Automated Verification of Dynamic Reliability Block Diagrams Using Colored Petri Nets[1]

Ryan Robidoux and Haiping Xu

Computer and Information Science Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
E-mail: {ryan.robidoux, hxu}@umassd.edu

Abstract—The increasing reliance on computer technology nowadays has resulted in a rapidly growing need to build reliable and fault resistant computer-based systems. Computer system reliabilities are conventionally modeled and analyzed using techniques such as fault tree analysis (FTA) and reliability block diagrams (RBD), which provide static representations of system reliabilities. A recent extension to RBD, called dynamic reliability block diagrams (DRBD), provides a framework for modeling dynamic reliability behaviors of computer-based systems. However, analyzing a DRBD model in order to locate and identify design errors, such as a deadlock error or a faulty state, is not trivial when done manually. A feasible approach to verifying a DRBD model is to develop a formal model of the DRBD, and 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 approach is compositional and provides a potential solution to automated verification of DRBD models.

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

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

NOMENCLATURE

BNF	Backus-Naur form.
CPN	Colored Petri nets.
DRBD	Dynamic reliability block diagram.
DFTA	Dynamic fault tree analysis.
FTA	Fault tree analysis.
RBD	Reliability block diagram.
RML	Reliability markup language.
SDEP	State-based dependency controller.
SPARE	Spare part controller.
SRBD	State-based reliability block diagram.
XML	Extensible markup language.

1. Introduction

In our modern society, there is an increased 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 (MTTF). 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. Some typical examples of such methods are reliability block diagrams (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 capability to accurately model a system's dependencies 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 the RBD, called dynamic reliability block diagrams (DRBD), was introduced with new controller constructs that support modeling dynamic, dependent and redundant relationships between components in a computer-based system [4]. 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 evaluation of a system's reliabilities. In our recent work, we demonstrated some preliminary results on how to formally verify a DRBD model using colored Petri nets (CPN) [4-5]. However, the proposed approach does not include a generalized solution for converting a DRBD model into CPN. In this paper, we present an algorithm that supports automatic conversion of a DRBD model into CPN. Hence, automatic verification of the DRBD model can be accomplished by analyzing the state space of the CPN using existing Petri net tools.

The rest of the paper is organized as follows. Section 2 summarizes the related efforts in reliability modeling. Section 3 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), was introduced to represent a DRBD model in XML format. Section 4 outlines the procedures to convert a DRBD into CPN. Section 5 provides a case study that illustrates how to create a DRBD model and convert it into a CPN model for formal analysis. Finally, Section 6 gives the conclusions and future work.

2. 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 [6]. Since FTA does not account for dynamic system properties, it was extended into dynamic FTA (DFTA) in order to model dynamic relationships between components [3, 7]. DFTA introduces additional gates for modeling sequential and sparing behaviors, but it has limited capability 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 [8]. 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 an interruption in the 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 behaviors, 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 well-known SHARPE tool. Blake *et al.* use an extension of Markov models to specify the reliability of multiprocessor systems using parametric sensitivity analysis [12]. 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 proposed dynamic RBD (DRBD) as an extension to RBD models [4-5]. New modeling constructs have been introduced and formally specified in Object-Z formalism [32], 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, a DRBD controlling construct can be used to model a general state-based dependency. Some similar work to our DRBD approach includes extensions to the RBD model proposed

in [9]; however, such extensions are unreasonably complex and difficult to use. As a result, they are not practically usable.

Petri nets [13-15] have been widely used in industry for modeling and analyzing computer-based systems such as intelligent mobile robots, semiconductor manufacturing systems, and automated production systems [16-18]. There are 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 [19]. They present the updated results in the areas of deadlock prevention, detection and recovery, and avoidance. Hsieh formulates a fault-tolerant deadlock avoidance controller synthesis problem for assembly processes based on a class of Petri nets, called controlled assembly Petri net (CAPN) [20]. 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 resources failures. Roszkowska studies the problems concerning the deadlock-free supervisory control for compound manufacturing processes using Petri nets [21]. She develops a polynomial algorithm that is capable of distinguishing between safe and unsafe states, and also formally proves that the proposed supervisor ensures the required behaviors of the system. Wu and Zhou propose a novel control policy for deadlock avoidance for automated guided vehicle (AGV) system using extended CROPN (colored resource-oriented Petri net) models, and the complexity of deadlock avoidance for the whole system is bounded by the complexity in controlling the AGV system [22].

Although the above Petri net based approaches can be used for deadlock detection and avoidance, they are not aimed at modeling system reliabilities. A few efforts on reliability modeling using Petri nets are summarized as follows. Bobbio *et al.* use the generalized stochastic Petri net (GSPN) to support system dependability analysis [23]. 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) [24]. 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 [25]. 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 [26]. 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 [27]. 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) [28] to verify the correctness of a DRBD model. As we demonstrate in the case study in Section 5, 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.

3. Dynamic Reliability Block Diagrams

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* and *SPARE* controllers 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 of RBD where each component is associated with a state representing the activeness of the component in the system. An SRBD model defines the static topology 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].

3.1 State-Based Reliability Block Diagrams

An SRBD is a network of dynamic system components called structural components. As defined in Figure 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 that represent atomic, 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 Figure 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 Figure 1, a serial component may contain only one component; thus, an SRBD with a single simple component or a single 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>)
```

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

Figure 2 shows an example of an SRBD model. In this example, two simple components (*C*1 and *C*2) are contained within a serial component, which itself is contained in a parallel component along with a third simple component (*C*3). Note that if not specified explicitly, we assume all simple components are initially in "Active" states.

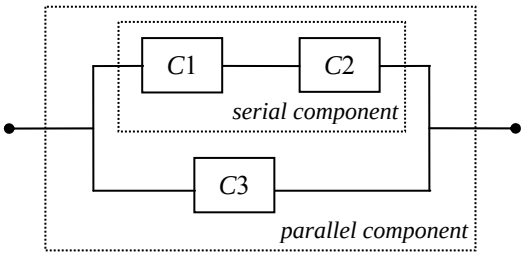


Figure 2. An example of a state-based reliability block diagram

3.2 DRBD Controller Blocks

Controller blocks defined in a DRBD model can be used to model dynamic relationships between components. Figure 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. 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 [29]. A warm spare component is a tradeoff between hot and cold configuration in terms of reconfiguration time and power consumption. To simplify matters, in this paper, we assume all spare components used in our examples are cold spares.

```
<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>
...
```

Figure 3. Definition of DRBD in BNF

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

On the other hand, a state controller models the state-based dependency relationships between the components in a system. With a state controller block, a *trigger* event that changes the state of a simple or spare component leads to *target* events, which are state changes on target components. An event can be one of the three types, namely "Activation," "Deactivation," and "Failure." An "Activation" event happening on a simple component causes the component to enter an "Active" state. Similarly, a "Deactivation" or "Failure" event happening on a component causes the component to enter a "Standby" or "Failed" state, respectively. A target component can be a simple or spare component, and the number of target components must be greater than zero. Figure 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.

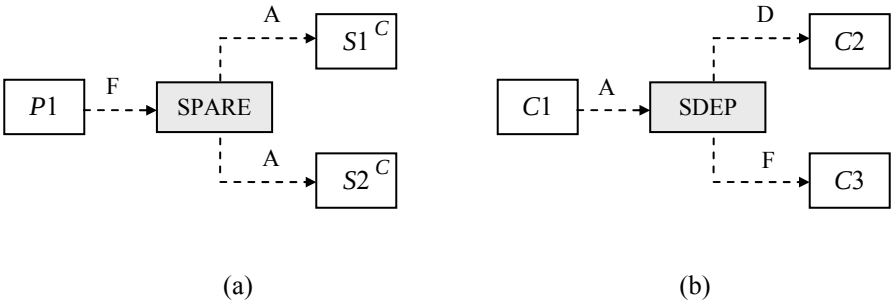
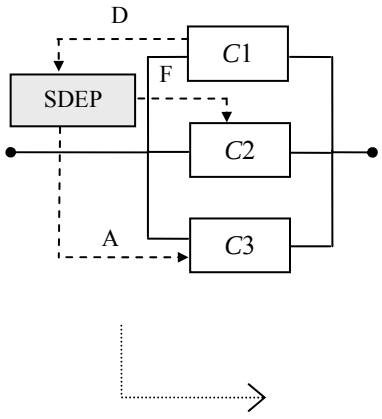


Figure 4 (a) SPARE controller block and (b) SDEP controller block

3.3 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 behaviors 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. Figure 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 failure of *C*1 leads to *C*2's failure as well as *C*3's activation. The right hand side

of Figure 5 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 the `MAIN` 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 only consists of simple ones, 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-based representation of a DRBD model

```
<?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>
```

Figure 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, in Figure 5, we define a state controller `C1_SDEP` that is within the `<stateController>` and `</stateController>` tags. Inside the definition of the state controller `C1_SDEP`, we define the trigger and target events 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 the 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 in a standard way to use that information [30]. Currently, two dominant APIs for processing XML-based documents are SAX and 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 the efficiency purpose, in this project, we have adopted the DOM specification to process XML-based representation of a DRBD model described in RML.

4. Conversion of DRBD Models into Colored Petri Nets

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 in the DRBD model are converted into Petri nets and added into the converted CPN model. The following sections give the detailed descriptions for the conversion procedures.

4.1 Conversion of SRBD into Colored Petri Nets

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. Figure 6 shows the conversion of a simple component into a colored Petri net, called simple component CPN.

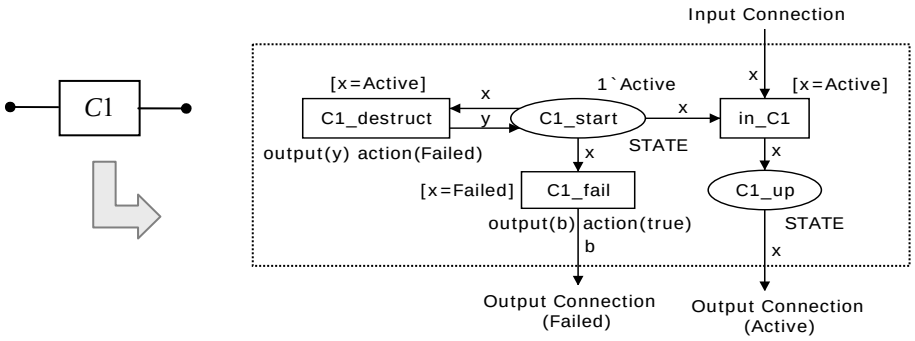


Figure 6. Simple component CPN for a simple component

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 Figure 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 Figure 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)*.

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

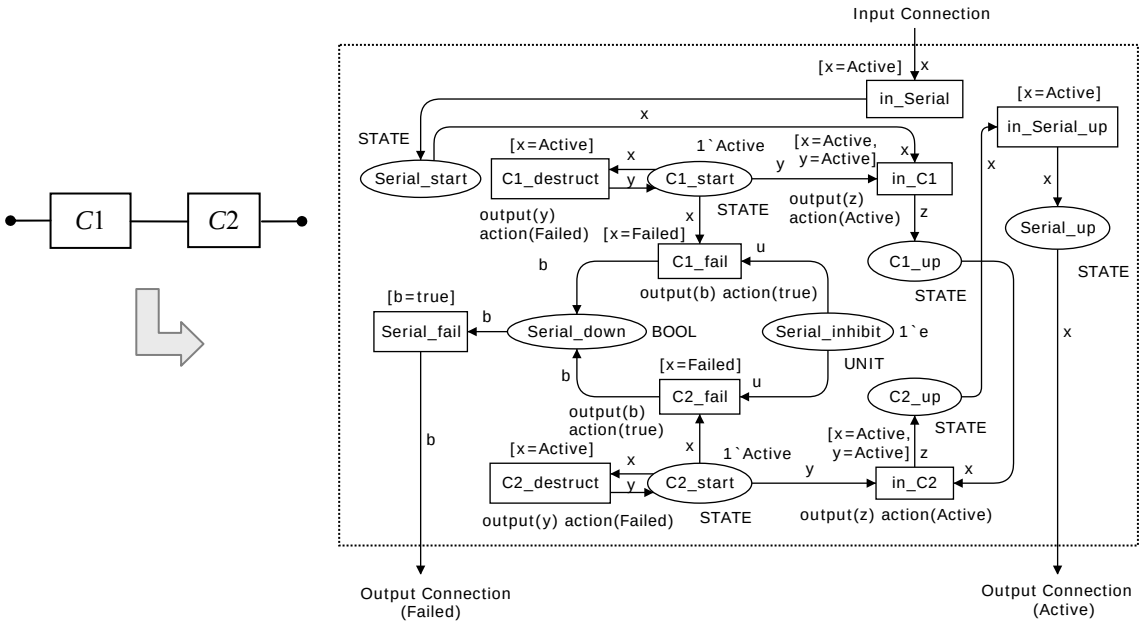


Figure 7. Serial component CPN for a serial component

The behavior of *C*1 in Figure 7 is the same as that of the simple component *C*1 in Figure 6. Note that both *C*1 and *C*2 in Figure 7 are modeled in exactly the same way as *C*1 in Figure 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 of the components *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. Figure 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 CPN 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).

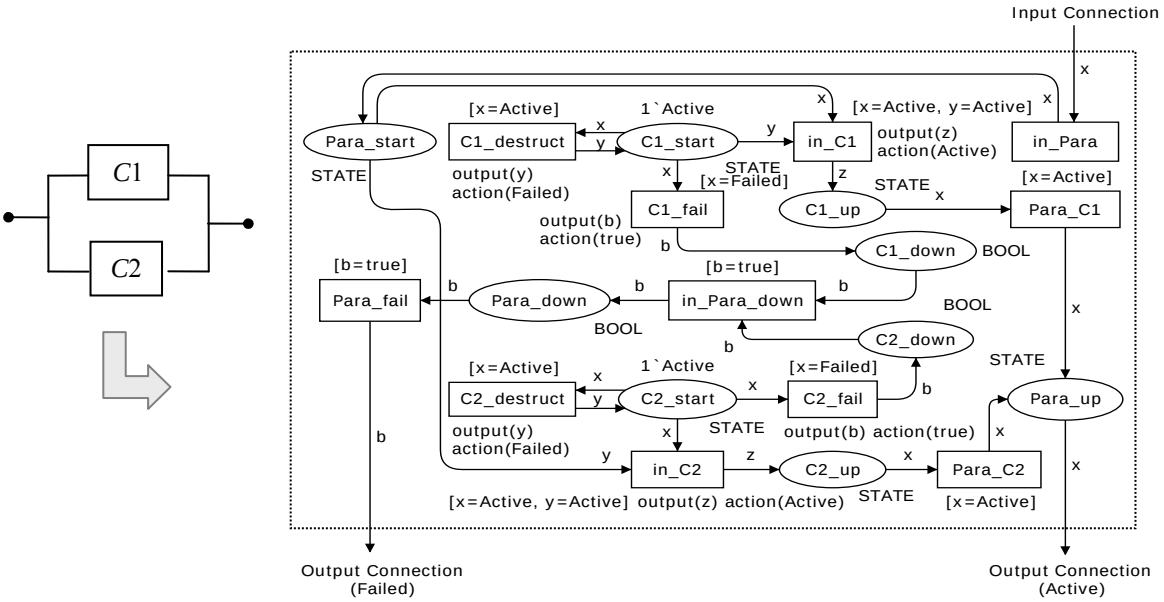


Figure 8. Parallel component CPN for a parallel component

*C*1 and *C*2 in Figure 8 are modeled in the same way as shown in Figure 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.

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 cases, colored Petri nets 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 Figure 9, which is defined as a recursive function `convert_Serial_Component` with a parameter of type `SerialComponent`.

```
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
```

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

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

`convert_Serial_Component` is called recursively. When all contained components in a parallel component have been converted into a CPN, all simple component CPN and serial component CPN are connected together (as shown in Figure 8) to create a parallel component CPN. Similarly, when all contained components in a serial component have been converted into colored Petri nets, all simple component CPN and parallel component CPN are connected together (as shown in Figure 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 Figure 10, an SRBD is treated as serial component *MAIN* with three major places *MAIN_start*, *MAIN_up*, and *MAIN_down*. *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, *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 dead marking state existing in the CPN model.

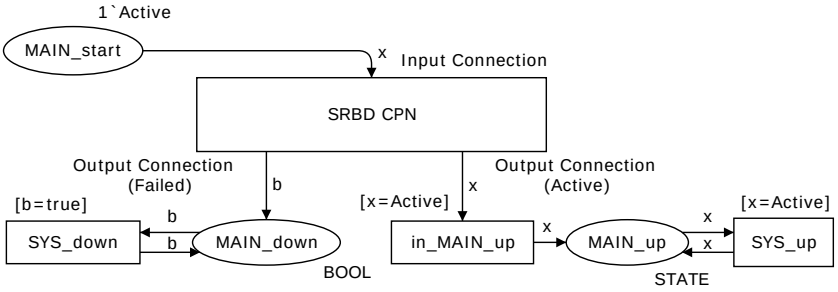


Figure 10. The complete CPN model for an embedded SRBD

4.2 Conversion of DRBD Controllers into Colored Petri Nets

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. Figure 11 shows a *SPARE* controller block that models the spare

part relationship between a 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 Figure 11, there are two 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.

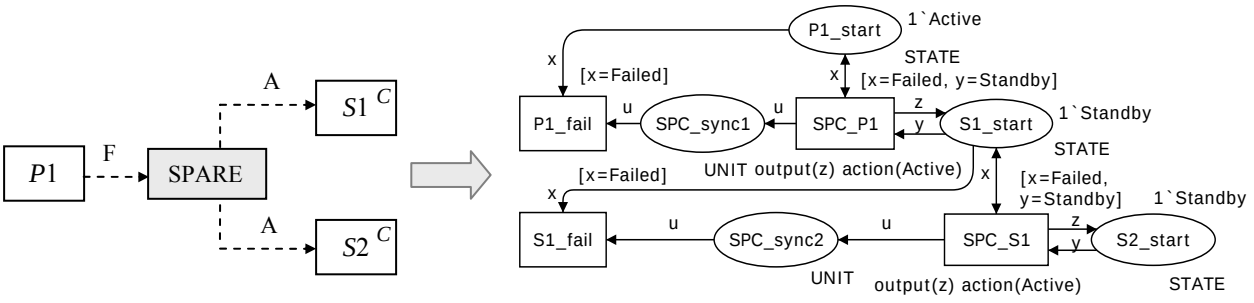


Figure 11. Spare controller CPN for a SPARE block

In a DRBD model, a state controller (i.e., an SDEP block) models a state-based dependency relationship between simple components. Figure 12 shows the state controller CPN for an SDEP block with a trigger component *C*1 and two target components *C*2 and *C*3. 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" token and a "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. Note that if the trigger event on 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. On the other hand, if the trigger event on *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 an "Standby" token in place *C*1*_start*.

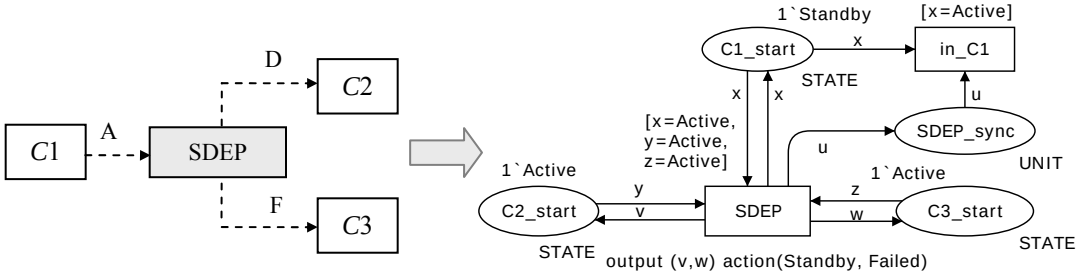


Figure 12. State controller CPN for an SDEP block

5. A Case Study: Conversion of DRBD into CPN for Formal Verification

5.1 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.

Figure 13 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.

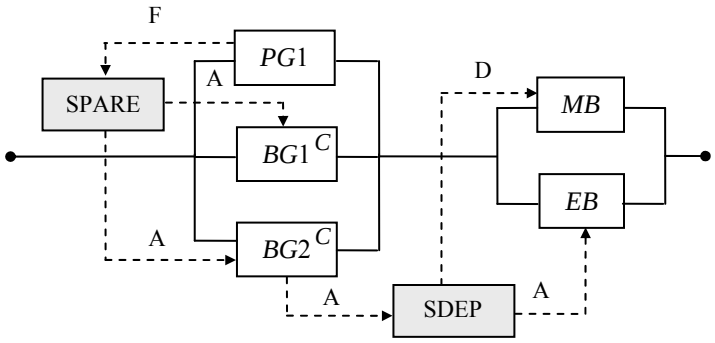


Figure 13. DRBD model of a redundant generator system

5.2 Automatic Generation of a CPN Model

According to the algorithm in Section 4, the DRBD model of the redundant generator system can be converted into a CPN model as shown in Figure 14. 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 Figure 14, 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 component is not functioning.

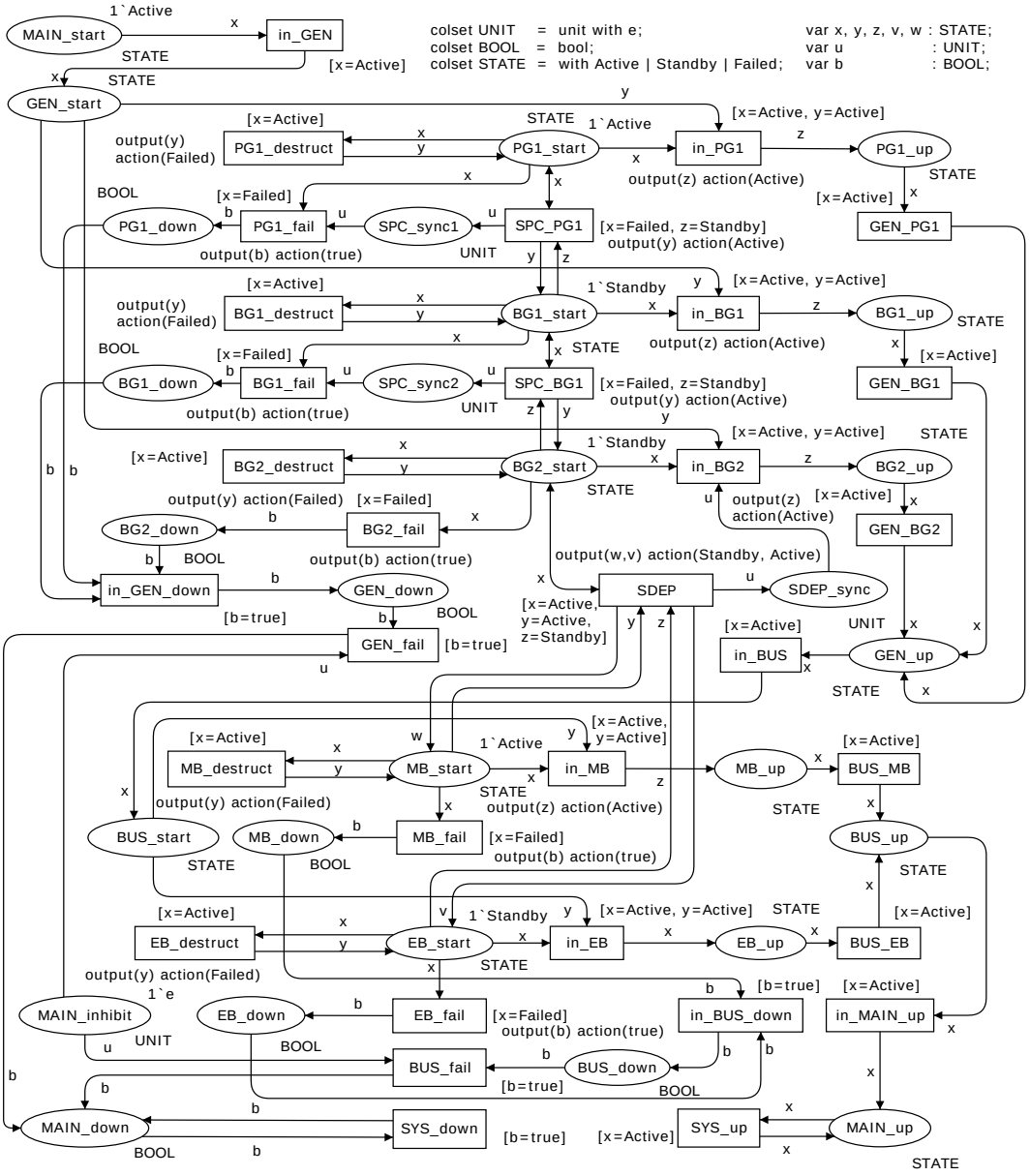


Figure 14. CPN model converted from the DRBD model in Figure 13

The second structural component contained in *MAIN* is parallel component *BUS* representing the parallel power bus circuit in the DRBD model shown in Figure 13. 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 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 inhibit 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* block. The *SPARE* 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 Figure 13, 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 component *BG*2, *MB* and *EB*, and its firing deposits a "Standby" token and an "Active" token into place *MB_start* and *EB_start*, respectively. *SDEP_sync* is used to ensure that the "Active" token is not accidentally removed before transition *SDEP* fires.

5.3 Analysis of DRBD Model Using Colored Petri Nets

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 [31], we can generate a report detailing the properties of the CPN model in Figure 14. The report, shown as *Result-1* in Table 1, indicates that there are three dead marking states in the CPN model, namely S_{78}, S_{171}, and S_{282}. The dead marking states 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 the dead marking states, we find the following firing sequences that lead to the dead marking states S_{78}, S_{171}, and S_{282}.

Table 1. Analysis results of the CPN application model in Figure 14

Result-1 (Before Revision)		**Result-2 (After Revision)**	
Statistics	**Liveness Properties**	**Statistics**	**Liveness Properties**

State Space
  Nodes: 288
  Arcs: 763
  Secs: 0
  Status: Full
Scc Graph
  Nodes: 288
  Arcs: 744
  Secs: 0
	Dead Markings
	  [78,171,282]
	Dead Transition Instances
	  Generator'BUS_fail 1
	  Generator'in_BUS_down 1
	Live Transition Instances
	  None
		State Space
		  Nodes: 897
		  Arcs: 2836
		  Secs: 1
		  Status: Full
		Scc Graph
		  Nodes: 897
		  Arcs: 2700
		  Secs: 0
			Dead Markings
			  None
			Dead Transition Instances
			  None
			Live Transition Instances
			  None

$\sigma_1 = <S_1, \textit{in\_GEN}, S_4, \textit{in\_PG1}, S_{10}, \textit{GEN\_PG1}, S_{19}, \textit{MB\_destruct}, S_{30}, \textit{in\_BUS}, S_{49}, \textit{MB\_fail}, S_{78}>$

$\sigma_2 = <S_1, \textit{PG1\_destruct}, S_2, \textit{in\_GEN}, S_6, \textit{SPC\_PG1}, S_{14}, \textit{in\_BG1}, S_{26}, \textit{GEN\_BG1}, S_{43}, \textit{MB\_destruct}, S_{55}, \textit{PG1\_fail}, S_{87}, \textit{MB\_fail}, S_{128}, \textit{in\_BUS}, S_{171}>$

$\sigma_3 = <S_1, \textit{PG1\_destruct}, S_2, \textit{SPC\_PG1}, S_7, \textit{BG1\_destruct}, S_{15}, \textit{SPC\_BG1}, S_{28}, \textit{SDEP}, S_{46}, \textit{EB\_destruct}, S_{74}, \textit{EB\_fail}, S_{114}, \textit{in\_GEN}, S_{143}, \textit{in\_BG2}, S_{184}, \textit{PG1\_fail}, S_{222}, \textit{BG1\_fail}, S_{251}, \textit{GEN\_BG2}, S_{271}, \textit{in\_BUS}, S_{282}>$

From firing sequence σ_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 σ_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 σ_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 Figure 13, 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 Figure 14 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 to analyze the revised CPN model, and get the *Result-2* as shown in Table 1. As illustrated by the table, the revised CPN model has no dead marking states, which guarantees the correctness of the revised DRBD model.

It is worth noting that the correct CPN model can be further used for analysis and evaluation of system reliability properties as demonstrated in [23-24]. Due to page limitations, detailed descriptions on reliability evaluation are not discussed in this paper, but will be presented in our future work.

6. 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 verifying dynamic reliability block diagram (DRBD) models of computer-based systems. In the procedure, DRBD models are first converted into colored Petri nets (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 dead marking 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. In our future work, we will study how to analyze a DRBD model for system reliability evaluation, and develop a comprehensive development environment that supports editing, verification and evaluation of DRBD models for complex and large computer-based systems.

References

[1] S. M. Rinaldi, "Modeling and Simulating Critical Infrastructures and Their Interdependencies," In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, Track 2, January 2004, Big Island, HI, USA.

[2] M. Rausand and A. Hoyland, *System Reliability Theory: Models, Statistical Methods an 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 *Proceedings of 3rd International High-Assurance Systems Engineering Symposium*, IEEE Computer Society Press, 1998, pp. 21–28.

[4] H. Xu and L. Xing, "Formal Semantics and Verification of Dynamic Reliability Block Diagrams for System Reliability Modeling," In *Proceedings of the 11th International Conference on Software Engineering and Applications*, November 19-21, 2007, Cambridge, Massachusetts, USA, pp. 155–162.

[5] H. Xu, L. Xing, and R. Robidoux, "DRBD: Dynamic Reliability Block Diagrams for System Reliability Modeling," *Technical Report*, Computer and Information Science Dept., UMass Dartmouth, August 2007.

[6] 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, 1981.

[7] H. Boudali, P. Crouzen, and M. Stoelinga, "Dynamic Fault Tree Analysis using Input/Output Interactive Markov Chains," In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, June 2007, Edinburgh, UK, pp. 708-717.

[8] A. Abd-Allah, "Extending Reliability Block Diagrams to Software Architecture," *Technical Report USC-CSE-97-501*, University of Southern California, March 1997.

[9] S. Distefano and L. Xing. "A New Approach to Modeling the System Reliability: Dynamic Reliability Block Diagrams," In *Proceedings of the 52nd Annual Reliability & Maintainability Symposium*, January 2006, Newport Beach, CA, pp. 189-195.

[10] R. A. Sahner and K. S. Trivedi, "Reliability Modeling using SHARPE," *IEEE Transactions on Reliability*, Vol. R-36, No.2, June 1987, pp. 186-193.

[11] C. Leangsuksun, H. Song, and L. Shen, "Reliability Modeling Using UML," In *Proceeding of 2003 International Conference on Software Engineering Research and Practice*, June 2003, Las Vegas, Nevada, USA, pp. 259-262.

[12] J. T. Blake, A. L. Reibman, and K. S. Trivedi, "Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor Systems," In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1988, pp. 177-186.

[13] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989, pp. 541-580.

[14] R. Zurawski and M. C. Zhou, "Petri Nets and Industrial Applications: A Tutorial," *IEEE Transactions on Industrial Electronics*, Vol. 41, No. 6, December 1994, pp. 567-583.

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

[16] F.-Y. Wang, K. J. Kyriakopoulos, A. Tsolkas, and G. N. Saridis, "A Petri-Net Coordination Model for an Intelligent Mobile Robot," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 4, July-August 1991, pp. 777-789.

[17] M. Jeng, X. Xie, and S.-L. Chung, "ERCN* Merged Nets for Modeling Degraded Behavior and Parallel Processes in Semiconductor Manufacturing Systems," *IEEE Transactions on Systems, Man and Cybernetics*, *Part A*, Vol. 34, No. 1, Jan. 2004, pp. 102-112.

[18] S. A. Reveliotis, "Avoidance versus Detection and Recovery Problem in Buffer-Space Allocation of Flexibly Automated Production Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, October 2000, Vol. 30, No. 5, pp. 799-811.

[19] M. P. Fanti and M. C. Zhou, "Deadlock Control Methods in Automated Manufacturing Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, January 2004, Vol. 34, No. 1, pp. 5-22.

[20] F.-S. Hsieh, "Fault-Tolerant Deadlock Avoidance Algorithm for Assembly Processes," *IEEE Transactions on Systems, Man and Cybernetics, Part A*, January 2004, Vol. 34, No. 1, pp. 65-79.

[21] E. Roszkowska, "Supervisory Control for Deadlock Avoidance in Compound Processes," *IEEE Transactions on Systems, Man and Cybernetics, Part A*, January 2004, Vol. 34, No. 1, pp. 52-64.

[22] N. Wu and M. C. Zhou, "Modeling and Deadlock Avoidance of Automated Manufacturing Systems with Multiple Automated Guided Vehicles," *IEEE Transactions on Systems, Man, and Cybernetics,*

Part B, December 2005, Vol. 35, No. 6, pp. 1193-1202.

[23] A. Bobbio, G. Franceschinis, L. Portinale, and R. Gaeta, "Exploiting Petri Nets to Support Fault-Tree Based Dependability Analysis," In *Proceedings of 8th International Workshop on Petri Nets and Performance Models*, September 1999, pp. 146-155.

[24] M. Everdij and H. Blom, "Petri-Nets and Hybrid-State Markov Processes in a Power-Hierarchy of Dependability Models," In *Proceedings of the IFAC Conference on Analysis and Design of Hybrid Systems*, June 2003, Saint-Malo, Brittany, France.

[25] N. G. Leveson and J. L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, March 1987, Vol. 13, No. 3, pp. 386-397.

[26] U. Buy and R. Sloan. "A Petri Net-Based Approach to Real-Time Program Analysis," In *Proceedings of the 7th International Workshop on Software Specification and Design*, December 6-7, 1993, Redondo Beach, California, pp. 56-60.

[27] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzc, "A Unified High-Level Petri Net Formalism for Time-Critical Systems," *IEEE Transactions on Software Engineering*, February 1991, Vol. 17, No. 2, pp. 160-172.

[28] 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.

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

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

[31] A. V. Ratzer, L. Wells, H. M. Lasen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN Tools for Editing, Simulating and Analyzing Colored Petri Nets," In *Proceedings of the 24th International Conference on the Application and Theory of Petri Nets*, Eindhoven, Netherlands, June 2003, pp. 450-462.

[32] 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.