Proceedings of the 11th IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 19–21, 2007, Cambridge, MA, USA
ISBN Hardcopy: 978-0-88986-705-5 / CD: 978-0-88986-706-2

FORMAL SEMANTICS AND VERIFICATION OF DYNAMIC RELIABILITY BLOCK DIAGRAMS FOR SYSTEM RELIABILITY MODELING

Haiping Xu
Computer and Information Science Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
hxu@umassd.edu

Liudong Xing
Electrical and Computer Engineering Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
lxing@umassd.edu

ABSTRACT
With the rapid advances in computer science and technology, critical computer-based systems, such as those in aerospace, military, and power industries exhibit more complex dependent and dynamic behaviors, which cannot be fully captured by existing reliability modeling tools. In this paper, we introduce a new reliability modeling tool, called dynamic reliability block diagrams (DRBD), for modeling dynamic relationships between components, such as state dependency and redundancy. We give formal semantics for some key DRBD constructs using Object-Z formalism. In order to verify and validate the correctness of a DRBD model, we propose to convert a DRBD model into a colored Petri net (CPN), and use an existing Petri net tool, called CPN Tools, to analyze and verify dynamic system behavioral properties. Our case study and experimental results show that DRBD provides a powerful tool for system reliability modeling, and our proposed verification approach can effectively ensure the correct design of DRBD reliability models for complex and large-scale computer-based systems.

KEY WORDS
Reliability modeling, dynamic reliability block diagrams (DRBD), Object-Z formalism, formal verification, colored Petri net (CPN), and CPN Tools

1. Introduction

With the rapid advances in computer science and technology, system reliability becomes an issue of increasing practical concern and research attentions. Critical computer based systems, such as those in aerospace, military, and power industries exhibit more and more complex dependent and dynamic behaviors. For example, due to a function dependency existing among components, where the failure of one component can cause other components to become inaccessible, components of a system do not necessarily fail independently. Failure to model dependency relationships accurately results in overstated or understated system reliability, which renders reliability analysis less effective in system design and tuning activities. Existing research efforts on this challenging issue do not fully address complexities of the dependency relationships among components. Sparing is another major dynamic behavior of mission critical systems. Specifically, a system with spares usually consists of one or more duplications of units for enhancing the system reliability. There are three types of sparing configurations, namely *hot*, *cold*, and *warm*. A hot spare operates in synchrony with a primary (i.e., online) component, and is prepared to take over at any time; a cold spare is unpowered until needed to replace a faulty component [1]. A warm spare is a trade-off between hot and cold spares in terms of reconfiguration time and power consumption. For all the three types of sparing approaches, a reconfiguration process happens when the primary component fails or is deactivated (i.e., put into the sleeping mode). Among the existing reliability modeling tools, only the dynamic fault tree (DFT) has the capability to model all the three types of redundant behaviors [2]. However, the DFT approach assumes that a reconfiguration can only be triggered by the failure of a primary component; it cannot model a situation where a reconfiguration is triggered by the deactivation of a primary component.

The reliability block diagram (RBD) model has been widely used as one of the most practical reliability modeling tools due to its simplicity [3, 4]. An RBD is a success-oriented network describing the functions of a system. Specifically, each RBD model consists of an input point, an output point, and a set of blocks. Each block represents a physical component that functions correctly. The blocks in the RBD are arranged in a way that illustrates the proper combinations of working components that keep the entire system operational [3]. Typically, if there is at least one path connecting between the input and output points, the system is operational. On the other hand, the failure of a component is indicated by the removal of the corresponding block in an RBD model; if enough blocks are removed to interrupt the connection between the input and output points, the system fails. The main virtues of the RBD model are that it is easy to read, and it is readily understood by customers, people who sell systems, engineers who design and test systems, and managers who make decisions on systems. With knowledge of the system, design engineers can easily construct and modify the corresponding RBD model, and communicate with people from different disciplines. However, similar to the other existing tools, the traditional RBD model has a distinct disadvantage that it cannot fully capture the dependent and dynamic behaviors of large and complex systems.

Motivated by the virtues of RBD models and the inadequacies of the existing modeling tools to model dependencies and dynamic relationships among components in a large and complex system, we introduce a new modeling approach called dynamic reliability block diagrams (DRBD). The DRBD model extends the traditional RBD model by fully considering the various dependencies and system dynamics. Some key DRBD modeling constructs, such as SDEP (State Dependency) block and SPARE (Spare Part) block, will be formally specified using the Object-Z formal specification language [5], which provides precise definitions for DRBD model behaviors. In addition, to ensure a correct design of a DRBD model that accurately represents the actual system in terms of its reliability behaviors, we will formally verify behavioral properties of the DRBD model. This requires a conversion of the DRBD model to a formalism, e.g., the Petri net formalism [6, 7], which is supported by effective analysis and verification tools.

2. Related Work

Previous work related to our research includes work on modeling dynamic reliability behaviors, and work on formal specification and verification of reliability models. Dynamic fault tree (DFT) has been proposed as an extension to the traditional (static) fault trees by including additional gates for modeling sequential and sparing behaviors [2]. The DFT approach also offers limited capability to model dependency relationships among components. However, it cannot accurately and fully model and analyze large systems subject to complex dynamics and dependencies. The ReliaSoft's BlockSim tool incorporates a standby container into the traditional RBD model for modeling the standby redundancy [8]; however, only cold spares are considered in this tool. More recently, Distefano and Xing introduced a set of DRBD constructs as an extension to the RBD models [9]. The DRBD constructs are used to model dynamic dependency relationships among components; however, compared to the DFT approach, the DRBD constructs introduced in [9] are very complicated and difficult to use, thus they are not practically usable. In this paper, we introduce a brand new set of DRBD constructs, which are based on simple notations; however, our newly introduced DRBD constructs are very powerful in modeling *state-based* dynamic system reliability behaviors.

Very little work has been done on formal specification of reliability modeling constructs. Coppit and his colleagues used the Z formalism to specify various DFT gates, such as AND, OR, KOFM and Priority AND (PAND) [10, 11]. The Z formalism is very useful in providing formal and precise definitions for DFT gates; however, in their approach, only state schemas are defined, while the operation schemas for modeling the dynamic behaviors of gates are missing. Furthermore, no solutions are provided for verification of DFT models to ensure a correct design. In contrast, we use the Object-Z formalism to specify both the state space and operations of a DRBD construct as a class schema. Furthermore, we propose and demonstrate a way of converting a DRBD model to a colored Petri net for formal verification and validation of DRBD models.

Additional related work to our proposed approach includes converting fault trees (FT) into the generalized stochastic Petri net (GSPN) to support dependability analysis [12]. The aim of the GSPN approach is to exploit the modeling and decision power of GSPN for both the qualitative and the quantitative analysis of the modeled system. Similarly, Everdij and Blom proposed to use dynamically colored Petri nets (DCPN) to develop PDP (Piecewise Deterministic Markov Processes) models [13]. They showed that DCPN has similar modeling power to PDP, and is more powerful than deterministic and stochastic Petri nets. Although the above approaches used Petri net formalism to model and analyze system reliability, they were not concerned with dynamic system reliability properties, such as state-based dependency. Furthermore, our approach has a major difference from their approaches: instead of providing quantitative analysis of system reliability directly using Petri nets, we use colored Petri nets to verify the correctness of a reliability model (i.e., a DRBD model). This is necessary because when dynamic reliability properties are involved, the DRBD model becomes very complicated for large and complex systems. Thus, it becomes vital to ensure the correctness of the DRBD model before any qualitative and quantitative analysis is conducted.

3. Dynamic Reliability Block Diagrams

Consider a cluster of sensor nodes in a clustered wireless sensor network system as shown in Figure 1. The cluster head manages the cluster by assigning duty cycles to sensor nodes and coordinating intra- and inter-cluster transmissions. The cluster head has a cold spare (i.e., the secondary cluster head in Figure 1) that is activated when the primary cluster head fails. All the sensor nodes within this cluster are divided into two mutually exclusive subsets (S1, S2). We assume each subset can provide desired sensing coverage, and in the beginning, sensor nodes in S1 are operational; while sensor nodes in S2 are in the sleeping mode. To preserve the limited energy of sensor nodes, the duty cycle of sensor nodes will be adjusted. At some other time, sensor nodes in S1 will be put into the sleeping mode and sensor nodes in S2 will be activated to maintain the desired sensing coverage. The entire cluster is considered to be operational when the sensor nodes in at least one of those two subsets are operational and one cluster head is functioning.

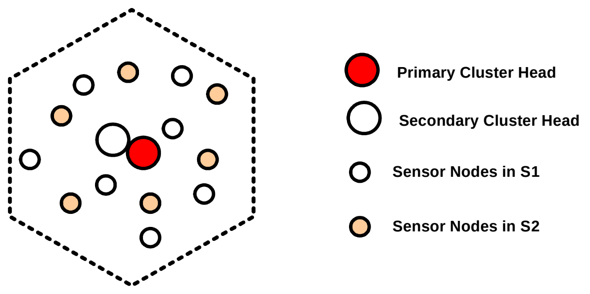


Figure 1. A clustered wireless sensor network system

For modeling the state dependency between the two subsets of the sensor nodes, i.e., the deactivation (or sleeping) of one subset of the sensor nodes leads to the activation (or wake-up) of the other subset of sensor nodes, we propose a new DRBD component called State Dependency (SDEP) block. Figure 2 illustrates the general structure of this block, where *A* means activation, *D* means deactivation, and *F* means failure. The occurrence of the trigger event will force all the dependent events to occur. Both the trigger event and the dependent events can be activation, deactivation or failure of a system component. In other words, our proposed SDEP block can be used to model nine types of dependencies among the system components: (*A*, *A*), (*A*, *D*), (*A*, *F*), (*D*, *A*), (*D*, *D*), (*D*, *F*), (*F*, *A*), (*F*, *D*), and (*F*, *F*). Here, it is important to mention that a DFT model cannot capture the above (*D*, *A*) state dependency behavior of the sensor network system; actually it can only be used to model the (*F*, *D*) state dependency where the failure of a component causes some other dependent components to become inaccessible or unusable. Compared with the existing DFT approach, the DRBD is more powerful in modeling dependent behaviors.

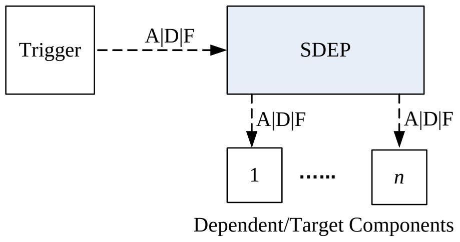


Figure 2. SDEP (state dependency) block

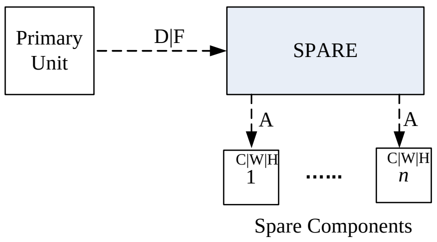


Figure 3. SPARE (spare part) block

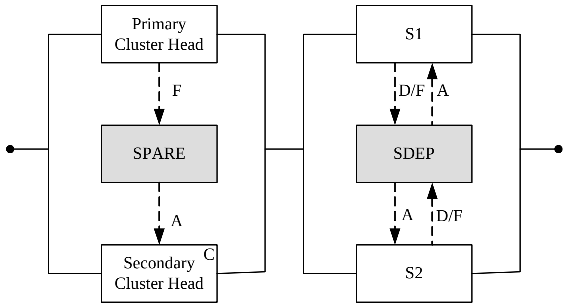


Figure 4. The DRBD model of the example system

For modeling the cold standby sparing cluster head subsystem, we propose another new DRBD component called SPARE (spare part) block. Figure 3 illustrates the general structure of this block, where *A*|*D*|*F* have the same meaning as in the SDEP block, and *C*|*W*|*H* means *cold*|*warm*|*hot* spare. Specifically, this block models the behavior that the deactivation or failure of the primary component will lead to the activation of a spare component, which could be in cold, warm, or hot standby state. All the spare units will be used in the specified order, i.e., from left to right.

Figure 4 gives the DRBD model of the wireless sensor network system as described in Figure 1. The model is developed using SDEP and SPARE blocks. The blocks labeled S1 and S2 represent a series structure of the sensor nodes that constitute the corresponding subset. Besides the capability of modeling the state dependencies and the various sparing behaviors, more new DRBD blocks and concepts have been proposed to model other dynamic relationships such as sequence dependency and load sharing. Sequence dependence enforces the order of occurrence of input events. A classic example of sequence dependence is a fault-tolerant system with one primary component and one standby spare connected with a switch controller [14]. If the switch controller fails after the primary component fails, and thus the standby component is switched into active operation, then the system can continue to operate. However, if the switch controller fails before the primary component fails, then the standby component cannot be activated, and the system fails even though the spare part is still operational. Similarly, a load sharing represents a condition where two or more components share the same workload. A load sharing condition usually involves components that perform the same task. Components in the load sharing redundancy exhibit different failure characteristics when one or more of them have failed or have been deactivated. The traditional fault trees, DFT, and RBD model do not consider the load sharing behavior. The BlockSim tool supports the load sharing configuration, but only considers the case where the increased load on the operating components happens when other load sharing components fail [8]. Both sequence dependency and load sharing are defined as DRBD constructs in our proposed approach; however, due to space limitation, in this paper, we only introduce those blocks used in our examples.

4. Formal Specification of DRBD Constructs

4.1 A Motivating Example

To support formal verification and validation of our proposed DRBD model, it is necessary to formally define the DRBD modeling constructs. Formal definitions of the DRBD constructs can not only provide the denotational semantics for the development of DRBD models in a precise manner, but also help to eliminate ambiguity in a constructed DRBD model. For example, Figure 5 shows a DRBD model with state dependencies and sparing relationship defined among components *C*1, *C*2, *C*3 and *C*4, where *C*4 is a cold spare for *C*2. Suppose component *C*1 fails at some time. According to the state dependency (*F*, *F*) from *C*1 to *C*2, and (*F*, *D*) from *C*1 to *C*3, component *C*2 and *C*3 will become *Failed* and *Standby*, respectively. Since component *C*4 is a spare unit of component *C*2, the failure of component *C*2 will lead to the activation of component *C*4; however, since there is a (*D*, *D*) state dependency from component *C*3 to *C*4, the deactivation of component *C*3 will lead to the deactivation of component *C*4. To evaluate the system's reliability, we have to answer the following question: when component *C*1 fails, will component *C*4 be in a state of *Active* or *Standby*, or will the result be nondeterministic? The above question can actually be reduced to the following question: when component *C*1 fails, will the two state dependencies (from *C*1 to *C*2, and from *C*1 to *C*3) happen immediately and thus simultaneously, or with some nondeterministic time delay? To answer this type of questions, it is vital for us to formally define the denotational semantic of the DRBD constructs. Here we propose to use the Object-Z formalism [5] for this purpose.

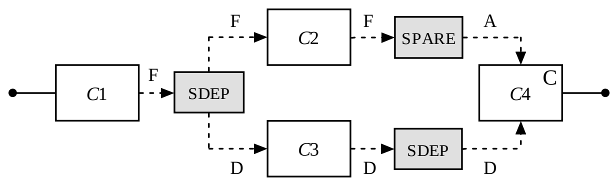


Figure 5. A DRBD model with conflicts

4.2 Specification of DRBD Constructs in Object-Z

Object-Z is an extension to the Z formal specification language for modular design of complex systems [5, 15]. It has strong data and state modeling capabilities, which is suitable for specifying the formal semantics of DRBD modeling constructs. Figure 6 shows the formal specification of the state dependency (SDEP) block in Object-Z. In this specification, we first define *Event* occurring on a component as an enumerated type with the following values: *Activation*, *Deactivation*, and *Failure*, which will lead a component to a state of *Active*, *Standby* and *Failed*, respectively. Then we define a state dependency as a block that consists of a trigger component and a set of target/dependent components. The relationship between the trigger component and the target components are defined by a function *sdep*, which maps the trigger event happening at time *t*? to a set of target events happening at $t?+\delta_c$, where *c* represents a target component. This definition precisely states that the target events do not need to happen simultaneously as long as δ_c is sufficiently small. The three operations defined on a state dependency are *ActivateTrigger*, *DeactivateTrigger*, and *FailTrigger*. According to the definition, the activation of the trigger component may lead to one of three different states for each target component, i.e., *Active*, *Standby* and *Failed*, which correspond to three different relationships between the trigger and one of the targets, namely (*A*, *A*), (*A*, *D*), and (*A*, *F*) state dependency. Similarly, the formal definition of *DeactivateTrigger* operation specifies how state dependencies (*D*, *A*), (*D*, *D*), and (*D*, *F*) are enforced; while the formal definition of *FailTrigger* specifies how state dependencies (*F*, *A*), (*F*, *D*), and (*F*, *F*) are enforced.

$Event ::= Activation \mid Deactivation \mid Failure$

SDEP

$trigger : Component$
$targets : \mathbb{P}\ Component$
$nTargets : \mathbb{N}$
$triggerEvent : Event$
$targetEvents : Component \rightarrow Event$
$sdep : \mathbb{T} \times Component \times Event \rightarrow \mathbb{P}(\mathbb{T} \times Component \times Event)$

$nTargets = \#targets \wedge nTargets > 0 \wedge targets = dom\ targetEvents$
$\forall\, c \in targets \bullet c \neq trigger$
$\quad \wedge probability(c \mid triggerEvent) \neq probability(c)$
$\quad \wedge\ probability(triggerEvent \mid c) = probability(triggerEvent)$
$\{(t, trigger, triggerEvent) \mid t \in \mathbb{T}\} = dom\ sdep$

ActivateTrigger

$\Delta(trigger, targets)$
$t? : \mathbb{T}$

$(triggerEvent = Active) \wedge (trigger.state' = Active)$
$\forall\, c \in targets \bullet$
$\quad (t? + \delta_c, c, targetEvents(c)) \in sdep(t?, trigger, triggerEvent)$
$\quad \wedge\ ((targetEvents(c) = Activation \wedge c.state' = Active)$
$\quad \vee\ (targetEvents(c) = Deactivation \wedge c.state' = Standby)$
$\quad \vee\ (targetEvents(c) = Failure \wedge c.state' = Failed))$

DeactivateTrigger

$\Delta(trigger, targets)$
$t? : \mathbb{T}$

$(triggerEvent = Deactivation) \wedge (trigger.state' = Standby)$
$\forall\, c \in targets \bullet$
$\quad (t? + \delta_c, c, targetEvents(c)) \in sdep(t?, trigger, triggerEvent)$
$\quad \wedge\ ((targetEvents(c) = Activation \wedge c.state' = Active)$
$\quad \vee\ (targetEvents(c) = Deactivation \wedge c.state' = Standby)$
$\quad \vee\ (targetEvents(c) = Failure \wedge c.state' = Failed))$

FailTrigger

$\Delta(trigger, targets)$
$t? : \mathbb{T}$

$(triggerEvent = Failure) \wedge (trigger.state' = Failed)$
$\forall\, c \in targets \bullet$
$\quad (t? + \delta_c, c, targetEvents(c)) \in sdep(t?, trigger, triggerEvent)$
$\quad \wedge\ ((targetEvents(c) = Activation \wedge c.state' = Active)$
$\quad \vee\ (targetEvents(c) = Deactivation \wedge c.state' = Standby)$
$\quad \vee\ (targetEvents(c) = Failure \wedge c.state' = Failed))$

Figure 6. Object-Z specification of SDEP block

Figure 7 shows the formal specification of the SPARE (spare part) block in Object-Z. We define a spare part redundancy as a block that consists of a primary unit component and a set of alternative (spare) unit components. The relationships among them are defined by the function *switch*, which maps a state change event happened on a primary unit or an alternative unit at time *t*? to a state change event happened on an alternative unit at *t*?+δ_c, where *c* represents an alternative unit component. The *PrimarySwitch* operation defines when the primary unit is deactivated or failed, the first alternative unit will be activated; while the *AlternativeSwitch* operation defines when alternative unit *i*? is deactivated or failed, alternative unit *i*?+1 will be activated. Note that the previous state of an activated alternative unit must be *Standby*, which can be in one of the three cases: *Hot*, *Cold* and *Warm*.

$$Standby ::= Hot \mid Cold \mid Warm$$

SPARE

$$\begin{array}{l}
primaryUnit : Component \\
alternativeUnits : \mathbb{P}\ Component \\
nAlternatives : \mathbb{N} \\
alternative : \mathbb{N} \rightarrow Component \\
switch : \mathbb{T} \times Component \times Event \rightarrow \mathbb{T} \times Component \times Event \\
\hline
\forall\, c \in alternativeUnits \bullet c \neq primaryUnit \\
nAlternatives = \#alternativeUnits \wedge nAlternatives > 0 \\
\{1, 2, ..., nAlternatives\} = dom\ alternative \\
dom\ switch = \{(t, c, e) \mid t \in \mathbb{T}, c \in \{primaryUnit\} \cup \\
\quad \{(alternative(i)) \mid 1 \leq i \leq nAlternatives - 1\}, \\
\quad e \in \{Deactivation, Failure\}\}
\end{array}$$

Init

$$\begin{array}{l}
trigger.state = Active \\
\forall\, i, where\ 1 \leq i \leq nAlternatives \bullet alternative(i).state = Standby
\end{array}$$

PrimarySwitch

$$\begin{array}{l}
\Delta(primaryUnit, alternativeUnits) \\
t? : \mathbb{T} \\
\hline
\forall\, e \in \{Deactivation, Failure\} \bullet \\
\quad switch(t?, primaryUnit, e) = (t? + \delta_c, alternative(1), Activation) \\
(primaryUnit.state = Active) \wedge \\
\quad (primaryUnit.state' \in \{Standby, Failed\}) \\
\quad \wedge\ (alternative(1).state' = Active)
\end{array}$$

AlternativeSwitch

$$\begin{array}{l}
\Delta(alternativeUnits) \\
t? : \mathbb{T}, i? : \mathbb{N} \\
\hline
\forall\, e \in \{Deactivation, Failure\}, 1 \leq i? \leq nAlternatives - 1 \bullet \\
\quad switch(t?, alternative(i), e) = \\
\quad (t? + \delta_c, alternative(i + 1), Activation) \\
\quad \wedge\ (alternative(i?).state = Active) \\
\quad \wedge\ (alternative(i?).state' \in \{Standby, Failed\}) \\
\quad \wedge\ (alternative(i? + 1).state = Standby) \\
\quad \wedge\ (alternative(i? + 1).state' = Active)
\end{array}$$

Figure 7. Object-Z specification of SPARE block

The introduction of new DRBD constructs as an extension to the RBD model can greatly enhance the modeling power for system reliability modeling. However, to derive a correct result from a DRBD model in an industrial setting, we must first face one major concern, which is how we can be certain that the model is correct. In other words, how can we be confident that the model is an accurate representation of the actual system for its reliability properties. This problem is not severe when we use the standard RBD models, because it only contains a few static modeling constructs. However, when we design DRBD models, which involve new dynamic conceptual modeling constructs, engineers are more potentially to bring design errors into the model due to the complexity of the newly introduced dynamic modeling constructs, e.g., the state dependency. Such design errors could be very subtle and very difficult to detect when the model is not trivial, and it will result in an incorrect reliability model, which will lead to inaccurate results when the model is evaluated. Traditional simulation approaches to model testing is not suitable for DRBD models because it is hard (almost impossible) to cover all execution paths. A promising way to solve this problem is to use formal methods to verify the behavioral properties of the model before the evaluation process starts. That is, to verify if the DRBD model satisfies the specified behavioral properties of the system under investigation. For example, we may use temporal logic [16] to specify the following system property of a computer system: "if component *A* fails, component *B* and *C* will also fail, which leads to the failure of the whole system *S*." The temporal formula in LTL (Linear Temporal Logic) can be written as `[](¬A→(¬B∧¬C)∧<>¬S)`, where the box `[]` and the diamond `<>` represent the *always* operator and *eventually* operator, respectively. The above temporal formula says that it is always true that the failure of *A* immediately leads to the failure of *B* and *C*, and will eventually leads to the failure of the system *S*. One way to verify such a system behavioral property is to use the model checking technique [17]. In this case, a `valid` result shows that the reliability model developed for system *S* does have this property; while an `invalid` result shows that the model developed for system *S* is incorrect. When a DRBD model is proved to be incorrect, any quantitative evaluation results derived from the DRBD model might not be usable, so the DRBD model must be corrected and re-verified.

5. Formal Verification of DRBD Models

Although the semantics of DRBD components and constructs can be formally defined in Object-Z as shown in Section 4, it is not straightforward and feasible to verify the behavioral properties of DRBD models based on the Object-Z formalism due to a lack of analysis and verification tool support [18]. A better approach to verifying a DRBD model is to convert it into a formal model such as a state machine or a Petri net model, which is supported by powerful verification tools. We adopt the Petri net formalism because it has a distinct advantage of being easy to understand and use due to its graphical formulation, and the powerful, but intuitive rules for defining structure and dynamic behaviors [6]. Petri nets provide an intuitive, graphically defined, way to express conditions, events, and their relationships, as well as essential characteristics like nondeterminism and concurrency. Specifically, in a Petri net graph, there are nodes that represent conditions (called "places") and nodes that represent events (called "transitions"). When some specified conditions hold true, events can occur – this is depicted in the net model by the "firing" of transitions, which causes new conditions to hold. So, a Petri net provides an executable model that directly defines the concept of a system's state space. Although most research concerned with automated analysis of concurrent and distributed systems is based on some type of state-space exploration and cannot avoid the state-space explosion problem. Based on our significant experience with Petri nets for many years, the Petri net formalism is capable of achieving an effective balance between theoretical concepts and practical techniques.

In the following, we use a DRBD model of a router example to show how we can convert it into a colored Petri net and use an existing Petri net tool to detect possible design errors in the DRBD model.

Figure 8 shows the DRBD model of the router example, which contains a router (Component *C*1) connected with a computer (Component *C*2). The computer can access the Internet only through the router. Therefore, there is a (*F*, *D*) state dependency from Component *C*1 to *C*2, i.e., when the router fails, the computer will be deactivated for its network connection, and will be in a "Standby" state. To make the system more reliable, we introduce a cold-spare part for the router, which is represented by Component *C*3 in Figure 9. The DRBD model in Figure 9 describes when the Component *C*1 fails, Component *C*3 will be automatically activated and continue to provide network access for Component *C*2.

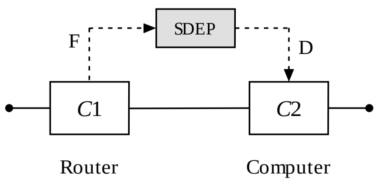


Figure 8. DRBD model of a router example

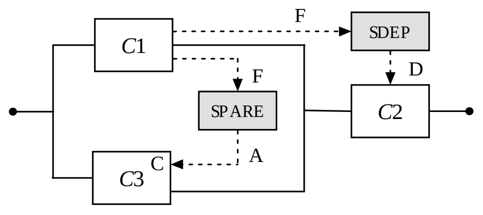


Figure 9. DRBD model of a router with a cold-spare part

The DRBD model in Figure 9 can be converted into a colored Petri net as shown in Figure 10. The three components in Figure 9 are modeled by the three places *Component_1*, *Component_2* and *Component_3*, and each place can contain a colored token with three different colors, i.e., "Active", "Standby" and "Failed". Each component may fail when its corresponding *Fail_i* (*i* = 1..3) transition fires, in which case, the "Active" token in the corresponding component place will be replaced by a "Failed" token. Note that since we only allow *cold-standby* component in this example, a component in a "Standby" state cannot fail.

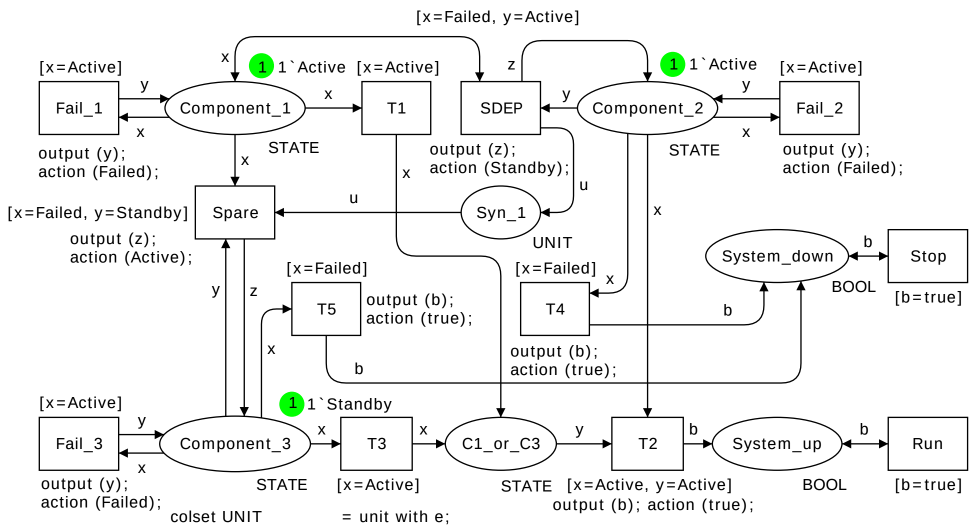
Initially, the three component places contain "Active", "Standby" and "Active" colored token, respectively. According to the DRBD model in Figure 9, when either Component 1 or Component 3 is active, the parallel structure is functioning. This is represented by firing either transition *T1* or *T3* when either the place *Component_1* or *Component_3* contains an "Active" token, thus an "Active" token can be deposited into the place *C1_or_C3*. When Component *C*2 is also active, the transition *T2* may fire, and a Boolean token "true" is deposited into place *System_up*, which enables the *Run* transition. This scenario shows that when Component *C*1 (or Component *C*3) and Component *C*2 are functioning, the system must be functioning. On the other hand, when either Component *C*2 or *C*3 fails, the transition *T4* or *T5* may fire, and the firing of the transition *T4* or *T5* deposits a Boolean token "true" into place *System_down*, which indicates that the system cannot be functioning. However, if Component *C*1 fails, it will activate Component *C*3 due to the spare part redundancy; thus, it should not lead to the failure of the whole system.

When Component *C*1 fails and Component *C*2 is active, the transition *SDEP* can fire, and deposit a "Standby" token into the *Component_2* place. Meanwhile, a unit token is deposited into the synchronization place *Syn_1* to ensure that the firing of transition *SDEP* precedes the firing of transition *Spare*, so the transition *SDEP* will not accidentally become disabled when the "Failed" token in place *Component_1* is removed due to the firing of transition *Spare*. When transition *Spare* fires, it deposits an "Active" token into place *Component_3* to activate the spare part. This should lead to the continuous functioning of the whole system.

We now use an existing Petri net tool called CPN Tools [19] to analyze our Petri net model. After running the analysis tool, we get the Result-1 as shown in Table 1.

Table 1. Results from state space analysis tool

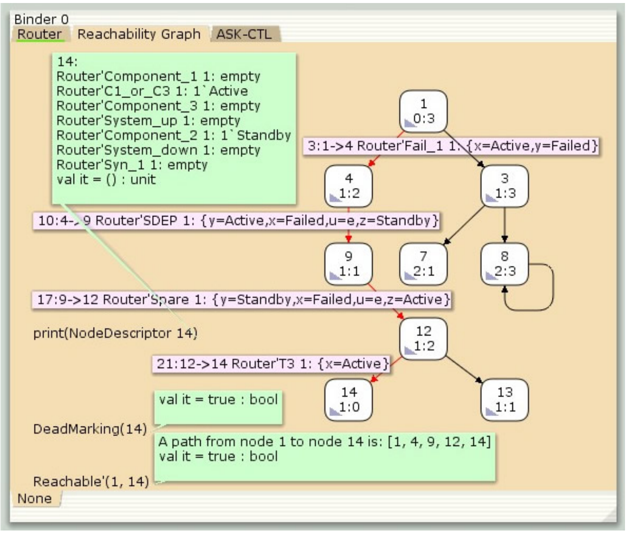
```
           Result-1                      Result-2
Statistics                          Statistics
--------------------                --------------------
State Space                         State Space
   Nodes:  15                          Nodes:  27
   Arcs:   23                          Arcs:   48
   Secs:   0                           Secs:   0
   Status: Full                        Status: Full

Scc Graph                           Scc Graph
   Nodes:  15                          Nodes:  27
   Arcs:   18                          Arcs:   36
   Secs:   0                           Secs:   0

Liveness Properties                 Liveness Properties
--------------------                --------------------
  Dead Markings                       Dead Markings
     [14]                                None
```



```
colset UNIT      = unit with e;
colset BOOL      = bool;
colset STATE     = with Active | Standby | Failed;
var x, y, z       : STATE;
var u             : UNIT;
var b             : BOOL;
```

Figure 10. The colored Petri net model converted from the DRBD model in Figure 9

The result shows that there is a dead marking (state 14) in the reachability graph of the Petri net model. From Figure 11 (a snapshot of the state space tracing), we can see that in the dead marking S_{14}, both the places *System_up* and *System_down* contains no token. This implies that the Petri net model must contain a deadlock. By tracing the dead marking state using the CPN Tools, we find the following firing sequence that leads to the dead marking. The firing sequence is: S_1, *Fail_1*, S_4, *SDEP*, S_9, *Spare*, S_{12}, *T3*, S_{14}. From Figure 10, it is easy to see that dead marking is due to the firing of transition SDEP that deposits a "Standby" token in place *Component_2* when Component *C*1 fails. When Component *C*3 is activated, Component *C*2 should also be activated accordingly. However, such state dependency from Component *C*3 to *C*2 is not presented in the DRBD model in Figure 9. This design error must be fixed by adding an (*A*, *A*) state dependency from Component *C*3 to Component *C*2 in Figure 9. Accordingly, we need to revise the colored Petri net model in Figure 10 as follows: (1) add a new transition *SDEP_1* with both the places *Component_2* and *Component_3* as the input places and output places; (2) add a new synchronization place *Syn_2* with *SDEP_1* as input transition and *T3* as output transition; (3) set the condition of transition *SDEP_1* such that *Component_3* contains an "Active" token and *Component_2* contains a "Standby" token; and (4) set the output of transition *SDEP_1* as an "Active" token deposited into the *Component_2* place. We analyze the revised colored Petri net again using the CPN Tools, and now we get the results as shown in Table 1, Result-2. The results indicates that there is no dead marking in the revised Petri net model, which ensures that the revised Petri net model is deadlock free.


Binder 0
Router | Reachability Graph | ASK-CTL

```
14:
Router'Component_1 1: empty
Router'C1_or_C3 1: 1`Active
Router'Component_3 1: empty
Router'System_up 1: empty
Router'Component_2 1: 1`Standby
Router'System_down 1: empty
Router'Syn_1 1: empty
val it = () : unit
```

```
3:1->4 Router'Fail_1 1: {x=Active,y=Failed}
10:4->9 Router'SDEP 1: {y=Active,x=Failed,u=e,z=Standby}
17:9->12 Router'Spare 1: {y=Standby,x=Failed,u=e,z=Active}
21:12->14 Router'T3 1: {x=Active}
```

```
print(NodeDescriptor 14)
DeadMarking(14)
val it = true : bool
Reachable'(1, 14)
A path from node 1 to node 14 is: [1, 4, 9, 12, 14]
val it = true : bool
```

None

Figure 11. State space tracing of the dead marking state

In addition, we can also use model checking technique to verify some other properties of the colored Petri net model. For example, the following code in ML language defines three functions: *IsRunning*, *IsFailed_1* and *IsFailed_2*. The function *IsRunning* returns *true* if the place *System_up* contains a colored token "true", which indicates the system is functioning. Similarly, the function *IsFailed_1* (or *IsFailed_2*) returns *true* when the place *System_down* contains one (or two) colored token(s) "true", which indicates that one (or both) of the Component *C*2 and *C*3 fails. An operator `FORALL_UNTIL(`A_1`, `A_2`)` is true if for all paths in the reachability graph of the Petri net model, A_1 is true for each state along the path until reaching a state on the path where A_2 must hold. Therefore, the CTL formula `myASKCTLformula` specifies that in all paths of the reachability graph, it eventually becomes true where either the system is functioning, or the system is not functioning due to one or both of Component *C*2 and *C*3 fails. Note that in the CTL formula, "TT" represents the constant "true" value.

```
fun IsRunning n =(Mark.System_up 1 n = 1`true);
fun IsFailed_1 n =(Mark.System_down 1 n = 1`true);
fun IsFailed_2 n =(Mark.System_down 1 n = 2`true);
val  fail  =  OR(NF("Is  failed",  IsFailed_1),
NF("Is failed", IsFailed_2));
val sys = OR(NF("Is running", IsRunning), fail);
val myASKCTLformula = FORALL_UNTIL(TT, sys);
eval_node myASKCTLformula InitNode;
```

The model checking result of the above formula for the revised Petri net model is "true", which implies the proper behavior of the net model. However, when we evaluate the formula for the initial design of the Petri net model (as shown in Figure 10), a "false" result is returned. This is because both of the places *System_up* and *System_down* are empty along the path that leads to the dead marking, which indicates a design error.

6. Conclusion and Future Work

Existing system reliability modeling approaches cannot fully capture dynamic relationships between components, such as state dependency and redundancy. In this paper, we propose a new modeling approach called dynamic reliability block diagrams (DRBD) to resolve the shortcomings of the existing work. Our proposed approach provides a powerful but easy-to-use modeling tool for complex and large computer-based systems. Our formal verification approach can be used to ensure a correct design of DRBD models. In our future work, we will design conversion algorithms to support automatic translation of DRBD models to colored Petri nets. We also plan to develop a system reliability modeling tool that supports editing, formal verification, and evaluation of DRBD models for complex and large-scale systems.

References

[1] B. W. Johnson, *Design and analysis of fault tolerant digital systems* (Boston, USA, Addison-Wesley Longman Publishing Co. Inc., 1989).

[2] R. Manian, J. B. Dugan, D. Coppit, and K. J. Sullivan, Combining various solution techniques for dynamic fault tree analysis of computer systems, *Proc. of the IEEE International High-Assurance Systems Engineering Symposium*, 1998.

[3] M. Rausand and A. Høyland, *System reliability theory: models, statistical methods, and applications* (New York, USA, Wiley-Interscience, 2003).

[4] W. Wang, J. M. Loman, R. G. Arno, P. Vassiliou, E. R. Furlong, and D. Ogden, Reliability block diagram simulation techniques applied to the IEEE std. 493 standard network, *IEEE Transactions on Industry Applications*, *40*(3), May/June 2004, pp. 887-895.

[5] 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.

[6] T. Murata, Petri nets: properties, analysis and applications, *Proc. of the IEEE*, Vol. 77, No. 4, April 1989, pp. 541-580.

[7] K. Jensen, *Colored Petri nets: basic concepts, analysis methods and practical use, volume 2, analysis methods* (Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997).

[8] BlockSim, *System reliability analysis software using an RBD or fault tree approach*, ReliaSoft Corporation, http://www.reliasoft.com/BlockSim/, accessed on June 12, 2007.

[9] S. Distefano and L. Xing, A new approach to modeling the system reliability: dynamic reliability block diagrams, *Proc. of the 52nd Annual Reliability & Maintainability Symposium*, Newport Beach, CA, January 2006, pp. 189-195.

[10] D. Coppit, K. J. Sullivan, and J. B. Dugan, Formal semantics of models for computational engineering: a case study on dynamic fault trees, *Proc. of the International Symposium on Software Reliability Engineering*, San Jose, California, 2000, pp. 270-282.

[11] D. Coppit and K. J. Sullivan, Formal specification in collaborative design of critical software tools, *Proc. of the Third IEEE International High-Assurance Systems Engineering Symposium*, Washington, D.C., November 13-14, 1998, pp. 13-20.

[12] A. Bobbio, G. Franceschinis, R. Gaeta, L. Portinale, Exploiting Petri nets to support fault tree based dependability analysis, *Proc. of the 8th International Workshop on Petri Nets and Performance Models (PNPM)*, 1999, pp. 146-155.

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

[14] E. J. Henley and H. Kumamoto, *Probabilistic Risk Assessment: Reliability Engineering, Design, and Analysis* (IEEE Press, 1992).

[15] S. Stepney, R. Barden, and D. Cooper, editors, *Object orientation in Z*, Workshops in Computing, Springer, 1992, pp. 59-77.

[16] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification* (Springer-Verlag New York, Inc, 1992).

[17] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking* (MIT Press, 2001).

[18] G. Kassel and G. Smith, Model checking Object-Z classes: some experiments with FDR," *Proc. of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001, pp. 445-452.

[19] A. V. Ratzer, L. Wells, H. M. Lassen, *et. al.*, CPN Tools for editing, simulating, and analyzing colored Petri nets, *Proc. of the 24th International Conference on the Application and Theory of Petri Nets*, Eindhoven, The Netherlands, June 2003.