# Automated Modeling of Dynamic Reliability Block Diagrams Using Colored Petri Nets

Ryan Robidoux, Haiping Xu, *Senior Member*, *IEEE*, Liudong Xing, *Senior Member*, *IEEE*, and MengChu Zhou, *Fellow*, *IEEE*

*Abstract*—**Computer system reliability is conventionally modeled and analyzed using techniques such as fault tree analysis (FTA) and reliability block diagrams (RBD), which provide static representations of system reliability properties. A recent extension to RBD, called dynamic reliability block diagrams (DRBD), defines a framework for modeling dynamic reliability behavior of computer-based systems. However, analyzing a DRBD model in order to locate and identify design errors, such as a deadlock error or faulty state, is not trivial when done manually. A feasible approach to verifying it is to develop its formal model, and then analyze it using programmatic methods. In this paper, we first define a reliability markup language (RML) that can be used to formally describe DRBD models. Then we present an algorithm that automatically converts a DRBD model into a colored Petri net (CPN). We use a case study to illustrate the effectiveness of our approach and demonstrate how system properties of a DRBD model can be verified using an existing Petri net tool. Our formal modeling approach is compositional, thus it provides a potential solution to automated verification of DRBD models.**

*Index Terms*—**System reliability, reliability block diagram (RBD), extensible markup language (XML), colored Petri net (CPN), time Petri net, formal modeling and analysis, automated verification, deadlock detection.**

## NOMENCLATURE

| | |
|---|---|
| API | Application programming interface. |
| BNF | Backus-Naur form. |
| CPN | Colored Petri net. |
| DFTA | Dynamic fault tree analysis. |
| DOM | Document object model. |
| DRBD | Dynamic reliability block diagram. |
| FTA | Fault tree analysis. |
| PDP | Piecewise deterministic Markov processes. |
| RBD | Reliability block diagram. |
| RML | Reliability markup language. |
| SAX | Simple API for XML. |
| SDEP | State-based dependency controller. |
| SPARE | Spare part controller. |
| SRBD | State-based reliability block diagram. |
| XML | Extensible markup language. |

## I. INTRODUCTION

IN OUR modern society, there is an increasing reliance on computer-based systems that control critical infrastructures such as telecommunication networks, banking systems, and nuclear power plants. Such infrastructures are critical because the failure of the supporting computer-based systems (e.g., interrupted phone service, financial loss, and nuclear meltdown) can be catastrophic [1]. Therefore, ensuring the reliability of such systems has become a growing need in the computing world. There are many existing methods that can be used to evaluate a system's reliability, such as measuring a system's mean time to failure. In order to efficiently evaluate or predicate a system's reliability performance, an effective system reliability model is required. Most reliability modeling approaches are based on statistical methods. Their typical examples are reliability block diagram (RBD), fault tree analysis (FTA), and Markov chains [2]. The above methods, however, can only provide system reliability models where a system component must be either active or failed; thus, they are very limited in their ability to accurately model a system's dependency relationships and dynamic reliability properties. Dynamic FTA (DFTA) is another modeling tool that can support modeling a functional dependency in a system, where the failure of a component causes some other dependent components to become inaccessible or unusable [3]. However, DFTA cannot be used to model a general state-based dependency relationship between components, e.g., a state-based dependency where the activation of a component causes the deactivation of a dependent one.

Recently, an extension to RBD, called dynamic reliability block diagram (DRBD) [4-5], was introduced with new controller constructs that support modeling dynamic, dependent and redundant relationships between components in a computer-based system. Although it has been shown that the DRBD approach is very effective in modeling a system's dynamic reliability properties, subtle flaws in it can be easily introduced due to its modeling complexity. Therefore, formal verification of a DRBD model is an essential step in developing a correct system reliability model for the

This work was supported in part by the College of Engineering, University of Massachusetts Dartmouth.

R. Robidoux is with the Computer and Information Science Department, University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA (e-mail: u_rrobidoux@umassd.edu).

H. Xu is with the Computer and Information Science Department, University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA (corresponding author, phone: 508-910-6427; fax: 508-999-9144; e-mail: hxu@umassd.edu).

L. Xing is with the Electrical and Computer Engineering Department, University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA (e-mail: lxing@umassd.edu).

M. C. Zhou is with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA (e-mail: zhou@njit.edu).

evaluation of a system's reliability. In our recent work, we demonstrated some preliminary results on how to formally verify a DRBD model using colored Petri nets (CPN) [4-5], which provide the formal semantics for DRBD models. The approach follows the basic philosophy of recent efforts on converting a UML (Unified Modeling Language) diagram to Petri nets for formal verification [6]. However, the proposed approach is not compositional for formal modeling, and does not provide a generalized solution to automatically convert a DRBD model into CPN. In this paper, we present an algorithm that supports automatic conversion of a DRBD model into CPN. Hence, DRBD's automatic verification can be accomplished by analyzing the state space of the CPN using existing CPN tools. Note that since our proposed formal modeling approach is compositional, our approach scales with the sizes of DRBD models.

The rest of the paper is organized as follows. Section II summarizes the related efforts in reliability modeling. Section III provides a formal definition of DRBD with its embedded state-based RBD (SRBD). In order to efficiently process DRBD models, an XML-based language, called the reliability markup language (RML), is introduced to represent a DRBD model in XML format. Section IV outlines the procedures to convert DRBD into CPN. Section V provides a case study that illustrates how to create a DRBD model and convert it into a CPN model for formal analysis. Finally, Section VI presents the conclusions and future work.

## II. Related Work

Reliability modeling is an integral step in creating reliable and fault-resistant computer-based systems. Currently, many industries require that some form of qualitative system reliability analysis be integrated into the design phase of a computer-based system [3]. One of the major analysis approaches for system reliability is FTA, which provides a detailed analysis of a system's failure probabilities. Fault trees are logic diagrams that depict potential, critical events within a system. A fault tree model represents the relationship between a critical event and the reasons for the event's occurrence, such as specific component failures [7]. Since FTA does not account for dynamic system properties, it is extended into dynamic FTA (DFTA) in order to model dynamic relationships between components [3, 8]. DFTA introduces additional gates for modeling sequential and sparing behavior, but it has limited ability to model complex systems that involve dynamic component dependency such as a general state-based dependency [4]. On the other hand, an RBD represents a network of system components and their connections [2]. The network consists of an input point and output point, a number of blocks representing system components, and multiple paths from the input point to output point. The multiple paths represent successful system operations, where an interruption of these paths may lead to the failure of the whole system [9]. Therefore, an RBD model represents the static topology of a computer-based system's reliability, where the topology can be a serial, parallel or hybrid structure. Contrary to FTA, RBD models are success-oriented networks that describe the function of a system by probabilistic means [2]. Component blocks in an RBD are arranged to illustrate the proper combinations of working components that keep the entire system operational. Failure of a component can be represented by removing the component as well as its connections with other components from the network. When a sufficient number of components in a system fail, the whole system may also fail if there is no connection between the input and output point.

Additional related work on system reliability modeling can be summarized as follows. The SHARPE (symbolic hierarchical automated reliability and performance evaluator) tool expands the use of Markov models in reliability verification of computer systems [10]. Sahner and Trivedi recognize that Markov models can capture important dynamic system behavior, but may also grow exponentially with the number of system components. Their research produces a hierarchical modeling technique for analyzing complex reliability models, which allows for the flexibility of Markov models where necessary, and retains the efficiency of combinatorial solutions where possible. Leangsuksun, *et al.* adopt UML technology to model the reliability of two-tier computer systems [11]. They use UML deployment diagrams to model system components and their relationships, and manually create failure and repair rate for components in order to construct statistical fault trees and Markov Chain models. The system reliability is then calculated using the SHARPE tool. Similarly, Dammag and Nissanke also propose a visual model, called *Safecharts*, which can be used to specify and design safety critical systems [12]. The novel feature of Safecharts is its safety annotation that provides an explicit ordering of states according to risk levels. In order to support standards compliance testing and verification for safety-critical systems, Hsiung, *et al.* attempt to integrate Safecharts into VERTAF (Verifiable Embedded Real-Time Application Framework), which is an application framework for design and verification of embedded real-time software [13]. Blake, *et al.* use an extension of Markov models to specify the reliability of multiprocessor systems using parametric sensitivity analysis [14]. Their approach creates an upper and lower bound for each system parameter of interest in order to compute the optimistic and conservative bounds for the reliability of a multiprocessor system. Similar to the FTA and RBD approaches, most of the above methods only consider a system component as a bi-state component, which must be either active or failed. Therefore, they suffer from the same weakness as FTA and RBD models for modeling dynamic system reliability properties. In our previous work, we propose dynamic RBD (DRBD) as an extension to RBD models [4-5]. New modeling constructs have been introduced and formally specified in Object-Z formalism [15], and can be used to model dynamic reliability properties of system components, e.g., state-based dependency and spare part relationships. Unlike DFTA, DRBD models are defined upon state-based components where a component can be active, standby or failed. Thus, DRBD controlling constructs support modeling general state-based dependencies. Reference [5] gives an introduction to DRBD models as well as additional related work on system reliability modeling.

Petri nets [16-18] have been widely used in industry for modeling and analyzing computer-based systems such as intelligent mobile robots and semiconductor manufacturing systems [19-20]. There is some related work to our approach that uses Petri nets for deadlock detection and avoidance. Fanti and Zhou give a survey on state-of-the-art modeling and deadlock control methods for discrete manufacturing systems based on digraphs, automata, and Petri net approaches [21]. They present the updated results in the areas of deadlock prevention, detection and recovery, and avoidance. Li, *et al.* develop a methodology to synthesize supervisors for a special class of Petri nets that can be used to model flexible manufacturing systems [22]. In their research, a mixed integer programming based deadlock detection technique is used to find minimal siphons efficiently in a plant model. Hsieh formulates a fault-tolerant deadlock avoidance controller synthesis problem for assembly processes based on a class of Petri nets [23]. He proposes a fault-tolerant deadlock avoidance approach that consists of two algorithms, namely a nominal algorithm to avoid deadlocks for nominal system state and an exception handling algorithm to deal with resource failures. Wu and Zhou propose a novel control policy for deadlock avoidance for automated guided vehicle system using colored resource-oriented Petri nets, and the complexity of deadlock avoidance for the whole system is bounded by the complexity in controlling the system [24]. More recently, Li, *et al.* summarize a variety of Petri net based deadlock prevention policies for flexible manufacturing systems [25]. Their work facilitates engineers in choosing a suitable method for their industrial applications. They further suggest developing polynomial algorithms in order to improve the computational efficiency of deadlock prevention methods that are based on the theory of regions.

Although the above Petri net based approaches can be used for deadlock detection and avoidance, they are not aimed at modeling system reliability. A few efforts on reliability modeling using Petri nets can be summarized as follows. Bobbio, *et al.* use the generalized stochastic Petri net (GSPN) to support system dependability analysis [26]. Their approach involves converting fault trees into a GSPN model for the purpose of obtaining both qualitative and quantitative analysis results for the modeled system. Everdij and Blom develop piecewise deterministic Markov processes (PDP) models using dynamically colored Petri nets (DCPN) [27]. They show that DCPN has similar modeling power to PDP, and is more powerful than deterministic and stochastic Petri nets. Petri nets are also applied in safety analysis of a system as shown by Leveson and Stolzy, where Petri nets are used to design and analyze the safety and fault tolerance of a system [28]. Using timed Petri nets, they prove that paths to high risk states can be removed based on reachability analysis. Buy and Sloan propose a method to automatically analyze the timing properties of concurrent systems [29]. Their method uses simple time Petri nets to analyze concurrent software systems developed in Ada. Ghezzi, *et al.* introduce a high-level Petri net formalism, called ER nets (environment/relationship nets) to model time critical software systems [30]. They prove that ER nets can provide a satisfactory solution to analyzing the timing and functionality of such systems. While the above approaches are similar to our research efforts using Petri nets, they are not concerned with formalizing dynamic reliability properties of a computer system, such as a state-based dependency. Furthermore, instead of providing quantitative analysis of system reliability directly using Petri nets, our approach currently focuses on using colored Petri nets (CPN) [31] to verify the correctness of a DRBD model, namely the safety properties and liveness properties [32] of the corresponding system. Although there are many previous efforts for formal modeling and analysis of various systems using Petri nets [33-37], automated system modeling using colored Petri nets is rare. As we demonstrate in the case study in Section V, it is vital to provide an automated mechanism to ensure the correctness of a DRBD model because a DRBD model can become complicated when dynamic reliability properties are involved.

## III. Dynamic Reliability Block Diagram

The novelty of DRBD is its ability to model dynamic system reliability behaviors such as state-based dependency and redundancy [4]. The DRBD approach introduces new controller blocks, such as *SDEP* (state-based dependency controller) and *SPARE* (spare part controller) for modeling state-based dependency and spare part relationships, respectively. A DRBD model consists of a state-based RBD (SRBD) and a number of controller blocks. SRBD is an extension to RBD where each component is associated with a state representing the activeness of the component in the system. An SRBD model defines the static structure of a DRBD model, while the controller blocks model the dynamic reliability properties of the system. The DRBD designs described in this paper follow the notations and constructs introduced in [4-5].

### *A. State-Based Reliability Block Diagram*

An SRBD is a network of dynamic system components called structural components. As defined in Fig. 1 in a Backus-Naur form (BNF), a structural component can be one of the three component types, namely simple component, parallel component and serial component. Simple components are a special case of structural components, which represent atomic and physical system components with a state. A component with a state can be formally defined as a finite state machine consisting of three states, “Active”, “Standby” and “Failed”, which may change at runtime. An “Active” component is an online component that is actively performing tasks. A component in a “Standby” state is ready to perform tasks, but it is still waiting to be set online. A “Failed” component is no longer online and cannot work properly. The two other structural component types are used to define the topology of a DRBD. In Fig. 1, parallel components and serial components are defined as sets of structural components sandwiched between the tags `<parallel>…</parallel>` and `<serial>…</serial>`, respectively. The state of a structural component can be logically determined by aggregating the states of its contained components. Contained structural components within a parallel component (i.e., simple or serial components) can operate in parallel; therefore, only one of them must be in an "Active" state for the parallel component to be considered as active. A failed parallel component indicates that all of its contained structural components are in "Failed" states. Conversely, a serial component is not considered as active unless all of its contained structural components (simple or parallel component) are in "Active" states because the failure of any of its contained components leads to the failure of the whole serial component. Note that according to the definition of SRBD in Fig. 1, a serial component may contain only one component; thus, an SRBD with a single simple or parallel component can also be viewed as a serial component.

```
<srbd> ::= <structural component>
<structural component> ::= <simple component>
     |<serial component>|<parallel component>
<simple component> ::= <simple>
     <component id><component state></simple>
<component id> ::= <string>
<component state> ::= <Active>|<Standby>|<Failed>
<serial component> ::= <serial>
     <simple or parallel component>
     {<simple or parallel component>}</serial>
<simple or parallel component> ::=
     <simple component>|<parallel component>
<parallel component> ::= <parallel>
     <simple or serial component>
     <simple or serial component>{<simple or
      serial component>}</parallel>
<simple or serial component> ::=
     <simple component>|<serial component>)
```

Fig. 1. Definition of SRBD in Backus–Naur form (BNF).

Fig. 2 shows an example of an SRBD model. In this example, two simple components ($C1$ and $C2$) are contained within a serial component, which itself is contained in a parallel component along with a third simple component ($C3$). Note that if not specified explicitly, we assume that all simple components are initially in "Active" states.

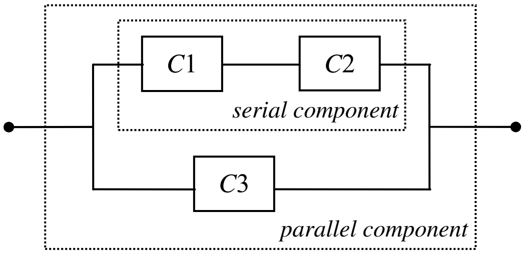


Fig. 2. An example of a state-based reliability block diagram.

### B. DRBD Controller Blocks

Controller blocks defined in a DRBD model can be used to model dynamic relationships between components. Fig. 3 shows the formal definition of a DRBD model with two types of controllers, spare and state controllers, in a BNF format. Note that additional types of controllers, e.g., a load sharing block [5], can also be formally specified in a similar way.

```
<drbd> ::= <srbd><controller>{<controller>}
<controller>::= <spare controller>
       |<state controller>|...
<spare controller> ::= <spareCon><primary event>
       <spare event>{<spare event>}</spareCon>
<primary event> ::= <primary component>
       (<Deactivation>|<Failure>)
<primary component> ::= <simple component>
<spare event> ::= <spare component><Activation>
<spare component> ::= <simple component>
       <ordering number><sparing configuration>
<ordering number> ::= <natural number>
<sparing configuration > ::= <cold>|<warm>|<hot>
<state controller> ::= <stateCon><trigger event>
      <target event>{<target event>}</stateCon>
<trigger event> ::= <trigger component><event>
<trigger component> ::= <simple component>
       |<spare component>
<target event> ::= <target component><event>
<target component> ::= <simple component>
       |<spare component>
<event> ::= <Activation>|<Deactivation>|<Failure>
...
```

Fig. 3. Definition of DRBD in BNF.

A spare controller can be used to model redundant system behavior, where $n$ spare components ($n > 0$) are used to back up a primary component. The deactivation or failure of the primary component (i.e., the primary event) triggers the first spare component to enter an "Active" state. Similarly, the deactivation or failure of the first spare component triggers the second spare one to enter an "Active" state, and so on. The activation of a spare component is called a spare event, while the event of deactivation or failure of a spare component is implicitly defined. A spare component is a simple component with an ordering number and a sparing configuration. The ordering number of a spare component is defined as a natural number, and the standby spare component with the lowest ordering number should always be activated first when a primary component or a spare component is deactivated or failed. The sparing configuration signifies the "activeness" of a spare part. There are three types of sparing configurations, namely *hot*, *cold* and *warm*. A hot spare component operates in synchrony with a primary (i.e., online) component, and is prepared to take over at any time; while a cold spare component is unpowered until needed to replace a faulty component [38]. A warm spare component is a tradeoff between hot and cold configuration in terms of reconfiguration time and power consumption. Without loss of generality, in this paper, we assume that all spare components used in our examples are cold spares.

Fig. 4 (a) illustrates a *SPARE controller block* with a primary component, $P1$, and two cold spares, $S1$ and $S2$ with ordering numbers 1 and 2, respectively. In this example, the first spare part *S*1 is activated if *P*1 fails, and *S*1's failure leads to the activation of the second spare component *S*2. Note that the capitalized letter "*C*" at the upper right corner of blocks *S*1 and *S*2 denotes that both are cold spares.

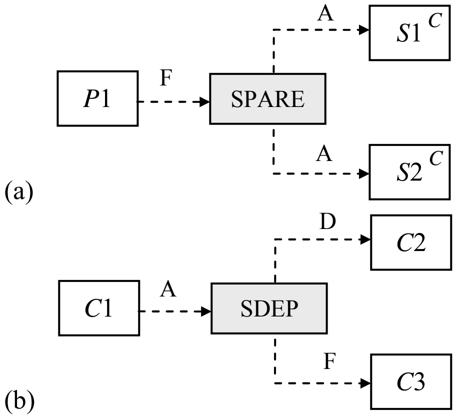
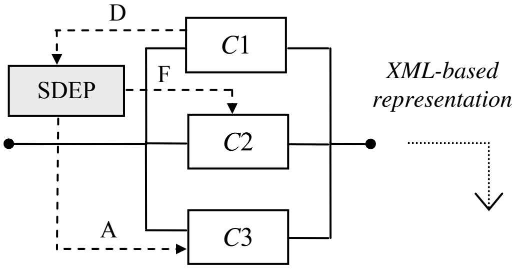


Fig. 4. (a) SPARE controller block. (b) SDEP controller block.

On the other hand, an *SDEP controller block* models the state-based dependency relationships between components in a system. With an SDEP controller block, a *trigger event* due to change of state on a *trigger component* leads to *target events*, which are state changes on *target components*. Both a trigger and target component can be a simple or spare component, and the number of target components must be greater than zero. An event can be one of the three types, namely "Activation," "Deactivation," and "Failure." An "Activation" event happening on a simple or spare component causes it to enter an "Active" state. Similarly, a "Deactivation" or "Failure" event happening on a simple or spare component causes the component to enter a "Standby" or "Failed" state, respectively. Fig. 4 (b) shows an example in which the activation of *C*1 leads to the deactivation and failure of *C*2 and *C*3, respectively. Note that both *C*2 and *C*3 are initially assumed in "Active" states, and otherwise, the states of *C*2 and *C*3 may remain unchanged when *C*1 is activated.

### *C. DRBD Model in Reliability Markup Language*

The reliability markup language (RML) is an XML-based schema defined to formally describe the components, structure and dynamic behavior of a DRBD. RML is designed based on the BNF definition of DRBD models. All components and controllers in a DRBD model have nested RML elements that describe their properties according to their respected BNF definitions. Fig. 5 shows a DRBD model with three parallel simple components *C*1-*C*3, which are dependent on each other. The *SDEP* controller block specifies that the deactivation of *C*1 leads to *C*2's failure as well as *C*3's activation. The figure also shows the XML-based representation of the DRBD model in RML. An RML file uses the opening `<rml>` tag to signify the beginning of a DRBD definition. Following it, an SRBD model is defined as the top structural component, called `MAIN` component. Component `MAIN` is defined as a serial component within the tags `<serialComponent>` and `</serialComponent>`, which may contain any number of structural components (simple or parallel ones). In this example, the only structural component contained in `MAIN` is a parallel component that is defined within the tags `<parallelComponent>` and `</parallelComponent>`. The parallel component has an identification of `PCom`, which consists of three simple components *C*1-*C*3. Each of them is defined within the tags `<simpleComponent>` and `</simpleComponent>`, and has an initial state defined inside the `<initialState>`… `</initialState>` tags. In this example, the parallel component consists of simple ones only, but in a more general case, it may contain serial components. Similarly, a serial component may also consist of any number of simple or parallel components.



```
<?xml version="1.0"?>
<rml>
  <serialComponent id = "MAIN">
    <parallelComponent id = "PCom">
      <simpleComponent id = "C1">
        <initialState>Active</initialState>
      </simpleComponent>
      <simpleComponent id = "C2">
        <initialState>Active</initialState>
      </simpleComponent>
      <simpleComponent id = "C3">
        <initialState>Standby</initialState>
      </simpleComponent>
    </parallelComponent>
  </serialComponent>
  <stateController id = "C1_SDEP">
    <triggerEvent>
      <id>C1</id>
      <event>Deactivation</trigger>
    </triggerEvent>
    <targetEvent>
      <id>C2</id>
      <event>Failure</event>
    </targetEvent>
    <targetEvent>
      <id>C3</id>
       <event>Activation</event>
    </targetEvent>
  </stateController>
</rml>
</xml>
```

Fig. 5. XML-based representation of a DRBD model in RML.

After an SRBD has been defined, controllers are to be added into the RML file using specific XML tags. For example, state controller `C1_SDEP` can be defined within the

`<stateController>` and `</stateController>` tags as shown in Fig. 5. Inside the `C1_SDEP` definition, the trigger and target events can be defined using `<triggerEvent>` … `</triggerEvent>` and `<targetEvent>` … `</targetEvent>` tags, respectively. Corresponding to (*D*, *F*) and (*D*, *A*) state-based dependency between component *C*1 and *C*2, and *C*1 and *C*3, respectively, we define the trigger event that occurs on *C*1 with a `Deactivation` event, and two target events, which occur on *C*2 and *C*3 with the events of `Failure` and `Activation`, respectively. When both SRBD model and controllers have been defined, the RML file is ended by the closing tag `</rml>`.

The motivation and major advantage of using RML to describe a DRBD model is to allow access and mutation of a DRBD model as an XML document. XML documents not only support a standard information encoding and storage format, but also allow programmers to use that information in a standard way [39]. Currently, two dominant APIs for processing XML-based documents are Simple API for XML (SAX) and Document Object Model (DOM). The SAX specification defines a low level API, which is an event-based approach that can parse through XML data and call handler functions when certain parts of the document are found. On the other hand, the DOM specification defines a tree-based approach to processing XML data. Based on the hierarchical structure of the XML data, the DOM approach creates an internal tree, which can be navigated at runtime. For efficiency reasons, in this project, we have adopted the DOM specification to process RML files.

## IV. Conversion of DRBD Models into CPN

In order to verify the correctness of a DRBD model, we need to convert it into CPN using a two-step procedure. First, the embedded SRBD of a DRBD model is converted into a CPN model. Then, the controller blocks are converted into Petri nets and added into the converted CPN model. The following sections give the detailed descriptions for the conversion procedures. Note that the CPN models described in the following sections employ CPN-ML, which is a powerful programming language of CPN as implemented in CPN Tools [40]. We assume readers have the basic knowledge of CPN-ML [41].

### *A. Conversion of SRBD into CPN*

Before we present the algorithm to convert the embedded SRBD of a DRBD model into a CPN model, we first describe how to convert each type of structural components in an SRBD into CPN. In order to model the component state, a colored token called a *state* token is introduced, which has three possible values, i.e., "Active", "Standby" and "Failed". The movement of these tokens in a CPN model signifies the state changes of the components in the DRBD model. Fig. 6 shows the conversion of a simple component into a CPN, called *simple-component CPN*.

A simple-component CPN contains two places, i.e., *C*1*_start* and *C*1*_up*. *C*1*_start* contains an initial token with color "Active" (denoted as `1`Active` in Fig. 6), indicating that its initial state is active. When *C*1 remains active and the other input place to transition *in_C*1 also contains an "Active" token (we do not show the other input place of transition *in_C*1 in Fig. 6, but it is connected to *in_C*1 through the *Input Connection* of the simple-component CPN), *in_C*1 may fire. Its firing deposits an "Active" token into *C*1*_up*, indicating that *C*1 is active. The "Active" token in *C*1*_up* can be further passed along to other modules through *Output Connection (Active)*. On the other hand, if transition *C*1*_destruct* fires while *C*1 is active, the "Active" token in *C*1*_start* is removed, and a "Failed" token is deposited into *C*1*_start*. In this case, transition *C*1*_fail* is enabled and can fire. When *C*1*_fail* fires, it generates a "true" token indicating that *C*1 fails. The generated "true" token can be further passed to other modules through *Output Connection (Failed)*.

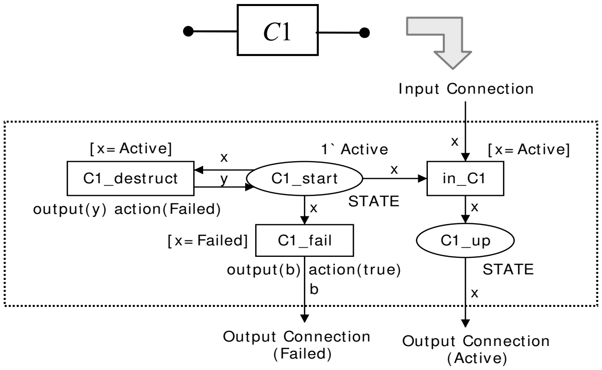


Fig. 6. Simple-component CPN for a simple component.

A *serial-component CPN* is a set of serially connected structural component CPN. Fig. 7 shows a serial component in DRBD containing two simple components, *C*1 and *C*2, and its CPN representation.

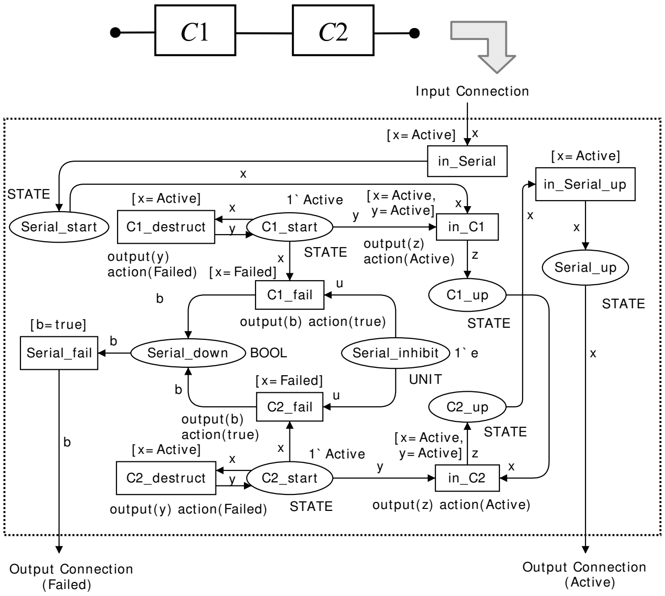


Fig. 7. Serial-component CPN for a serial component.

Similar to a simple-component CPN, a serial-component CPN has an interface that consists of an *Input Connection* (through its *in_Serial* transition) and two *Output Connections* (through its *Serial_up* place and *Serial_fail* transition). When transition *in_Serial* receives an "Active" token through *Input Connection*, it can fire, and its firing deposits an "Active" token into place *Serial_start*. This token enables transition *in_C*1 if place *C*1*_start* also contains an "Active" token.

The behavior of *C*1 in Fig. 7 is the same as that of the simple component *C*1 in Fig. 6. Note that both *C*1 and *C*2 in Fig. 7 are modeled in exactly the same way as *C*1 in Fig. 6. When both places *C*1*_up* and *C*2*_start* contain an "Active" token, transition *in_C*2 is enabled, and its firing deposits an "Active" token into *C*2*_up*. The "Active" token in *C*2*_up* further enables transition *in_Serial_up*, and may place an "Active" token in place *Serial_up*. Similar to a simple-component CPN, an "Active" token in *Serial_up* indicates that the serial component is functioning properly. The firing procedure also implies that the serial component is active only when both of its contained simple components, *C*1 and *C*2, are active.

On the other hand, when either *C*1 or *C*2 fails, transition *C*1*_fail* or *C*2*_fail* can fire. When either fires, a "true" token is deposited into place *Serial_down*, which enables transition *Serial_fail*. Firing *Serial_fail* generates a "true" token indicating that the serial component cannot function properly due to the failure of its contained components. The firing procedure also implies that the serial component becomes failed when either *C*1 or *C*2 fails. Note that when both *C*1 and *C*2 fail, only one of the transitions, either *C*1*_fail* or *C*2*_fail*, can fire because place *Serial_inhibit* limits the capacity of place *Serial_down* to one; thus, *Serial_fail* will not accidentally fire twice.

A parallel component contains a set of structural components (simple or serial components) that are connected in parallel. Fig. 8 shows the DRBD model of a parallel component with two simple components *C*1 and *C*2, and its CPN representation. Similar to a simple-component and a serial-component CPN, a *parallel-component CPN* has an *Input Connection* (through its *in_Para* transition) and two *Output Connections* (through its *Para_up* place and *Para_fail* transition).

Components *C*1 and *C*2 in Fig. 8 are modeled in the same way as shown in Fig. 6. When *Input Connection* passes an "Active" token to transition *in_Para*, its firing deposits an "Active" token into place *Para_start*, which enables both *in_C*1 and *in_C*2. When *C*1 or *C*2 is active, transition *in_C*1 or *in_C*2 may fire, and can deposit an "Active" token in place *C*1*_up* or *C*2*_up*, respectively. The "Active" token in either *C*1*_up* or *C*2*_up* enables *Para_C*1 or *Para_C*2, and eventually leads to an "Active" token in place *Para_up*. Similar to a serial-component CPN, an "Active" token in *Para_up* indicates that the parallel component can function properly. Note that at any time, only one of the transitions (either *in_C*1 or *in_C*2) may fire. Thus the capacity of place *Para_up* must be one.

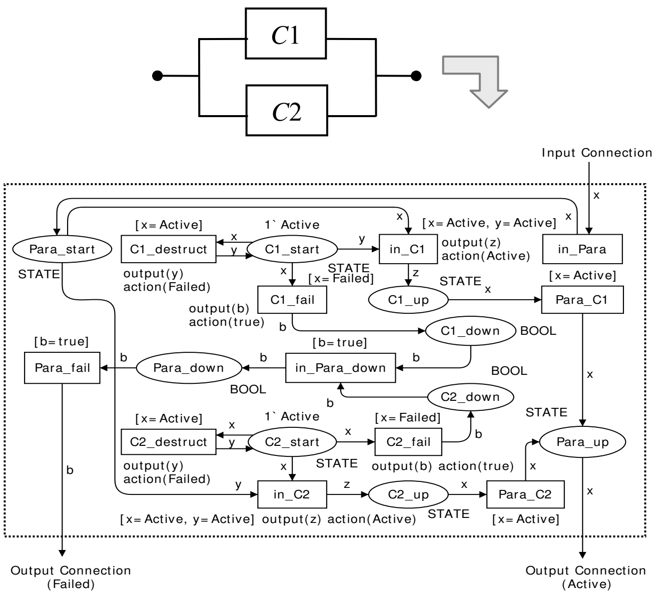


Fig. 8. Parallel-component CPN for a parallel component.

On the other hand, if both *C*1 and *C*2 fail, there will be a "true" token in both places *C*1*_down* and *C*2*_down*, which enables transition *in_Para_down*. Its firing deposits a "true" token into place *Para_down*, which enables transition *Para_fail*. Firing *Para_fail* generates and passes a "true" token to other modules through *Output Connection*. The firing procedure implies that the parallel component is not functioning due to the failure of both *C*1 and *C*2.

It is worth noting that although in the above examples, both serial and parallel components contain simple components only, they may contain serial or parallel components in a more general case. In such a case, CPN models can be composed in exactly the same way as described. This is because both a serial-component CPN and a parallel-component CPN have the same interface as a simple-component CPN. Thus, our conversion approach is compositional.

We now provide a recursive algorithm for automatically converting an SRBD model into a CPN model. The proposed recursive algorithm treats the previous techniques as a function that recursively expands structural components in order to derive a CPN that formally defines an entire SRBD. The algorithm is illustrated as pseudocode in Fig. 9, which is defined as a recursive function `convert_Serial_Component` with a parameter of type `SerialComponent`.

The algorithm starts with viewing a SRBD model as a serial component, and creating the needed input and output connections. As defined in Fig. 1, a serial component can contain one or more than one simple or parallel components. Thus we use a *for*-loop to convert each of the contained structural components. If a contained component is a simple or spare component, we convert it directly into a simple-component CPN as shown in Fig. 6; otherwise, if it is a parallel component, we first create the needed input and output connections for the parallel-component CPN, and then use a *for*-loop again to convert each of contained structural

```
function convert_Serial_Component(SerialComponent se_com)
   create input/output connections for se_com;
   foreach StructuralComponent s_com in se_com
      if (s_com is simpleComponent | spareComponent)
          convert s_com directly into a simple component CPN;
      else if (s_com is ParallelComponent)
          create input and output connections for s_com;
          foreach StructuralComponent p_com in s_com
             if (p_com is SimpleComponent | SpareComponent )
                convert p_com directly into a simple component CPN;
             else if (p_com is SerialComponent)
                convert_Serial_Component(p_com);
          end
          create all parallel connections in s_com;
   end
   create all serial connections in se_com;
end function
```

Fig. 9. Recursive algorithm for converting a SRBD into a CPN.

components into a CPN. For each contained structural component in the parallel component, we check whether it is a simple or spare component. If it is a simple or spare component, we convert it directly into a simple-component CPN; otherwise, if it is a serial one, the function `convert_Serial_Component` is called recursively. When all contained components in a parallel component have been converted into CPNs, all simple-component CPN and serial-component CPN are connected together (as shown in Fig. 8) to create a parallel-component CPN. Similarly, when all contained components in a serial component have been converted into CPNs, all simple-component and parallel-component CPNs are connected together (as shown in Fig. 7) to create a serial-component CPN.

The resulting CPN for an SRBD contains open input and output connections. In order to develop a complete CPN model for the SRBD, we introduce additional places and transitions into the SRBD CPN. As shown in Fig. 10, an SRBD is treated as serial component *MAIN* with three major places *MAIN_start*, *MAIN_up*, and *MAIN_down*.

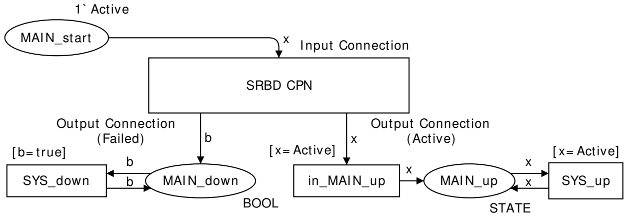


Fig. 10. The complete CPN model for an embedded SRBD.

Place *Main_start* initially contains an "Active" token, and connects to the SRBD CPN through *Input Connection*. Similarly, *Main_up* and *Main_down* connect to the SRBD CPN through *Output Connection (Active)* and *Output Connection (Failed)*, respectively. Note that since *Output Connection (Active)* can only pass a token to a transition, place *Main_up* connects to the SRBD CPN through an intermediate transition *in_Main_up*. In addition, two transitions, *SYS_up* and *SYS_down*, are connected to *MAIN_up* and *MAIN_down*, respectively. When there is a "true" token in either *MAIN_up* or *MAIN_down*, *SYS_up* or *SYS_down* can fire, which denotes that the system is functioning or failing. Note that when we execute the CPN model, it should eventually end up with firing of either *SYS_up* or *SYS_down*; otherwise, there must be a deadlock state existing in the CPN model.

### B. *Conversion of DRBD Controllers into CPN*

The next step in converting a DRBD model into a CPN is to convert DRBD controllers into controller CPN, and add them into the CPN model developed for the embedded SRBD model in step one. A controller CPN consists of a set of transitions and arcs that connect to the *start* places of the corresponding simple-component CPN. Fig. 11 and Fig. 12 illustrate the algorithms for converting a spare controller into a spare-controller CPN and converting a state controller into a state-controller CPN, respectively. Note that in the algorithm presented in Fig. 12, when the trigger event is *deactivation*, no synchronization place needs to be introduced. We now use the spare controller and state controller examples in Fig. 4 to illustrate how these algorithms work. Fig. 13 shows a *spare-controller CPN* for the SPARE controller block in Fig. 4 (a). The SPARE controller block models the spare part relationship between primary component *P*1 and two cold spare parts *S*1 and *S*2. When *P*1 fails, *S*1 is activated, and similarly, when *S*1 fails, *S*2 is activated. In order to model such a cascading relationship in CPN, we introduce two transitions *SPC_P*1 and *SPC_S*1, which connect the start places of *P*1 and *S*1, to the start places of *S*1 and *S*2, respectively. When *P*1 fails, and *S*1 is in its standby state, transition *SPC_P*1 may fire, which removes the "Standby" token in place *S*1*_start*, and deposits an "Active" token into *S*1*_start.* This indicates that *S*1 changes its state from "Standby" to "Active" due to the failure of *P*1. Similarly, when *S*1 fails, transition *SPC_S*1 may fire, which changes the state of *S*2 from "Standby" to "Active". Note that in the spare-controller CPN model in Fig. 13, there are two

```
function convert_Spare_Controller(SpareController sp_con)
   create place P1_start and transition P1_fail for primary component P1;
   foreach SpareComponent Si (i = 1 to n) in sp_con
      create place Si_start and transition Si_fail;
   end
   create transition SPC_P1 that connects P1_start and S1_start such that
   when P1 fails and S1 is standby, S1 is activated;
   foreach SpareComponent Si (i = 1 to n-1) in sp_con
      create transition SPC_Si that connects Si_start and S(i+1)_start
      such that when Si fails and S(i+1) is standby, S(i+1) is activated;
   end
   create place SPC_sync1 that connects transitions SPC_P1 and P1_fail;
   foreach SpareComponent Si (i = 1 to n-1) in sp_con
      create place SPC_sync(i+1) that connects transitions SPC_Si and
      Si_fail;
   end
end function
```

Fig. 11. Algorithm for converting a spare controller into a spare-controller CPN.

```
function convert_State_Controller(StateController st_con)
   create place C1_start for trigger component C1;
   foreach TargetComponent Ci (i = 2 to n) in st_con
      create place Si_start for Ci;
   end
   create transition SDEP that connects all places Ci_start (i = 1 to n)
   according to the trigger and target events defined in st_con;
   if (trigger event is activation)
      create transition in_C1 for trigger component C1;
      create place SDEP_sync that connects transitions SDEP and in_C1;
   else if (trigger event is failure)
      create transition C1_fail for trigger component C1;
      create place SDEP_sync that connects transitions SDEP and C1_fail;
end function
```

Fig. 12. Algorithm for converting a state controller into a state-controller CPN.

synchronization places: *SPC_sync*1 and *SPC_sync*2. When transition *SPC_P*1 (*SPC_S*1) fires, a unit token is deposited into place *SPC_sync*1 (*SPC_sync*2), which enables transition *P*1*_fail* (*S*1*_fail*). Thus, *SPC_sync*1 (*SPC_sync*2) can be used to ensure that the firing of transition *SPC_P*1 *(SPC_S*1) precedes that of transition *P*1*_fail* (*S*1*_fail*), and the "Failed" token in place *P*1*_start* (*S*1*_start*) will not be accidentally removed before transition *SPC_P*1 (*SPC_P*2) fires.

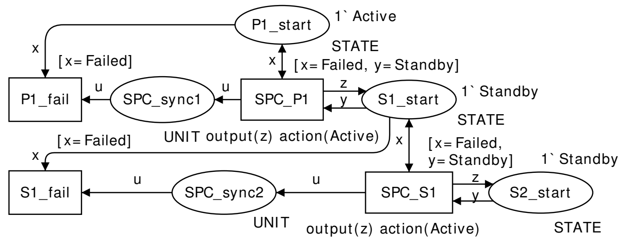


Fig. 13. Spare-controller CPN for the SPARE block in Fig. 4 (a).

In a DRBD model, a state controller (i.e., an SDEP controller block) models a state-based dependency relationship between simple components. Fig. 14 shows a *state-controller CPN* for the SDEP controller block with a trigger component *C*1 and two target components *C*2 and *C*3 defined in Fig. 4 (b). The SDEP block is modeled by an *SDEP* transition in the state-controller CPN, which connects the *start* places of the three components. When *C*1 becomes active, and both *C*2 and *C*3 are also active, transition *SDEP* becomes enabled. Its firing deposits a "Standby" and "Failed" token into places *C*2*_start* and *C*3*_start*, respectively. It also deposits a unit token into synchronization place *SDEP_sync*, which may enable transition *in_C*1 when *C*1*_start* contains an "Active" token. Thus, *SDEP_sync* ensures that the firing of *SDEP* precedes that of *in_C*1, and the "Active" token in place *C*1*_start* will not be accidentally removed before *SDEP* fires.

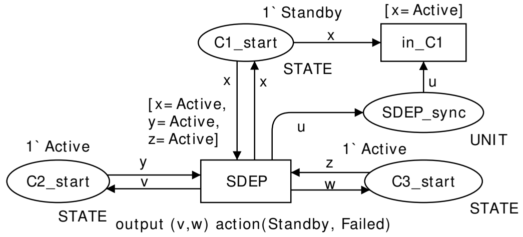


Fig. 14. State-controller CPN for the SDEP block in Fig. 4 (b).

Note that if the trigger event from simple component *C*1 is *failure* instead of *activation*, synchronization place *SDEP_sync* should be connected to transition *C*1*_fail* instead of *in_C*1. This case is illustrated in Fig. 15. On the other hand, if the trigger event from *C*1 is *deactivation*, no synchronization place is needed. This is because when *C*1 becomes standby, neither of *C*1*_fail* and *in_C*1 is enabled, and *SDEP* is the only one enabled due to a "Standby" token in place *C*1*_start*.

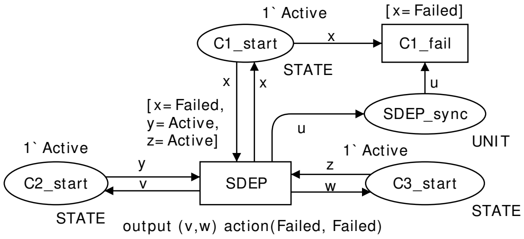


Fig. 15. State-controller CPN with *failure* trigger event.

Finally, the converted controller CPN models can be added into the CPN model developed for the embedded SRBD model in step one. This procedure can be done by merging the *start* places (e.g., *P*1*_start* in Fig. 13) and *status* transitions (e.g., *P*1*_fail* in Fig. 13 and *in_C*1 in Fig. 14) from the controller CPN models with the corresponding places and transitions defined in the CPN model for the embedded SRBD model. We illustrate this process in a case study presented in the following section.

## V. Case Study: Conversion of DRBD into CPN for Formal Verification

### A. DRBD Model of a Redundant Generator

Consider a coast guard vessel whose electrical system is powered by three generators: primary, backup, and secondary backup one used only for emergency. The primary and backup generators can provide the vessel with enough kilowatts (KW) output to power all electrical components and equipment; while the emergency generator has less wattage output and can supply only power to the vessel's essential equipment such as navigational lights, emergency lights and other equipment that keeps the engine running. Initially, only the primary generator is running, and the other two generators are in standby states. At runtime, if the primary one fails, it automatically triggers the backup one to switch from standby to online. Similarly, if the backup one fails, the emergency generator is activated. Connected in series to the generators is a power bus that is a series of circuit breakers that feed electricity from a generator to the electrical components on the ship. The power bus in this system contains two parallel buses, namely main and emergency buses. The main bus contains the breakers for all of the ship's components, while the emergency bus powers only the vessel's essential equipment.

Fig. 16 shows the DRBD model for the system described above. It consists of two parallel components that are connected in serial. The first parallel component contains the generator components and is composed of the primary generator (*PG*1), backup generator (*BG*1) and emergency generator (*BG*2). *PG*1 is a simple component, initially in an "Active" state; while *BG*1 and *BG*2 are cold spare components, which are initially in "Standby" states. A spare controller (*SPARE*) is introduced to model the cascading failure of *PG*1 and *BG*1. If *PG*1 fails, *BG*1 is activated, and upon failure of *BG*1, *BG*2 enters its "Active" state. The second parallel component models the power buses. The two power buses, main bus (*MB*) and emergency bus (*EB*), are represented in the DRBD model as simple components within the power bus parallel component. Since the emergency generator *BG*2 does not output enough wattage to power *MB* when it enters its "Active" state, *MB* must be deactivated and *EB* must enter its "Active" state. This state-based dependency between *BG*2 and the power buses is modeled by an *SDEP* state controller.

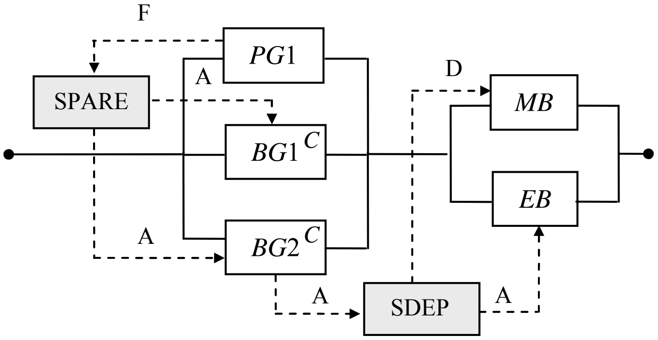


Fig. 16. DRBD model of a redundant generator system.

### B. Automatic Generation of a CPN Model

According to the algorithm presented in Section IV, the DRBD model of the redundant generator system can be converted into a CPN model as shown in Fig. 17. The first structural component within *MAIN* serial component is a parallel component representing the set of generators, denoted as *GEN*. During the conversion of *GEN* into CPN, CPN models corresponding to each generator (*PG*1, *BG*1, or *BG*2) are first created and then connected in parallel according to the algorithm. These parallel connections are illustrated in Fig. 17, where each generator CPN initially contains an "Active", "Standby", and "Standby" token in their start places *PG*1*_start*, *BG*1*_start*, and *BG*2*_start*, respectively. When any of these components is active, there is an "Active" token in one of places *PG*1*_up*, *BG*1*_up*, and *BG*2*_up*, which enables transitions *GEN_PG*1, *GEN_BG*1, and *GEN_BG*2, respectively. When one of these transitions fires, an "Active" token is deposited into place *GEN_up*, indicating that the *GEN* parallel component is active. On the other hand, if all of the places *PG1_down*, *BG1_down*, and *BG2_down* contain a "true" token, transition *in_GEN_down* may fire, which deposits a "true" token into place *GEN_down*. This enables transition *GEN_fail*, and its firing generates a "true" token indicating that the *GEN* structural

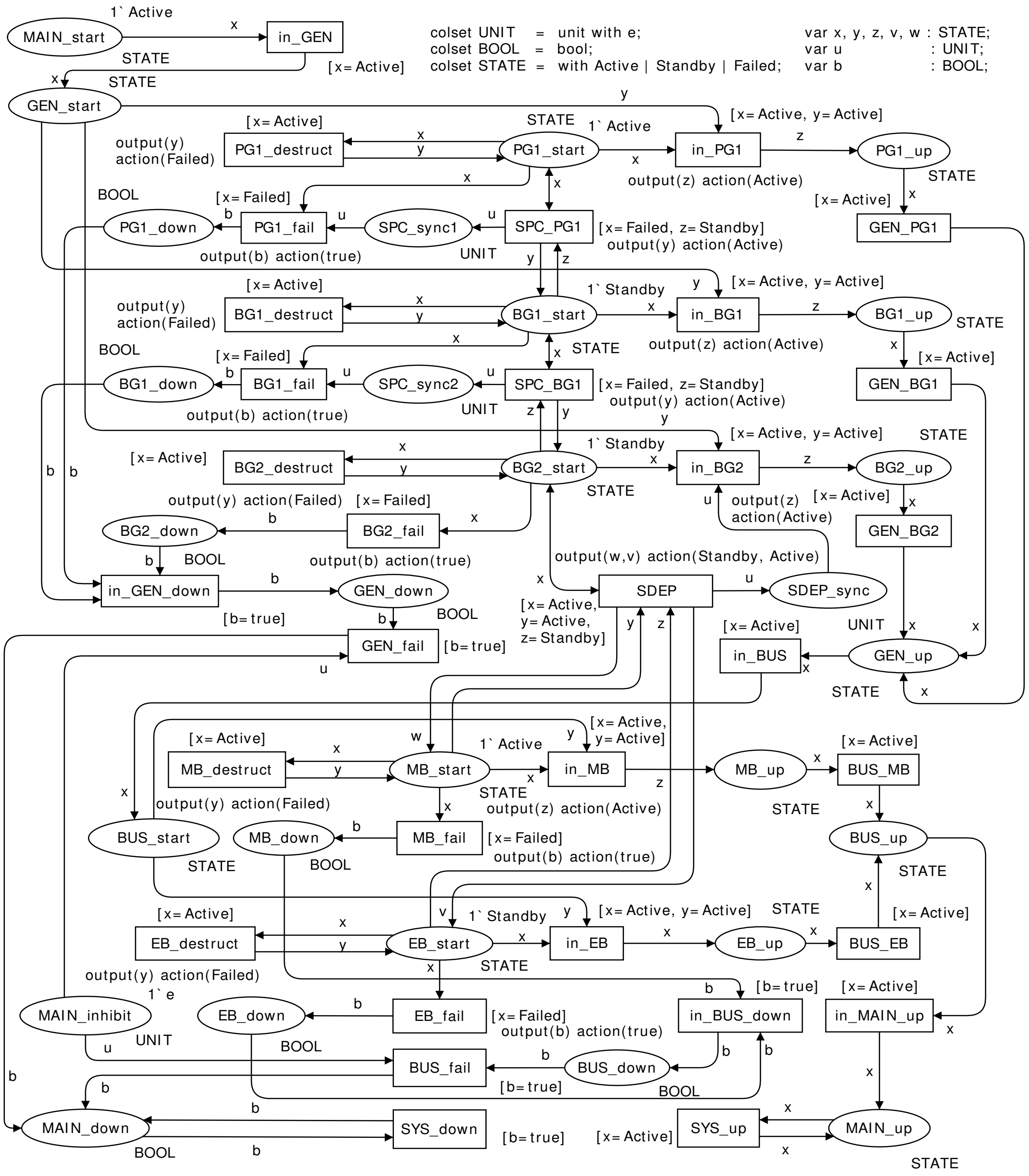


Fig. 17. CPN model converted from the DRBD model in Fig. 16.

component is not functioning.

The second structural component contained in *MAIN* is parallel component *BUS* representing the parallel power bus circuit in the DRBD model shown in Fig. 16. The conversion of *BUS* into CPN follows the same procedure as for parallel component *GEN*. When either of the buses is active, an “Active” token is deposited into place *BUS_up*, indicating that *BUS* is active. On the other hand, when both buses fail (indicated by a “true” token in both places *MB_down* and *EB_down*), transition *in_BUS_down* may fire, and its firing leads to a “true” token in place *BUS_down*. When the bus is down, transition *BUS_fail* may fire, and its firing generates a “true” token that can be passed to place *MAIN_down*.

Once *GEN* and *BUS* are converted into their corresponding CPN models, they can be connected serially within component *MAIN*. The serial connection between the two structural components is simply made by connecting place *GEN_up* from parallel-component CPN of *GEN* to transition *in_BUS* from parallel-component CPN of *BUS*. In addition, since *GEN* is the first serially connected structural component, its transition *in_GEN* is connected to place *MAIN_start*. Similarly, since *BUS* is the last serial component, its place *BUS_up* is connected to transition *in_MAIN_up*. On the other hand, both transitions *GEN_fail* and *BUS_fail* are connected to place *MAIN_down*. However, due to inhibitor place *MAIN_inhibit*, only one of the transitions may fire, which ensures that the capacity of *MAIN_down* is one.

In step two of the conversion, the DRBD controllers are converted into CPN and added into the CPN model developed in step one. In this example, we have two controllers, i.e., *SPARE* and *SDEP* controller block. The *SPARE* controller block models the redundant behaviors of the three generators (*PG*1, *BG*1, and *BG*2) and is converted into two transitions *SPC_PG*1 and *SPC_BG*1 in the spare-controller CPN. The transition *SPC_PG*1 connects *PG*1*_start* and *BG*1*_start*, which is responsible for activating the backup generator *BG*1 when primary generator *PG*1 fails. Similarly, transition *SPC_BG*1 connects *BG*1*_start* and *BG*2*_start*, which is responsible for activating emergency generator *BG*2 when backup generator *BG*1 fails. Note that synchronization place *SPC_sync*1 is used to ensure that a "Failed" token in place *PG*1*_start* (*BG*1*_start*) is not removed before transition *SPC_PG*1 (*SPC_BG*1) fires. The state controller block *SDEP* in Fig. 16, which deactivates main power bus *MB* and activates emergency power bus *EB* when *BG*2 is activated, is converted into transition *SDEP* in the state-controller CPN. The *SDEP* transition connects the three start places of components *BG*2, *MB* and *EB*, and its firing deposits a "Standby" token and an "Active" token into places *MB_start* and *EB_start*, respectively. *SDEP_sync* is used to ensure that the "Active" token is not accidentally removed before transition *SDEP* fires.

In order to illustrate automatic generation of a CPN model from a DRBD model, we have implemented a prototype application that transforms an input file of DRBD model in RML into an output file of CPN model in XML that can be recognized by CPN Tools [40]. The prototype was implemented in Java 5, which provides a simple interface that can load an RML file and output a converted CPN model in XML. We use Document Object Model (DOM) technology [42] to parse an input RML file into a tree representing the corresponding DRBD structure for efficient processing and conversion. For details about the implementation of the prototype, refer to [43].

### C. Analysis of DRBD Model Using CPN Tools

Design errors in a DRBD model can be discovered by analyzing the state space of the CPN model converted from the DRBD model. Using an existing Petri net tool, called CPN Tools [40], we can generate a report detailing the properties of the CPN model in Fig. 17. The report, shown as the analysis results in Table I, indicates that the full state space (or called the occurrence graph) can be generated from the CPN model in zero second (almost instantaneously), which consists of 288 nodes and 763 arcs. Similarly, the CPN Tools can be used to further generate a strongly connected components (Scc) graph from the occurrence graph. The generated Scc graph consists of 288 nodes and 744 arcs, and plays an important role for analysis. The report also indicates that there are three deadlock states in the CPN model, namely $S_{78}$, $S_{171}$, and $S_{282}$. They imply that transition *SYS_up* or *SYS_down* of the CPN model cannot eventually fire; therefore, there must be some design errors in the DRBD model. By tracing these deadlocks using CPN Tools, we find the following firing sequences that lead to them.

$\sigma_1$ = <$S_1$, *in_GEN*, $\mathrm{S}_4$, *in_PG1*, $\mathrm{S}_{10}$, *GEN_PG1*, $S_{19}$, *MB_destruct*, $S_{30}$, *in_BUS*, $S_{49}$, *MB_fail*, $S_{78}$>

$\sigma_2$ = <$S_1$, *PG1_destruct*, $S_2$, *in_GEN*, $S_6$, *SPC_PG1*, $S_{14}$, *in_BG1*, $S_{26}$, *GEN_BG1*, $S_{43}$, *MB_destruct*, $S_{55}$, *PG1_fail*, $S_{87}$, *MB_fail*, $S_{128}$, *in_BUS*, $S_{171}$>

$\sigma_3$ = <$S_1$, *PG1_destruct*, $S_2$, *SPC_PG1*, $S_7$, *BG1_destruct*, $S_{15}$, *SPC_BG1*, $S_{28}$, *SDEP*, $S_{46}$, *EB_destruct*, $S_{74}$, *EB_fail*, $S_{114}$, *in_GEN*, $S_{143}$, *in_BG2*, $S_{184}$, *PG1_fail*, $S_{222}$, *BG1_fail*, $S_{251}$, *GEN_BG2*, $S_{271}$, *in_BUS*, $S_{282}$>

TABLE I
ANALYSIS RESULTS OF THE CPN MODEL IN FIG. 14

| Statistics | Liveness Properties |
|---|---|
| `State Space` | `Dead Markings` |
| `  Nodes:  288` | `  [78,171,282]` |
| `  Arcs:   763` | `Dead Transition Instances` |
| `  Secs:   0` | `  Generator'BUS_fail 1` |
| `  Status: Full` | `  Generator'in_BUS_down 1` |
| `Scc Graph` | `Live Transition Instances` |
| `  Nodes:  288` | `  None` |
| `  Arcs:   744` | |
| `  Secs:   0` | |

From firing sequence $\sigma_1$, it is easy to see that $S_{78}$ is due to the failure of main bus *MB* when the primary generator *PG*1 is functioning. Although emergency bus *EB* is in the "Standby" state, and can provide services if activated, no such spare part relationship between *MB* and *EB* exists in either the DRBD model or corresponding CPN model. The firing sequence $\sigma_2$ shows the similar situation when *PG*1 fails, and backup generator *BG*1 is active, but *MB* fails and *EB* is still in a "Standby" state. The firing sequence $\sigma_3$ illustrates a different scenario. When both *PG*1 and *BG*1 fail, and emergency generator *BG*2 is activated, *MB* and *EB* will be deactivated and activated, respectively, due to the *SDEP* relationship between *BG*2 and bus components *MB* and *EB*. However, at this point of time, when *EB* fails, the *BUS* parallel component cannot be considered as "failed" because *MB* is still in a "Standby" state. Therefore, in the parallel-component CPN of *BUS*, neither place *BUS_up* will receive an "Active" token nor transition *BUS_fail* can fire. This leads to another deadlock situation in the CPN because no token will be deposited into either of places *MAIN_down* and *MAIN_up*. As a consequence, none of transitions *SYS_down* and *SYS_up* can fire eventually.

In order to correct the design errors in the DRBD model, we need to define *EB* as a spare part of *MB* by introducing a *SPARE* block that links *MB* and *EB*, and labeling the links from *MB* to *SPARE*, and *SPARE* to *EB* by *D* | *F* and *A*, respectively. This implies that when *MB* is deactivated or failed, *EB* is automatically activated. As a result, in Fig. 16, the link from *SDEP* to *EB* labeled by *A* is no longer needed, and can be deleted. Now based on the revised version of the DRBD model, we fix the CPN model in Fig. 17 as follows.

1. Add transition *SPC_MB* with places *MB_start* and *EB_start* as both of its input and output places.
2. Add synchronization place *SPC_sync3* with *SPC_MB* as

its input transition and *MB_fail* as its output transition.

3. Set the guard of transition *SPC_MB* such that *MB_start* contains a “Failed” or “Standby” token and *EB_start* contains a “Standby” token, i.e., `[x=Failed orelse x=Standby, y=Standby];`
4. Set the output of transition *SPC_MB* to deposit an “Active” token into place *EB_start* when *SPC_MB* fires, i.e., `output(z); action(Active).`
5. Modify the guard of transition *MB_fail* from `[x=Failed]` to `[x=Failed orelse x=Standby]`. This is because *MB* is deactivated only when both *PG*1 and *BG*1 are failed. In this case, *MB* should not be activated, and thus, should be considered as failed.
6. Delete the arcs between transition *SDEP* and place *EB_start*.

We now use the CPN Tools again to analyze the revised CPN model, and get the analysis results as shown in Table II. The results show that the revised CPN model has no deadlocks, which guarantees the correctness of the revised DRBD model in terms of deadlock-freeness. Further properties of the DRBD model can be analyzed using model checking techniques as demonstrated in previous work [5]. Refer to [44] for more examples of system properties that can be formally verified using existing Petri net tools.

TABLE II
ANALYSIS RESULTS OF THE CPN MODEL IN FIG. 14 (AFTER REVISION)

| Statistics | Liveness Properties |
|---|---|
| `State Space` | `Dead Markings` |
| `  Nodes:  897` | `  None` |
| `  Arcs:   2836` | `Dead Transition Instances` |
| `  Secs:   1` | `  None` |
| `  Status: Full` | `Live Transition Instances` |
| `Scc Graph` | `  None` |
| `  Nodes:  897` | |
| `  Arcs:   2700` | |
| `  Secs:   0` | |

It is worth noting that the correct CPN model we developed for the redundant generator system can be further used for analysis and evaluation of system reliability properties. Such analysis and evaluation techniques are demonstrated in [26-27]. The detailed description of reliability evaluation on the CPN model is beyond the scope of this paper, but will be presented in our future work.

## VI. CONCLUSIONS AND FUTURE WORK

There is a growing demand to build reliable and stable computer systems. Building these types of systems involves creating an accurate and correct system reliability model. A reliability model ensures that the constructed system has the desired measures of reliability determined by the system designers. This paper presents a procedure for formal modeling and verifying dynamic reliability block diagram (DRBD) for computer-based systems. In the procedure, a DRBD model is first converted into CPN. Then, existing CPN tools are used to verify the behavioral properties of the DRBD model, where design flaws and faulty states of the DRBD model can be identified by tracing the deadlock states of the CPN model. Our case study shows that the proposed approach supports effective detection and tracing of subtle design errors in a DRBD model, and can provide a potential solution to automated verification of DRBD models. For future work, we plan to investigate automated verification approaches for safety critical system analysis. We will consider to use compositional time Petri nets [45] to model time sensitive dependency between components in a system. We will also study how to analyze a DRBD model for system reliability evaluation, and develop a comprehensive development environment that supports editing, verification, analysis and evaluation of DRBD models for complex and large computer-based systems.

## ACKNOWLEDGMENT

The authors would like to thank the Associate Editor and all anonymous referees whose comments and suggestions greatly helped us to improve the presentation and the quality of this paper.

## REFERENCES

[1] S. M. Rinaldi, “Modeling and simulating critical infrastructures and their interdependencies,” in *Proc. 37th Annual Hawaii Int. Conf. System Sciences (HICSS'04)*, Jan. 2004, Big Island, HI, USA, pp. 20054a (1-8).

[2] M. Rausand and A. Hoyland, *System Reliability Theory: Models, Statistical Methods, and Applications*, 2nd Edition, New York, USA, Wiley-Interscience, 2003.

[3] R. Manian, J. Dugan, D. Coppit, and K. Sullivan, “Combining various solution techniques for dynamic fault tree analysis of computer systems,” in *Proc. 3rd Int. Symp. High-Assurance Systems Engineering (HASE’98)*, Washington, D.C., USA, 1998, pp. 21–28.

[4] H. Xu and L. Xing, “Formal semantics and verification of dynamic reliability block diagrams for system reliability modeling,” in *Proc. 11th Int. Conf. Software Engineering and Applications*, Nov. 2007, Cambridge, Massachusetts, USA, pp. 155–162.

[5] H. Xu, L. Xing, and R. Robidoux, “DRBD: dynamic reliability block diagrams for system reliability modeling,” *International Journal of Computers and Applications (IJCA)*, vol. 31, no. 2, pp. 132-141, 2009.

[6] Z. Hu and S. M. Shatz, “Explicit modeling of semantics associated with composite states in UML statecharts,” *Journal of Automated Software Engineering*, vol. 13, no. 4, Oct. 2006, pp. 423-467.

[7] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*, NUREG-0492, U.S. Government Printing Office, Washington, DC, USA, 1981.

[8] H. Boudali, P. Crouzen, and M. Stoelinga, “Dynamic fault tree analysis using input/output interactive Markov chains,” in *Proc. 37th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN’07)*, June 2007, Edinburgh, UK, pp. 708-717.

[9] A. Abd-Allah, “Extending reliability block diagrams to software architecture,” *Technical Report USC-CSE-97-501*, University of Southern California, Mar. 1997.

[10] R. A. Sahner and K. S. Trivedi, “Reliability modeling using SHARPE,” *IEEE Trans. Reliab.*, vol. R-36, no. 2, pp. 186-193, Jun. 1987.

[11] C. Leangsuksun, H. Song, and L. Shen, “Reliability modeling using UML,” In *Proc. 2003 Int. Conf. Software Engineering Research and Practice*, Jun. 2003, Las Vegas, Nevada, USA, pp. 259-262.

[12] H. Dammag and N. Nissanke, “Safecharts for specifying and designing safety critical systems,” in *Proc. 18th IEEE Symp. Reliable Distributed Systems*, 1999, Lausanne, Switzerland, pp. 78-87.

[13] P.-A. Hsiung; S.-W. Lin; C.-H. Tseng; T.-Y. Lee; J.-M. Fu; W.-B. See, “VERTAF: an application framework for the design and verification of

embedded real-time software," *IEEE Trans. Softw. Eng.*, vol. 30, no. 10, pp. 656-674, Oct. 2004.

[14] J. T. Blake, A. L. Reibman, and K. S. Trivedi, "Sensitivity analysis of reliability and performability measures for multiprocessor systems," in *Proc. 1988 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, 1988, pp. 177-186.

[15] R. Duke, G. Rose, and G. Smith, "Object-Z: a specification language advocated for the description of standards," *Computer Standards and Interfaces*, vol. 17, North-Holland, 1995, pp. 511-533.

[16] T. Murata, "Petri nets: properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541-580, Apr. 1989.

[17] J. Wang, *Timed Petri Nets: Theory and Application*, Norwell, MA, Kluwer Academic Publishers, 1998.

[18] M. C. Zhou and K. Venkatesh, *Modeling, Simulation and Control of Flexible Manufacturing Systems: A Petri Net Approach*, Singapore, World Scientific, 1999.

[19] F.-Y. Wang, K. J. Kyriakopoulos, A. Tsolkas, and G. N. Saridis, "A Petri-net coordination model for an intelligent mobile robot," *IEEE Trans. Syst., Man and Cybern.*, vol. 21, no. 4, pp. 777-789, Jul.-Aug. 1991.

[20] M. Jeng, X. Xie, and S.-L. Chung, "ERCN* merged nets for modeling degraded behavior and parallel processes in semiconductor manufacturing systems," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 34, no. 1, pp. 102-112, Jan. 2004.

[21] M. P. Fanti and M. C. Zhou, "Deadlock control methods in automated manufacturing systems," *IEEE Trans. Syst., Man, and Cybern. A, Syst., Humans*, vol. 34, no. 1, pp. 5-22. Jan. 2004.

[22] Z. W. Li, H. S. Hu, and A. R. Wang, "Design of liveness-enforcing supervisors for flexible manufacturing systems using Petri nets," *IEEE Trans. Syst., Man, and Cybern. C, Appl. Rev.*, vol. 37, no. 4, pp. 517-526, Jul. 2007.

[23] F.-S. Hsieh, "Fault-tolerant deadlock avoidance algorithm for assembly processes," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 34, no. 1, pp. 65-79, Jan. 2004.

[24] N. Wu and M. C. Zhou, "Modeling and deadlock avoidance of automated manufacturing systems with multiple automated guided vehicles," *IEEE Trans. Syst., Man, and Cybern. B, Cybern.*, vol. 35, no. 6, pp. 1193-1202, Dec. 2005.

[25] Z. W. Li, M. C. Zhou, and N. Q. Wu, "A survey and comparison of Petri net-based deadlock prevention policy for flexible manufacturing systems," *IEEE Trans. Syst., Man, and Cybern. C, Appl. Rev.*, vol.38, no.2, pp. 172-188, 2008.

[26] A. Bobbio, G. Franceschinis, L. Portinale, and R. Gaeta, "Exploiting Petri nets to support fault-tree based dependability analysis," in *Proc. 8th Int. Workshop on Petri Nets and Performance Models*, Zaragoza, Spain, Sept. 1999, pp. 146-155.

[27] M. Everdij and H. Blom, "Petri-nets and hybrid-state Markov processes in a power-hierarchy of dependability models," in *Proc. IFAC Conf. Analysis and Design of Hybrid Systems*, June 2003, Saint-Malo, Brittany, France.

[28] N. G. Leveson and J. L. Stolzy, "Safety analysis using Petri nets," *IEEE Trans. Softw. Eng.*, vol. 13, no. 3, pp. 386-397, Mar. 1987.

[29] U. Buy and R. Sloan. "A Petri net-based approach to real-time program analysis," in *Proc. 7th Int. Workshop on Software Specification and Design*, Dec. 1993, Redondo Beach, California, pp. 56-60.

[30] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzc, "A unified high-level Petri net formalism for time-critical systems," *IEEE Trans. Softw. Eng.*, vol. 17, no. 2, pp. 160-172, Feb. 1991.

[31] K. Jensen, "Coloured Petri nets: basic concepts, analysis methods, and practical use," *Basic Concepts EATCS Monographs on Theoretical Computer Science*, vol. 2, Springer-Verlag, 1997.

[32] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125-143, Mar. 1977.

[33] Y. Y. Du, C. J. Jiang, and M. C. Zhou, "Modeling and analysis of real-time cooperative systems using Petri nets," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 37, no. 5, pp. 643-654, Sept. 2007.

[34] V. R. L. Shen and T. T.-Y. Juang, "Verification of knowledge-based systems using predicate/transition nets," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 38, no. 1, pp. 78-87, Jan. 2008.

[35] L. Ma and J. J. P. Tsai, "Formal modeling and analysis of a secure mobile-agent system," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 38, no. 1, pp. 180-196, Jan. 2008.

[36] J.-S. Lee, M. C. Zhou, and P.-L. Hsu, "Multiparadigm modeling for hybrid dynamic systems using a Petri net framework," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 38, no. 2, pp. 493-498, Mar. 2008.

[37] H. Wang and Q. Zeng, "Modeling and analysis for workflow constrained by resources and nondetermined time: an approach based on Petri nets," *IEEE Trans. Syst., Man and Cybern. A, Syst., Humans*, vol. 38, no. 4, pp. 802-817, Jul. 2008.

[38] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Boston, MA, Addison-Wesley Longman Publishing Co. Inc., 1989.

[39] C. Goldfarb and P. Prescod, *The XML Handbook*, Upper Saddle River, NJ, Prentice Hall, 2000.

[40] A. V. Ratzer, L. Wells, H. M. Lasen, M. Laursen, J. F. Qvortrup, *et al.*, "CPN Tools for editing, simulating and analyzing colored Petri nets," in *Proc. 24th Int. Conf. Application and Theory of Petri Nets*, Eindhoven, Netherlands, Jun. 2003, pp. 450-462.

[41] CPN Group, "CPN ML: language for declarations and net inscriptions," *CPN Tools Help*, Department of Computer Science, University of Aarhus, Jul. 2008, Retrieved on Aug. 26, 2008, from http://wiki.daimi.au.dk/cpntools-help/cpn_ml.wiki

[42] E. R. Harold, *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*, Boston, MA, Addison-Wesley Professional, 2002.

[43] R. Robidoux, "Automated verification of a computer system reliability model," *M.S. Thesis*, Computer and Information Science Department, University of Massachusetts Dartmouth, Jul. 2007.

[44] H. Xu and S. M. Shatz, "A framework for model-based design of agent-oriented software," *IEEE Trans. Softw. Eng.*, vol. 29, no. 1, pp. 15-30, Jan. 2003.

[45] J. Wang, Y. Deng, and M. Zhou, "Compositional time Petri nets and reduction rules," *IEEE Trans. Syst., Man and Cybern. B, Cybern.*, vol. 30, no. 4, pp. 562-572, Aug. 2000.

**Ryan Robidoux** received the B.S. degree and the M.S. degree in Computer Science from University of Massachusetts Dartmouth, MA, in 2004 and 2007, respectively.

He is currently a Research Associate and a Software Developer at the Kaput Center for Research and Innovation in Mathematics Education, and an adjunct faculty member in the Computer and Information Science Department at University of Massachusetts Dartmouth. His major research interests include neural network, software engineering, formal methods, and web services.

Ryan is a recipient of the Faculty Award for top students in the Computer and Information Science Department, University of Massachusetts Dartmouth, 2004.

**Haiping Xu** (S'97–M'03–SM'07) received the B.S. degree in Electrical Engineering from Zhejiang University, Hangzhou, China, in 1989, the M.S. degree in Computer Science from Wright State University, Dayton, OH, in 1998, and the Ph.D. degree in Computer Science from the University of Illinois at Chicago, IL, in 2003.

Prior to 1996, he successively worked with Shen-Yan Systems Technology, Inc. and Hewlett-Packard Co., as a Software Engineer, in Beijing, China. Since 2003, he has been with the University of Massachusetts Dartmouth, where he is currently an Associate Professor at the Computer and Information Science Department, and a Co-Director of the Concurrent Software Engineering Laboratory (CSEL). He has published over 40 research papers including 20 peer-reviewed journal publications. He has supervised about 30 M.S. theses and M.S. projects at University of Massachusetts Dartmouth, and co-supervised 2 Ph.D. dissertations. His research has been supported by the U.S. National Science Foundation (NSF) and the U.S. Marine Corps. His research interests include distributed software engineering, formal methods, cyber security, multi-agent systems, electronic commerce, trustworthy computing, reliability engineering and service-oriented systems.

Dr. Xu is a senior member of the Association of Computing Machinery (ACM). He is currently an Associate Editor for several journals including the

Journal of Computers (JCP) and International Journal of Computers and Applications (IJCA). He has served as a program committee Co-Chair for the International Conference on Software Engineering Theory and Practice (SETP), and a program committee member for over 30 international conferences. He is a recipient of the Outstanding Ph.D. Thesis Award in 2004, and has been included in the 11th Edition of *Who's Who Among America's Teachers*, 2006.

**Liudong Xing** (S'00–M'02–SM'07) received the B.E. degree in Computer Science from Zhengzhou University, Zhengzhou, China, in 1996 and the M.S. and Ph.D. degrees in Electrical Engineering from the University of Virginia, Charlottesville, in 2000 and 2002, respectively.

She was a Research Assistant with the Chinese Academy of Sciences, Shenyang, from 1996 to 1998. She joined University of Massachusetts Dartmouth, North Dartmouth, in 2002, where she is currently an Associate Professor with the Electrical and Computer Engineering Department. Her major field of study is on reliability engineering and fault-tolerant computing. Her current research interests include dependable computing and networking, hardware and software reliability engineering, fault-tolerant computing, and wireless sensor networks.

**MengChu Zhou** (S'88–M'90–SM'93–F'03) received the B.S. degree in Electrical Engineering from Nanjing University of Science and Technology, Nanjing, China, in 1983, the M.S. degree in Automatic Control from the Beijing Institute of Technology, Beijing, China, in 1986, and the Ph.D. degree in Computer and Systems Engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1990.

He joined New Jersey Institute of Technology (NJIT), Newark, in 1990, where he is currently a Professor of electrical and computer engineering in the Department of Electrical and Computer Engineering and the Director of Discrete-Event Systems Laboratory. He also serves as the Director of the M.S. in Computer Engineering Program and Area Coordinate of Intelligent Systems, NJIT. He is also with the School of Electro-Mechanical Engineering, Xidian University, Xi'an, China. He organized and chaired over 80 technical sessions and served on program committees for many conferences. He has led or participated in 36 research and education projects with a total budget of over ten million dollars, funded by the National Science Foundation, Department of Defense, Engineering Foundation, New Jersey Science and Technology Commission, and industry. He was invited to lecture in Australia, Canada, China, France, Germany, Hong Kong, Italy, Japan, Korea, Mexico, Singapore, Taiwan, and U.S. and served as a plenary speaker for several conferences. He has over 300 publications including seven books, more than 130 journal papers, and 17 book chapters. He is the coauthor (with F. DiCesare) of Petri Net Synthesis for Discrete Event Control of Manufacturing Systems (Kluwer Academic, 1993), the Editor of Petri Nets in Flexible and Agile Automation (Kluwer Academic, 1995), the coauthor (with K. Venkatesh) of Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach (World Scientific, 1998), the coeditor (with M. P. Fanti) of Deadlock Resolution in Computer-Integrated Systems (Marcel Dekker, 2005), the coauthor (with H. Zhu) of Object-Oriented Programming in C++: A Project-based Approach (Tsinghua University Press, 2005), and the coauthor (with B. Hruz) of Modeling and Control of Discrete Event Dynamic Systems (Springer, 2007). His research interests are automated manufacturing systems, life cycle engineering and sustainability evaluation, Petri nets, wireless ad hoc and sensor networks, system security, semiconductor manufacturing, and embedded control.

Dr. Zhou is a Life Member of the Chinese Association for Science and Technology, USA, and he served as its President in 1999. He served as an Associate Editor of the IEEE Transactions on Robotics and Automation from 1997 to 2000 and the IEEE Transactions on Automation Science and Engineering from 2004–2007. He is currently the Managing Editor of the IEEE Transactions on Systems, Man and Cybernetics (SMC)—Part C: Applications and Reviews, the Editor of the IEEE Transactions on Automation Science and Engineering, and an Associate Editor of the IEEE Transactions on SMC—Part A: Systems and Humans and the IEEE Transactions on Industrial Informatics. He served as a Guest Editor for many journals including the IEEE Transactions on Industrial Electronics and the IEEE Transactions on Semiconductor Manufacturing. He was the General Chair of the IEEE Conference on Automation Science and Engineering, Washington, D.C., August 23–26, 2008; the General Cochair of the 2003 IEEE International Conference on SMC, Washington, D.C., October 5–8, 2003; the Founding General Cochair of the 2004 IEEE International Conference on Networking, Sensing and Control, Taipei, March 21–23, 2004; and the General Chair of the 2006 IEEE International Conference on Networking, Sensing and Control, Ft. Lauderdale, FL, April 23–25, 2006. He was the Program Chair of 1998 and 2001 IEEE International Conference on SMC and the 1997 IEEE International Conference on Emerging Technologies and Factory Automation. He was the Founding Chair of the Discrete Event Systems Technical Committee, the Founding Cochair of the Enterprise Information Systems Technical Committee of IEEE SMC Society, and the Chair (founding) of the Semiconductor Manufacturing Automation Technical Committee of the IEEE Robotics and Automation Society. He was the recipient of the National Science Foundation Research Initiation Award; the Computer-Integrated Manufacturing University-LEAD Award by the Society of Manufacturing Engineers; the Perlis Research Award by NJIT; the Humboldt Research Award for U.S. Senior Scientists; the Leadership Award and the Academic Achievement Award by the Chinese Association for Science and Technology, USA; the Asian American Achievement Award by the Asian American Heritage Council of New Jersey; and the Distinguished Lecturership of IEEE SMC Society.