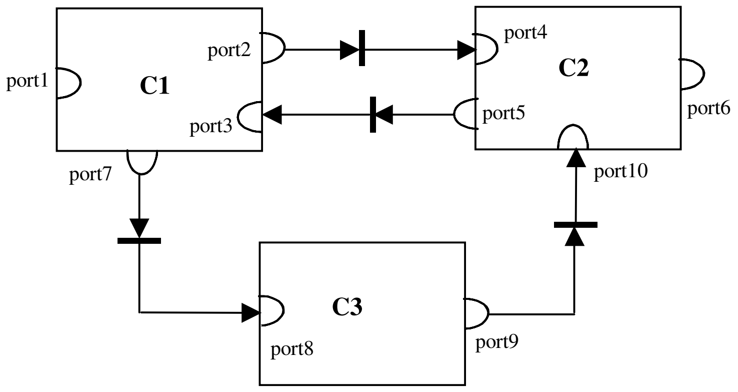


Fig. 2. Illustration of system structural model.

Step 2. Specify system-wide constraints

At this stage, the system-wide constraints imposed on the top-level structural model created in the previous step are formulated by formalizing required properties of the system in terms of constraints imposed on system components and connections between them. These constraints are specified using only the interface (ports) of the components. By formalizing the system-wide requirements in terms of architectural constraint, it not only removes ambiguity in the description, but also makes it easier to detect possible inconsistency or conflict between different (competing) requirements.

In this paper, computational tree logic (CTL) [10, 11] formulas are used to formally define the constraints. A brief introduction of CTL is given in Appendix 2. In the constraint specification, the CTL formulas only use ports (places) as atomic propositions. The atomic proposition is true if and only if the port contains a token. In the temporal structure $\Sigma = (S, R, L), S = \{M\}$ where $\{M\}$ is the set of reachable markings of a Petri net; R is a binary relation on S, which is indicated through firing transitions; and L is a mapping: $M \rightarrow PORT$, where $PORT$ is the set of ports.

For example, we may have the following constraint for the structural model shown in Fig. 1: AG (port1 $\rightarrow AF$ port2), which means that whenever port1 gets a token, port2 will eventually get a token.

Step 3. Decompose system-wide constraints to components

In this step, we decompose the system-wide constraints into a set of intermediate constraints that can be imposed on the components to guide component design. The constraints defined on a given component specify the functionalities of that component in terms of its contribution toward the satisfaction of the system-wide constraints under the given architecture. Because the original constraints allow different combinations of the intermediate constraints, the task of propagating the system-wide constraints onto the components requires one to carefully examine and explore the boundary among the components. This is because such propagation effectively partitions the system-wide function to individual component functions and determines the interface and protocols of interaction among them. For this reason, we consider a specific choice of constraint decomposition as a design issue, which should be made based on tradeoffs between different design considerations.

Step 4. Verify the consistency between system-wide constraints and component constraints

Because there is no easy way to automate the constraint decomposition, we need to verify if the intermediate constraints are consistent with the system-wide concurrent constraint, i.e., the component constraints collectively satisfy the system-wide constraints under the given system composition represented by the structural model. Only after these intermediate constraints are proven to be consistent with the system-wide constraints can it be meaningful to design the components against these intermediate constraints. This verification is facilitated and kept manageable by two facts: (a) the component constraints are of forms similar to the system-wide constraints because the former are generated from the latter; and (b) the component constraints are connected together by the known structure of the architecture model. The problem of verifying consistency between system-wide and component constraints can be stated as the component constraints, collectively under the connection structure given by the architectural model, satisfy the corresponding system-wide constraints.

Although showing that two arbitrary sets of temporal formulas being consistent is a difficult problem [2], two observations help to make the problem manageable in our context. First, the component constraints are "derived" from the system-wide constraints. Therefore, they share similar forms and structures. Second, we have the connectors of the component constraints available. Armed with these two pieces of information, we introduce a technique to check the consistency between the two sets of temporal constraints, which consists of the following steps:

(1) Assume C to be the set of components connected together under a given system architecture. From each constraint of component $c \in C$, we derive a small and constant-sized PN, which we call *component requirement model* of c denoted as CRM(c). CRM(c) can be constructed by translating the temporal formula representing the component constraint into its PN form. (See Sec. 3.) Notice that CRM(c) has the same ports as c because the formula is defined on the ports, i.e. these ports constitute the vocabulary of the formula.

(2) We plug the set of newly created PNs $\{\text{CRM}(c) \mid c \in C\}$ into the system architecture model (also represented as a PN), which results in a complete net model. Call this net the *constraint model of the architecture*. This PN represents the model of the component constraints based on the given system composition. This implies that if this PN satisfies the system-wide constraints, then the component constraints collectively are consistent to the system-wide constraints based on the given system architecture.

(3) Verify if the constraint model satisfies the system-wide constraints. A number of available techniques, e.g. reachability analysis, can be used for this verification.

Step 5. Perform incremental design and verification of the components

The completion of Step 4 has two important implications: (a) the component constraints can be "trusted" as the basis for component design; and (b) if every component design conforms to its component constraints, the resulting system with the inclusion of the component designs will automatically satisfy the system-wide constraints. This is an important conclusion because it significantly reduces the complexity of analysis. The component can be further decomposed, in which case, we iterate the above steps resulting in an incremental architectural composition and analysis process. If necessary, different decompositions that conform to the interface and constraints of a component can be developed and plugged into the system model to evaluate different design alternatives.

2.2. *Techniques for constraint verification*

2.2.1. *Reachability analysis*

Petri nets analysis methods may be classified into the following four groups: (1) the reachability tree method, (2) the linear algebraic approach, (3) reduction or decomposition techniques, and (4) simulation. The first method involves essentially the enumeration of all reachable markings. It should be able to apply to all classes of nets, but is limited to "small" nets due to the complexity of the state-space explosion. On the other hand, linear algebraic and reduction techniques are powerful but in many cases they are applicable only to a special subclass of Petri nets or special situations. For complex Petri net models, discrete-event simulation is another way to check the system properties [20, 26, 29].

2.2.2. *Model checking*

Model checking is a technique for verifying finite state systems [7, 15]. The method has been used successfully in practice to verify complex sequential circuit designs and communication protocols.

Given a Kripke structure $\mathrm{M} = (\mathrm{S}, \mathrm{R}, \mathrm{L})$ that represents a finite state concurrent system and a temporal logic formula f expressing some desired specification, model checking technique either give a counter example or find the set of all states in S that satisfy f: $\{s \in \mathrm{S} \mid \mathrm{M}, s \models f\}$ where:

- S is a finite set of states
- $\mathrm{R} \subseteq \mathrm{S} \times \mathrm{S}$ is the transition relation, with $(s, t) \in \mathrm{R}$ meaning that t is an immediate successor of s.
- $\mathrm{L} : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$, is the valuation of atomic propositions in each state, where AP is a finite set of atomic propositions.

In the first algorithm for solving the model checking problem, the nodes represent the states in S, the arcs in the graph give the transition relation R and the labels associated with the nodes describe the function L.

Model checking consisted of several tasks:

Modeling The first task is to convert a design into a formalism accepted by model checking tool, such as using Petri nets or SMV input language to define the system model.

Specification Specification is to state the properties that the design must satisfy. The specification is usually given in some logical formalism. It is common to use Temporal Logic, such as CTL, CTL*, RTCTL for different systems.

Checking The verification is completely automatic. However, in practice, it often involves human assistance. One such manual activity is the analysis of the results. In case of a negative result, the user is often provided with an error trace.

3. Constructing Component Requirement Models (CRM)

As discussed in Sec. 2, only after intermediate (component) constraints are proven to be consistent with the system-wide constraints can it be meaningful to design the components against these intermediate constraints. In this section, we discuss the construction of component requirement models (CRM) as the means to verify consistency between system-wide and component constraints. This is achieved by replacing the internal part of a component with one very simple PN structure in such a way that the resulting CRM satisfies the corresponding component constraint, so that we can use the CRM to represent the component constraint in the architectural model. The CRM is often composed of few transitions, to connect input ports and output ports directly, while maintain its external semantics in terms of the property specified by the component constraints. Again, refer to the Appendices for the underlying formal notations used in this paper.

3.1. *Map component constraint to simple PN*

Now we consider mapping a component constraint, which specifies the relationship between the input and output ports of the component, to a simple PN. We are interested in a class of system constraints that define the causal relationships between the inputs and outputs. Therefore, we use only a subset of CTL formulas that have the following general form:

$$AG(I \rightarrow O)$$

where I is a propositional logic formula whose atomic propositions are ports, representing the inputs of the component, and O is a T-formula (see below) representing the outputs of the component. To ease our descriptions, we first define some terms. A *G-node* is a CTL element in the form of $AG\,\neg p$ and an *F-node* is a CTL element in the form of $AF\ p$, where p is a restricted propositional logic formula whose atomic propositions are those ports and their negations. A *T-Formula* is defined as follows:

- Each $AG\,\neg p$ is a T-formula;
- Each $AF\ p$ is a T-formula;
- If p and q are T-formulas, so are $p \wedge q$ and $p \oplus q$.[a]

Both G-node and F-node are called *atomic node* or a T-node in T-formula.

Notice that in a T-formula, existential quantifier E is not used. That is because it only means that there is such a possibility for the formula following it being true. This limited information is not enough to determine properties of a component and therefore meaningless in modeling a system in our context.

The process of mapping a CTL formula to a simple PN proceeds in four steps:

(a) Construct the PN representation of I in the component constraint, called I-net;

[a]Symbol $\oplus$ stands for “exclusive or”.

(b) Construct the PN representation of O in the component constraint, called O-net;
(c) Combine I-net and O-net together to form the PN representation of the constraint; and
(d) Remove redundant nodes in the resulting PN.

The PN corresponding to a given formula can be constructed recursively. For any proposition P_i which appeared in a formula, we use a PN to represent it, and its logic value is indicated by a particular place p_i in the net. If p_i is marked, P_i is true; If p_i is unmarked, P_i is false. We call p_i the *characterization place* of proposition P_i. Moreover, if P_i appears in formula I, p_i is the source place of its PN representation, i.e., the preset of p_i is empty; if P_i appears in formula O, p_i is the absorbing place of its PN representation, i.e., the postset of p_i is empty.

Two kinds of transitions are used in the PN representation of a constraint. One is temporal transition, which represents the change from the state indicated by the pre-condition of a constraint to the states indicated by the post-condition, and whose firing represents a real action. The other is logic transition, which is only used to describe the relationship between primitive places (ports) and characterization places, and whose firing does not cause any real state change. Pictorially, a temporal transition is described by a thick bar, while a logic transition is described by a thin bar.

Both characterization places and logic transitions are *temporary nodes*, for they are introduced in the process of constructing a CTL formula's PN representation. When the construction is finished, we will apply some rules to remove these temporary nodes. So in the final PN representation of a CTL formula, there are only ports and temporal transitions.

3.1.1. *Construct I-net of a component constraint*

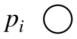
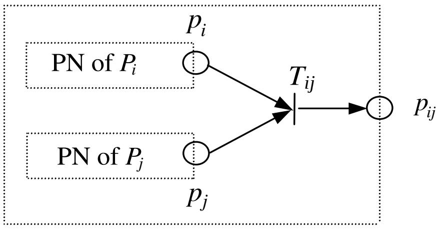
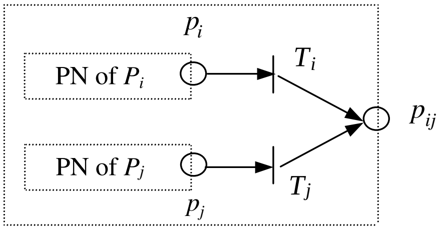
In Table 1, we list the PN construction for typical terms in an I-net. For example, for a compound proposition $P_i \wedge P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively, we introduce a new transition T_{ij} and a new place p_{ij} and then build the PN as shown in the table to represent this formula. The characterization place of the formula is p_{ij}. From the PN we know that once both p_i and p_j are true (marked), p_{ij} becomes true (marked) at the same time. Since p_i and p_j represent formula P_i and P_j respectively, p_{ij} represents the formula $P_i \wedge P_j$.

Based on the table, we can recursively construct the I-net of a component constraint. This process is illustrated in Sec. 3.2.

3.1.2. *Construct O-net of a component constraint*

Constructing an O-net is to map a T-formula to PN. We first map a P formula in each T-node ($G \neg P$ or FP) to a PN and then map the T-node to a PN. Finally, we construct a PN for the T-formula.

Table 1. Construct I net of a component constraint.

Term in I formula	PN representation
Atomic proposition p_i (p_i is a port)	
Compound proposition $P_i \wedge P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively	 p_i PN of P_i T_{ij} p_{ij} PN of P_j p_j
Compound proposition $P_i \vee P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively	 p_i PN of P_i T_i p_{ij} PN of P_j T_j p_j

In Table 2, we list the PN construction for typical terms in a P formula in a T-node. For a compound proposition $P_i \oplus P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively, we introduce two new immediate transitions T_i and T_j, and a new place p_{ij} and then build the PN as shown in the table to represent this formula. The characterization place of the formula is p_{ij}. Based on the table, we can recursively construct the PN of the P formula of a T-node in a component constraint. This process is illustrated in Sec. 3.2.

Table 3 lists the PN representation for G-node, F-node and their composition in a T-node. For example, if O in the component constraint $AG(I \rightarrow O)$ is an F-node $AF\ P$, assuming that the characterization place of the PN of formula P is p, we construct a new place denoted by p_i and a new transition T_i to form the PN as shown in the table to represent the formula. The characterization place of the formula is p_i.

3.1.3. *Combine I-net and O-net*

Once we generate the I-net and O-net of a component constraint, we are ready to construct the PN for the constraint by combining its I-net and O-net. Assume that the characterization places of its I-net and O-net are PI and PO, respectively. We introduce a new immediate transition denoted as Tc to build up the PN as shown in Fig. 3.

Table 2. Construct the PNs of P formulas in T-nodes.

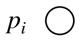
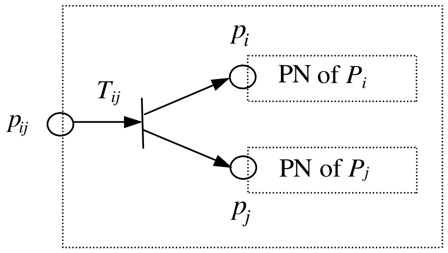
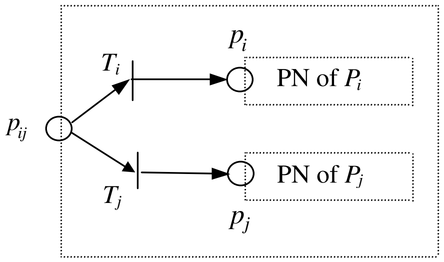
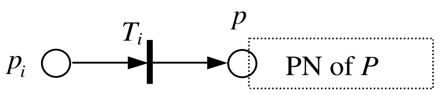
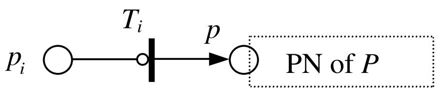
Term in P formula in T-node	PN representation
Atomic proposition p_i (p_i is a port)	
Compound proposition $P_i \wedge P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively	 p_i T_{ij} PN of P_i p_{ij} PN of P_j p_j
Compound proposition $P_i \oplus P_j$, where the characterization places of P_i and P_j are p_i and p_j, respectively	 p_i T_i PN of P_i p_{ij} PN of P_j T_j p_j

Table 3. The PN representation for G-node, F-node and their composition in a T-node.

Term in T-node	PN representation
F-node $AF\ P$, where the characterization place of the PN of formula P is p	
G-node $AG \neg P$, where the characterization place of the PN of formula P is p	

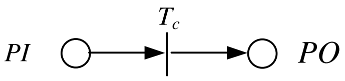


Fig. 3. PN for a component constraint.

3.1.4. *Remove temporary nodes*

The TPN obtained in Sec. 3.1.3 may contain some temporary nodes. To remove them, we developed four rules based on the principle of equal reachability regarding original nodes, shown in Fig. 4. Applying these rules results in the removal

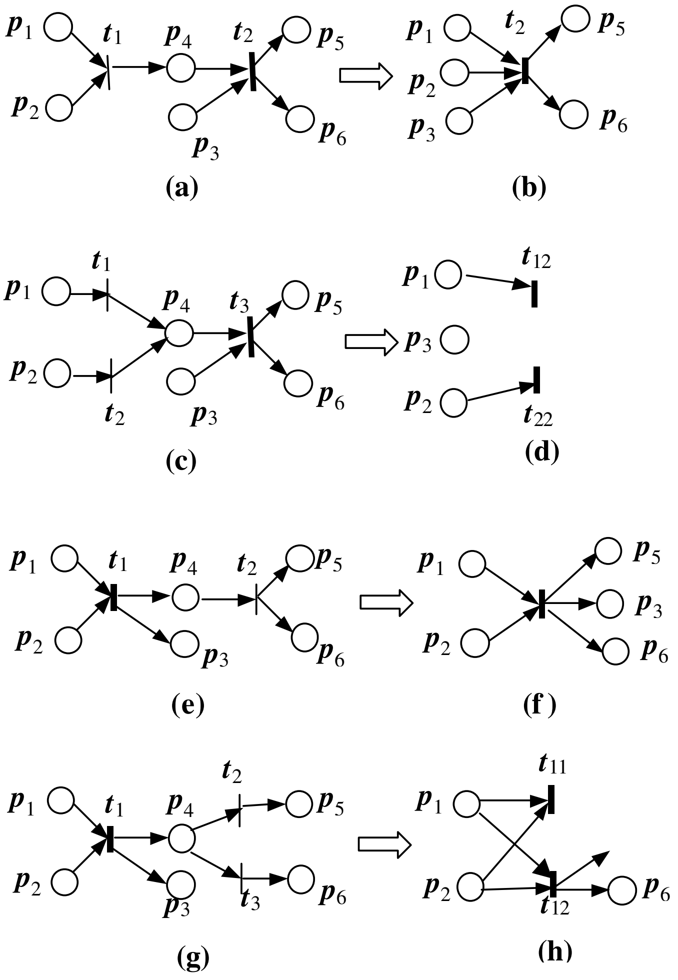


Fig. 4. Rules for removing temporary nodes.

of all temporary nodes introduced in representing a constraint formula by a PN. Particularly, the first rule eliminates temporary nodes introduced by the PN representation of a compound proposition in the form of $P_i \wedge P_j$ in an I-formula. The second rule removes temporary nodes introduced by the PN representation of a compound proposition in the form of $P_i \vee P_j$ in an I-formula. The third rule removes temporary nodes introduced by the PN representation of a compound proposition in the form of $P_i \wedge P_j$ in a P-formula. The fourth rule removes temporary nodes introduced by the PN representation of a compound proposition in the form of $P_i \oplus P_j$ in a P-formula. Besides, applying any of the four rules will also remove the temporary nodes introduced by combining I-net and O-net. In the case that a compound proposition is a mixture of the forms of propositions listed in Table 1 or Table 2, we can jointly apply more than one rule to remove the temporary nodes.

3.2. *Example*

Suppose that the constraint:

$$AG(PT1 \wedge PT2 \wedge PT3 \rightarrow AF\ (PT4 \wedge PT5 \wedge PT6))$$

is defined on the component shown in Fig. 5(a). If the constraint has been verified being true, we can reduce the component based on the logic formula that specifies the constraint. Figure 5(b) shows the I-net corresponding to the constraint, where place p_{12} is the characterization place of formula $PT1 \wedge PT2$, and p_I the characterization place of $PT1 \wedge PT2 \wedge PT3$. Similarly, Fig. 5(c) shows the PN representation of formula $PT4 \wedge PT5 \wedge PT6$, where place p_{45} is the characterization place of formula $PT4 \wedge PT5$, and p_{456} the characterization place of $PT4 \wedge PT5 \wedge PT6$. By introducing to Fig. 5(c) a new place p_0 and temporal transition, we obtain the PN of the T-node $AF\ (PT4 \wedge PT5 \wedge PT6)$ as shown in Fig. 5(d), which is also an O-net of the constraint. Combining Figs. 5(b) and (d) together gives Fig. 5(e). Removing redundant nodes using the rules in Fig. 4 gives Fig. 5(f), which is the reduced PN based on the constraint.

4. Modeling and Analysis of Communication Protocols — An Example

In this section, we illustrate the application of our methodology for constraint propagation and progressive verification to the modeling and analysis of communication protocols. As we know, Ethernet is one of the most commonly used technologies in building computer networks. Many methods have been developed for the evaluation of Ethernet protocol. These methods fall into two classes: one is global models, which take care of strong interactions between all the stations of a network, such as queuing model [1]; the other is isolation models, which describe the complex behavior of each station, but give accurate results when interactions between stations are low, such as the semi-Markov model [17]. We build and analyze an isolation model of the protocol using the methodology described in the following sections.

The transmission protocol in Ethernet is run by two processes [26]:

- The *Transmitter Process* (FTP) manages the different operations of the protocol: data encapsulation, transmission starting, collision handling with Collision Detect Signal, number of attempts increasing, and backoff operations.
- The *Deference Process* (DP) is used to delay the frame transmission when the channel is busy. The DP becomes active when the Carrier Service Signal is switched on. Its busy period ends when the inter-frame spacing operation is done.

4.1. *Architectural model of the ethernet protocol*

As is shown in Fig. 6, its architecture model is composed of two components: TP (transmitter process) and DP (deference process). Descriptions on ports and transitions are given in Table 4.

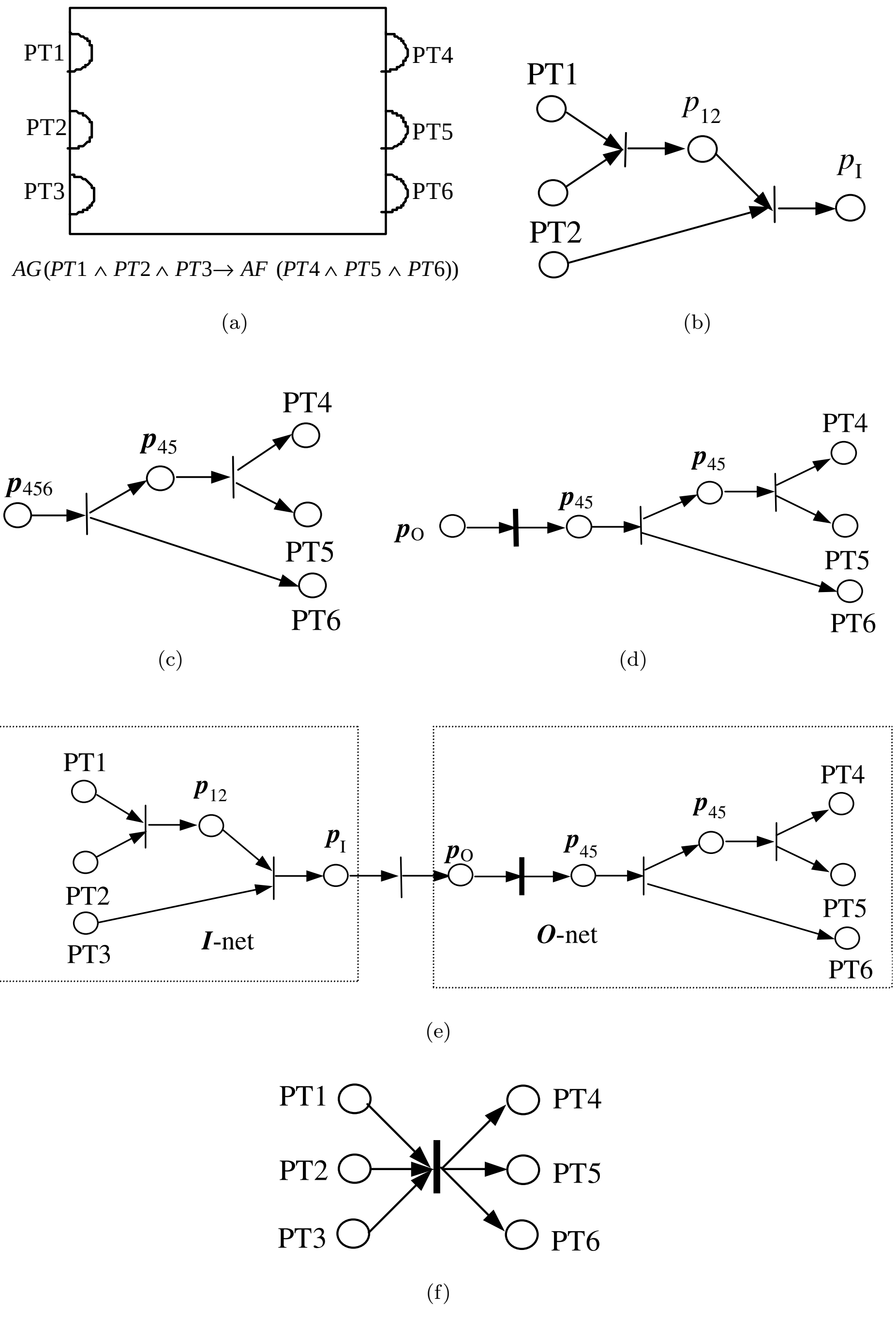


Fig. 5. (a) A component, (b) PN of formula $PT1 \wedge PT2 \wedge PT3$ (I-net of the constraint), (c) PN of formula $PT4 \wedge PT5 \wedge PT6$, (d) PN of T-nod AF ($PT4 \wedge PT5 \wedge PT6$ (O-net of the constraint), (e) PN of the constraint with redundant nodes, (f) Final reduced PN of the component.

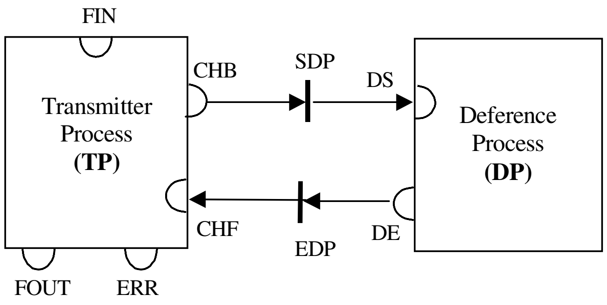


Fig. 6. Architectural model of the Ethernet protocol.

Table 4. Ports and transitions in Fig. 6.

Port	Description
FIN	Acquired channel
FOUT	Acquired channel while deferring busy period
CHB	Channel busy
CHF	Channel free
DS	Deference process started
DE	Deference process ended
Transition	**Description**
SDP	Start deference process
EDP	End deference process

One system-wide property which the design must satisfy is that once a frame transmission is issued, it will eventually be transmitted successfully or an exception of too many transmissions (network overload) is raised. This system-wide constraint can be formally specified based on the relationship between ports FIN, FOUT and ERR as:

$$AG(\mathrm{FIN} \to AF\ \mathrm{FOUT} \oplus \mathrm{ERR})\,. \tag{1}$$

4.2. *Intermediate component constraints*

In order to be able to design a component independent of the rest of a system, it is desirable that there is a set of constraints to completely describe what properties of the component are expected by its environment. Such properties are highly expected when we conduct the design of complex concurrent systems.

Based on the component control flows and a rational consideration of job process timing delays of components, we derived the following intermediate constraints:

TP:

$$AG\ (\mathrm{FIN} \to AF\ \mathrm{FOUT} \oplus \mathrm{ERR} \oplus \mathrm{CHB})\,. \tag{2}$$

(When a frame of message is due for transmission, one and only one of the following three events will happen: (a) the frame is directly transmitted out, (b) an error is detected for too many retransmits, and (c) a deference process is invoked or the frame is directly transmitted out.)

$$AG\ (\mathrm{CHF} \rightarrow AF\ \mathrm{FOUT} \oplus \mathrm{ERR} \oplus \mathrm{CHB})\,. \tag{3}$$

(When a deference process is finished, one and only one of the following three events will happen: (a) the frame is directly transmitted out, (b) an error is detected for too many retransmits, and (c) a deference process is invoked or the frame is directly transmitted out.)

DP:

$$AG\ (\mathrm{DS} \rightarrow AF\ \mathrm{DE})\,. \tag{4}$$

(When a deference process is invoked, the process ends eventually.)

As an important principle of system modeling and design, the derived intermediate constraints should be consistent with the system-wide constraints. Only under this condition, the design and development of components against their component constraints is meaningful. The next section presents such a consistency check procedure.

4.3. *Consistency verification among component constraints and global constraint*

Constructing the constraint models of components in the Ethernet protocol and plugging them into the architecture model, we obtain the architecture constraint model as shown in Fig. 7. In this figure, transitions TP1, TP2 and TP3 are introduced by building the PN representation of the component constraint of Formula (2), and transitions TP4, TP5 and TP6 introduced by building the PN representation of the component constraint of Formula (3). Based on this model, we can easily prove that the system-wide constraint is satisfied (if we put an initial token into place FIN, we know the token will eventually move into place FOUT).

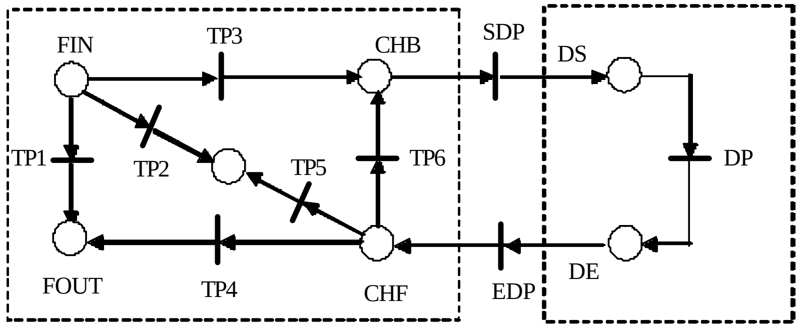


Fig. 7. Architecture constraint model of the Ethernet protocol.

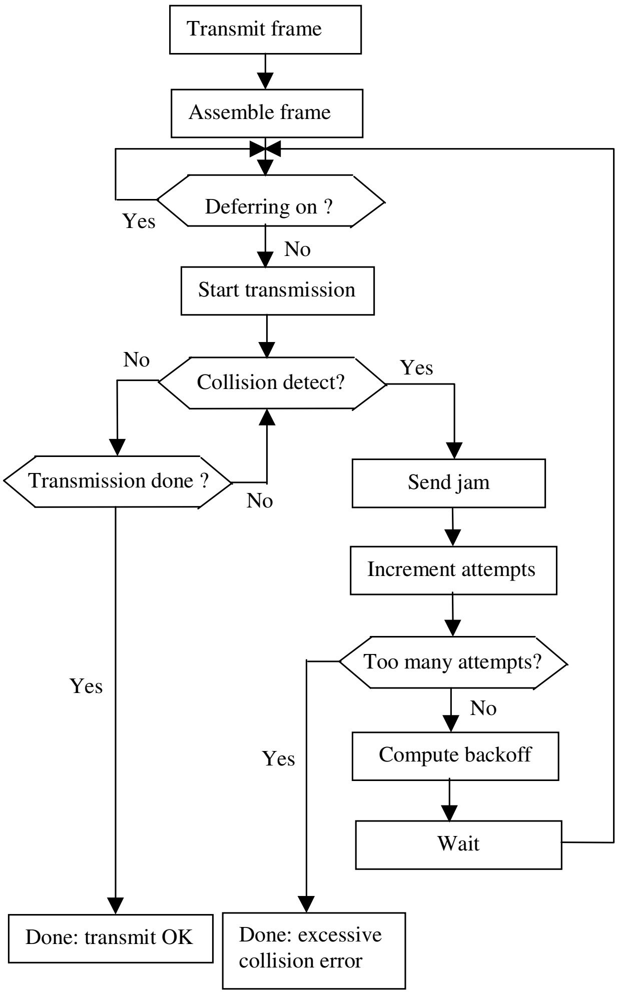


Fig. 8. Frame transmitter process.

4.4. *Component modeling and verification*

Because we have already verified that the derived component constraints are consistent with the system-wide constraints, we are ready to design each component against its component constraints. In this section, based on the control flow of components TP and DP illustrated in Figs. 8 and 9, respectively, we build their PN models, and verify if they satisfy the component constraints defined on them.

4.4.1. *Petri net model of component TP*

The PN model of component TP is shown in Fig. 10. The TP is ready to receive data from the upper layer protocol (place FIN marked), consequently no frame is

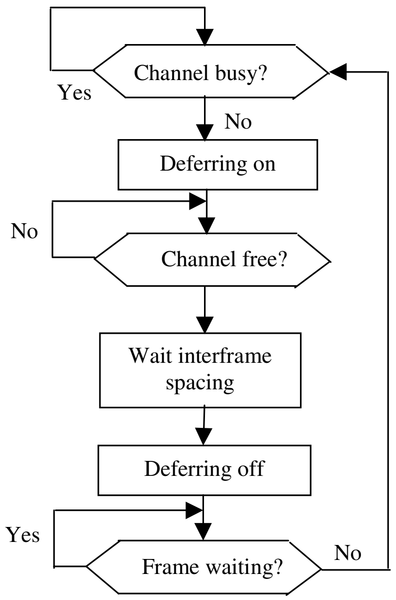


Fig. 9. Deference process.

in Data Link Layer and no frame is waiting to be sent (NFW marked). The TP encapsulates data (transition DE fired) when the Data Link Layer is required for data transmission. Now a frame is waiting to be sent (FW marked). When DP is not differing (NDEF marked) the FTP starts transmission (TST fired) and watches for collision (WTC marked), consequently the frame is no longer waiting (NFW marked again); otherwise the deference process starts (CB fires and CHB marked).

If no collision occurs during the slot time (NCOL fired) then the channel is acquired (AC marked). The TP reinitializes (RI fired) the number of possible attempts to sixteen (NPA marked with sixteen tokens) and ends the concurrent transmission (ECT fired). Then FTP is ready to receive new data (RTD marked).

If at least one collision occurs during the slot time (COL fired) the channel is not acquired (NAC marked) and the number of failed attempts is incremented (one more token from NPA to NFA when COL is fired). If sixteen attempts have failed (sixteen tokens in NFA) FTP is aborted with excessive collision error (ECE fired) and is ready again to receive upper layer data (RTD marked). On the contrary (NECE fired), TP computes backoff delay (BOD marked), and waits for the corresponding time (WBOT fired), then it becomes ready to retransmit the frame (RTR marked). At this point no frame is waiting (NFW marked), TP retransmits the frame (FR fired), and again a frame is waiting to be sent (FW marked). Now, the TP can proceed as before.

Notice that transitions RI and ECE are represented with a box instead of a bar, indicating that they are transitions with “high firing priority”. When both NFA and

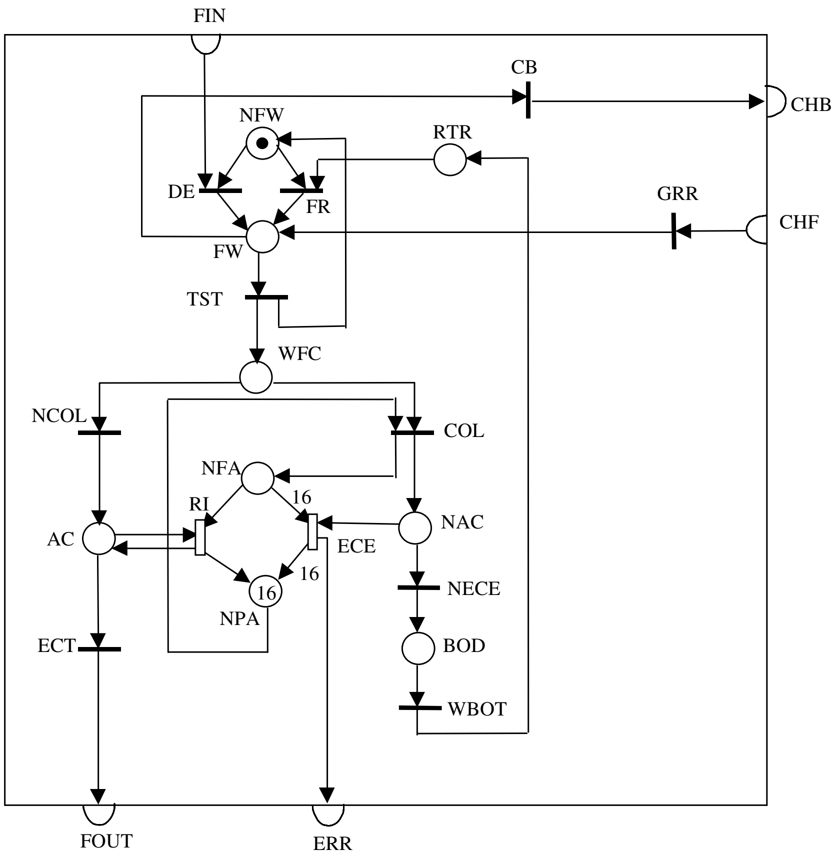


Fig. 10. PN model of the transmitter process.

AC are marked, RI will fire before ECT. The firing of NFA resets the maximum retry attempts after collision to 16 times. Similarly, when NFA contains 16 tokens and NAC is marked, ECE will fire before NECE. The firing of ECE remarks the failure of frame transmission due to excessive collision error.

4.4.2. *Petri net model of component DP*

The DP model is constructed in the same way, which is given in Fig. 11.

4.4.3. *Component verification and discussion*

To verify the component constraint of formula (2) defined on TP, we put an initial token into place FIN at the PN model of Fig. 10, which, combining with the token distribution in other places, indicates that a new frame arrives at the transmitter for transmission, and the transmitter is ready to handle it. We can use the reachability analysis method for PNs to trace the progress of the PN states. Doing so, we

Table 5. Legends of places and transitions of the GSPN in Fig. 10.

Place	Description
AC	Acquired channel
BOD	Backoff delay
CHB	Channel busy
CHF	Channel free
FIN	Frame in
FOUT	Frame out
FW	Frame waiting
NAC	Non acquired channel
NFA	Number of failed attempts
NFW	No frame waiting
NPA	Number of possible attempts
RTD	Ready to transmit data
RWIS	Ready to wait interframe spacing
RTR	Ready to retransmit
WFC	Watch for collision
ERR	Error for too many retransmits
Transition	**Description**
CB	Channel found busy
COL	Collision happens
COLD	Collision while deferring busy periods
DE	Data encapsulation
DST	Deferring slot time
ECE	Excessive collision error
ECT	End of current transmission
FR	Frame to transmit
GRR	Get ready for retransmitting
NCOL	No collision
NCOLD	Collision while deferring busy periods
NECE	Non excessive collision error
RI	Retransmit
SDEF	Start deference process
TST	Transmitting slot time
WBOT	Wait Backoff time
WIFS	Wait interframe spacing

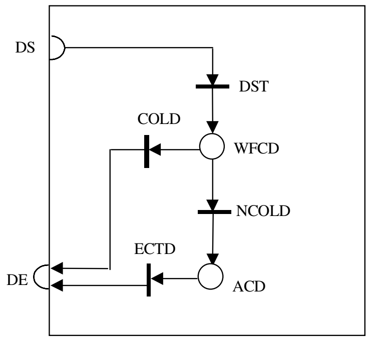


Fig. 11. PN model of the deference process.

Table 6. Legends of places and transitions of the PN in Fig. 11.

Place	Description
ACD	Acquired channel while deferring busy period
WFCD	Watch for collision while deferring busy period

Transition	Description
DST	Deferring slot time
NCOLD	Collision while deferring busy periods
ECTD	End of concurrent transmission while deferring busy period

find that the net will end at three possible absorbing states. The first state is characterized by the fact that the token initially stored in FIN disappears, place FOUT is added a token, and the token distribution in all other places remains unchanged. The second state is characterized by the fact that the token initially stored in FIN disappears, place ERR is added a token, and the token distribution in all other places remains unchanged. The third state is characterized by the fact that the tokens initially stored in FIN and NFW disappear, place CHB is added a token, and the token distribution in all other places remains unchanged. This result is consistent with the component constraint of formula (2). If we initially put a token into place CHF and remove the token from NFW, the reachability analysis will lead to the same three absorbing states, which is consistent with the component constraints of formula (3). In the same way, we can easily verify the constraint defined on component DP. The component constraints can also be verified by using a model checking tool.

5. Conclusion

System assembly is one of the major problems in engineering large-scale component-based systems. This is especially true for heterogeneous, COTS and GOTS-based distributed systems, which are typical in industrial applications. The problem of system assembly is not only to make constituent components work together, but also to ensure that the components as a whole behave consistently and guarantee certain end-to-end properties. To achieve these, a theory of software system constraints about components, their environment and about the system as a whole is the necessary foundation toward solid understanding of the composability of component-based systems.

Toward this end, we have presented an approach to use software system constraints and consistent constraint propagation to drive progressive verification of system design and refinement. For general systems, decomposition of a global constraint of a system to its constituent components is dictated by the way the system is decomposed, which is a delicate design decision and cannot be automated based on today's technology. Consequently, how to ensure consistency between system-wide constraints and decomposed component constraints under a given composition architecture is an important issue that must be addressed before component constraints can be used meaningfully to guide component design and analysis. Our approach provides a systematic way to verify the consistency by leveraging the structure of system composition, which can be automated. We have shown that such consistent constraint propagation allows us to progressively verify system-wide properties by means of showing individual components satisfying their corresponding component constraints. Because no composition of the results of component verification is necessary, it provides an effective means to manage the complexity of analysis. In terms of the verified properties, this approach allows a component model in the design to be safely replaced with interface- and constraint-conforming alternative models without requiring re-verification of overall design.

In this paper, Computational Tree Logic and Petri Nets are used as the underlying formalisms to describe system constraints and design compositions, respectively. The methodology presented here, however, is independent of individual formal notations, and can thus be used in conjunction with other appropriate formal notations.

We believe that the approach presented in this paper represents a positive step toward a methodology for formal design and analysis of dependable and predictable software systems. A unified constraint theory is needed to provide taxonomy of major types of constraints and their properties. In addition to the type of constraints described in this paper, research is needed to explore how to use the presented approach to address other type(s) of constraints. Furthermore, we will study how to combine our design level constraints verification with individual component implementation verification. These are subjects of ongoing and future investigation.

Acknowledgements

This work was supported in part by the National Science Foundation under Grants CCR-0098120, CCR-9988326, HRD-0317692, and National Science Council under Grant NSC 92-2213-E-468-001. We thank two anonymous reviewers for their helpful comments.

References

1. G. T. Almes and E. D. Lasoweka, The behavior of ethernet-like computer communication networks, *Proc. 7th Symp. Operating System Principles*, December 1979.
2. G. S. Bools and R. C. Jeffrey, *Computability and Logic*, 3rd edn. (Cambridge University Press, 1989).
3. R. Boyer and J. Moore, *A Computational Logic* (Academic Press, New York, 1979).
4. G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edn. (Benjamin/Cummings, 1994).
5. L. S. Cauman, *First-Order Logic* (Walter de Gruyter, Berlin, 1998).
6. E. M. Clarke, O. Grumberg *et al.*, Model checking and abstraction, *ACM Trans. Programming Languages and Systems* **16**(5) (1994) 1512–1542.
7. E. M. Clarke, O. Grumberg and D. A. Peled, *Model Checking* (MIT Press, Cambridge, MA, 1999).

8. S. C. Cheung and J. Kramer, Context constraints for compositional reachability analysis, *ACM Trans. Software Engineering and Methodology* **5**(4) (1996) 334–377.
9. Y. Deng, J. Wang, J. Tsai and K. Beznosov, An approach for modeling and analysis of security system architectures, *IEEE Trans. Knowledge and Database Engineering* **15**(2) (2003) 1099–1119.
10. E. A. Emerson, A. K. Mok, A. P. Sistla and J. Srinivasian, Quantitative temporal reasoning, *Concurrent Systems* **4** (1992) 331–352.
11. E. A. Emerson and A. P. Sistla, Symmetry and model checking, in *Proc. 5th Int. Conf. on Computer Aided Verification*, Crete, Greece, 1993.
12. I. Graham, *Migrating to Object Technology* (Addison-Wesley, 1995).
13. P. Inverardi and A. Wolf, Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Trans. Software Eng.* **21**(4) (1995) 373–386.
14. D. C. Luckham, J. Kenney, L. Augustin *et al.*, Specification and analysis of system architecture using Rapide, *IEEE Trans. Software Eng.* **21**(4) (1995) 336–355.
15. K. L. McMillan, *Symbolic Model Checking* (Kluwer Academic Publishers, Boston, 1993).
16. B. Meyer, *Object-Oriented Construction* (Prentice Hall, 1988).
17. I. Mitrani and E. Gelenbe, Control policies in CSMA/CD local area network: Ethernet controls, Report Inria, 1981.
18. T. J. Mowbray and W. A. Ruh, *Inside CORBA — Distributed Object Standards and Applications* (Addison-Wesley, 1997).
19. N. Medvidovic and R. Taylor, A classification and comparison framework for software architecture description languages, *IEEE Trans. Software Eng.* **26**(1) (2000) 70–93.
20. T. Murata, Petri nets: Properties, analysis and applications, *Proc. IEEE* **77**(4) (1989) 541–580.
21. J. Rumbaugh, M. Blaha, W. Premelani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design* (Prentice Hall, 1991).
22. W. Reisig, *Petri Nets — An Introduction* (Springer-Verlag, Berlin, 1985).
23. J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual* (Addison-Wesley, 1999).
24. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, Abstractions for software architecture and tools to support them, *IEEE Trans. Software Engineering* **21**(4) (1995) 313–335.
25. B. Thuraisingham and W. Ford, Security constraint processing in a multilevel secure distributed database management system, *IEEE Trans. Knowledge and Data Engineering* **7**(2) (1995) 274–293.
26. J. Wang, *Timed Petri Nets: Theory and Application* (Kluwer Academic Publishers, Boston, MA, 1998).
27. J. Wang, X. He and Y. Deng, Introducing software architecture specification and analysis in SAM through an example, *Information and Software Technology* **41**(7) (1999) 451–467.
28. J. Warmer and A. Kleppe, *The Object Constraint Language — Precise Modeling with UML* (Addison-Wesley, 1999).
29. J. Wang, Y. Deng and G. Xu, Reachability analysis of concurrent systems based on time Petri nets, *IEEE Trans. Systems, Man and Cybernetics, Part B* **30**(5) (2000) 725–736.

Appendix 1. Overview of Petri Net Notation

A Petri net [20, 22] is a 5-tuple $N = (P, T, B, F, M_0)$, where

$P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places* ;

$T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*, $P \cup T \neq \emptyset$, and $P \cap T = \emptyset$;

B: $(P \times T) \rightarrow N$ is a *backward incidence function* or *input function* that defines directed arcs from places to transitions, where N is the set of nonnegative integers;

F: $(P \times T) \rightarrow N$ is a *forward incidence function* or *output function* that defines directed arcs from transitions to places; and

M_0: $P \rightarrow N$ is the *initial marking.*

In terms of graphical representation, a place is denoted by a circle; and a transition by a box. Places and transitions are connected by directed arcs which specify the data or control flow. The execution of a Petri net is reflected by the change of *marking*, which is an assignment of *tokens* to the places. The change of markings represents the shifting of states of a system, which is triggered by firing transitions.

A transition is able to fire only if it is enabled. A transition is said to be enabled if each input place of it has no less tokens than the weight of the arc from the place to it, i.e., $M(p) \geq B(t, p)$ for any p in P. A Petri net executes by firing enabled transitions. A firing of an enabled transition t removes from each input place p the number of tokens equal to the weight of the directed arc connecting p to t. It also deposits in each output place p the number of tokens equal to the weight of the directed arc connecting t to p. Mathematically, firing t at M yields a new marking

$$M'(p) = M(p) - I(t, p) + O(t, p) \quad \text{for any } p \text{ in } P.$$

As a mathematical tool, Petri nets possess a number of properties. These properties, when interpreted in the context of the modeled system, allow the system designer to identify the presence or absence of the application domain specific functional properties of the system under design. Two types of properties can be distinguished, behavioral and structural ones. The behavioral properties are those which depend on the initial state or marking of a Petri net. The structural properties, on the other hand, do not depend on the initial marking of a Petri net. They depend on the topology, or net structure, of a Petri net. From the practical point of view, we are more interested in behavioral properties, such as reachability, boundedness, conservativeness, and liveness.

Appendix 2. Overview of Computational Tree Logic

CTL is a propositional branching time temporal logic [10, 11]. CTL formulas are built up from atomic propositions, propositional connectives, and temporal modalities. CTL formulas use four temporal operators: Fp ("eventually p"), Gp ("always p"), Xp ("next time p"), and pUq ("p until q"), and two path quantifier: A ("for all futures") and E ("there exists"). CTL formulas are defined recursively as follows:

- Each atomic proposition p is a formula.
- If p and q are formulae, so are $p \wedge q$ and $\neg p$.
- If p and q are formulae, so are $A(p\ U\ q)$, $E(p\ U\ q)$, and $EX\ p$.

A CTL formula is interpreted with respect to a temporal structure $M = (S, R, L)$, where S is a set of states, R is a binary relation on S that is total, and L is a labeling which assigns to each state a set of atomic propositions, those intended to be true at the state. Intuitively, this temporal structure M represents the reachability graph of the architecture. A full-path $x = s_0, s_1, s_2, \ldots$ in M is an infinite sequence of states such that $(s_i, s_{i+1}) \in R$ for each i; intuitively, a full-path captures the notion of an execution sequence.

Some other basic modalities of CTL are defined as abbreviations: $AF\ q$ abbreviates $A(\mathit{true}\ U\ q)$ and $EF\ q$ abbreviates $E(\mathit{true}\ U\ q)$. We also define the modality G as the dual of $F^{\leq k}$, i.e., $AG\ q$ abbreviates $\neg EF \neg p$, and $EG\ q$ abbreviates $\neg AF \neg p$. Formula $AG(p \rightarrow AF\ q)$ says that whenever p is true, q is guaranteed true in the future. $AG(\mathrm{p} \rightarrow EX\ q)$ says that whenever p is true, q is guaranteed true next. $AG(p \rightarrow q\ U\ r)$ says that whenever p is true, q is true until r becomes true. Below we list some examples to show the descriptive power of CTL formulae:

- *Safety property.* $AG\ p$ denotes that the constraint p always holds.
- *Liveness property.* $AF\ p$ denotes that constraint p eventually holds.
- *Sequential relationship* among events. For example, every X is followed by a Y, and then a Z. The corresponding RTCTL constraint is:

$$AG\ (X \rightarrow (AG(\neg Z\ U\ Y) \wedge AF(Z)))\,.$$