

DRBD: DYNAMIC RELIABILITY BLOCK DIAGRAMS FOR SYSTEM RELIABILITY MODELLING

Haiping Xu^{*}, Liudong Xing^{**} and Ryan Robidoux^{*}

^{*}Computer and Information Science Department

University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA

E-mail: {hxu, u_rrobidoux}@umassd.edu

^{**}Electrical and Computer Engineering Department

University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA

E-mail: lxing@umassd.edu

ABSTRACT

With the rapid advances of computer-based technology in mission-critical domains such as aerospace, military, and power industries, critical systems exhibit more complex, dependent, and dynamic behaviors. Such dynamic system behaviors cannot be fully captured by existing reliability modelling tools. In this paper, we introduce a new reliability modelling tool, called dynamic reliability block diagrams (DRBD), to model dynamic relationships between system components. Due to the complexity of DRBD models that involve dynamic conceptual modelling constructs, such as a state dependency (SDEP) block, design errors, which are subtle and difficult to detect, can be easily introduced during the modelling process. In order to formally verify and validate the correctness of a DRBD model, we propose a Petri net based approach by converting DRBD constructs into colored Petri nets (CPN). We use a case study to illustrate how to convert a DRBD model into colored Petri nets, and how to use an existing Petri net tool to analyze and verify dynamic system behavioral properties. Our case study and experimental results show that DRBD models are a powerful tool for system reliability modelling, and our proposed verification approach can effectively ensure the correct design of DRBD models for complex and large-scale computer-based systems.

Key Words: Reliability modelling, dynamic reliability block diagram (DRBD), colored Petri net (CPN), formal verification, model checking

1. Introduction

With the rapid advances in computer-based technology, system reliability becomes an issue of increasing practical concern and research attention. Mission-critical computer-based systems, such as those used in aerospace, military, and power industries, exhibit more and more complex, dependent, and dynamic behaviors. For example, due to a state dependency existing among components, where the failure of one component can cause other components to become inaccessible, system components do not necessarily fail independently. Failure to model state dependencies accurately usually 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 state dependency relationships among components. As a special type of state dependency, 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 a sleeping mode). Among the existing reliability modelling 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. Load sharing is another major dynamic behavior of mission-critical systems. 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 and DFT model do not consider the load sharing behavior [2]. The BlockSim tool supports the load sharing configuration, but it only considers the increased load on the operating components due to the failure of some load sharing components; it cannot model the practical case in which a component in a load sharing configuration is put into the sleeping mode [3].

The reliability block diagram (RBD) model has been widely used as one of the most practical reliability modelling tools due to its simplicity [4, 5]. 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 [4]. 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 understand for engineers who design and test systems, and managers who make decisions on system configuration. With knowledge of the system, design engineers can readily construct and modify the corresponding RBD model, and communicate with people from different disciplines. However, similar to 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 advantages of using RBD models and the inadequacies of existing modelling tools to model dependencies and dynamic relationships among components in a large and complex system, we introduce a new modelling approach called dynamic reliability block diagrams (DRBD). The DRBD model extends the traditional RBD model by fully considering various dependencies and system dynamics. Some key DRBD modelling constructs, such as SDEP (state dependency) block and SPARE (spare part) block, has been formally specified using the Object-Z formal specification language in our previous work [6], which provides precise definitions for DRBD model behaviors. To ensure a correct design of a DRBD model that accurately represents the actual system in terms of its reliability behaviors, we need to formally verify behavioral properties of DRBD models. This requires a conversion of the DRBD model to a formalism, e.g., the Petri net formalism [7, 8], which is supported by effective analysis and verification tools.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 introduces the key DRBD constructs. Section 4 describes how to convert DRBD constructs into colored Petri nets. Section 5 provides a case study to illustrate how to develop DRBD models and how to formally verify the correctness of a DRBD model using state space analysis and model checking techniques. Finally, Section 6 gives the conclusions and future work.

2. Related Work

Previous work related to our research includes work on modelling 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 modelling

sequential and sparing behaviors [2]. The DFT approach offers limited capability to model dependency relationships among components, and it cannot accurately and fully model and analyze large systems subject to complex dynamics and dependencies. Furthermore, the DFT approach cannot manage problems caused by concurrency among dependencies. The ReliaSoft's BlockSim tool incorporates a standby container into the traditional RBD model for modelling the standby redundancy [3], but only cold spares are considered in this tool. Reference [9] recently introduced a set of DRBD constructs as an extension to the RBD models. The DRBD constructs are used to model dynamic dependency relationships among components; however, in comparison 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 derived from our previous work [6]. Our proposed DRBD constructs are based on simple notations; yet they are very powerful in modelling dynamic system reliability behaviors.

Very little work has been done on formal specification and verification of reliability models. 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 modelling the dynamic behaviors of gates are missing. Furthermore, no solutions are provided for verification of DFT models to ensure a correct design. Xu and Xing used the Object-Z formalism to specify both the state space and operations of a DRBD construct as a class schema, and discussed about formal verification techniques for DRBD models [6]. Additional related work to our proposed approach includes converting fault trees (FT) into generalized stochastic Petri nets (GSPN) to support dependability analysis [12]. The aim of the GSPN approach is to exploit the modelling and decision power of GSPN for both qualitative and quantitative analysis of a 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 modelling power to PDP, and it is more powerful than deterministic and stochastic Petri nets. Although the above approaches used Petri net formalism to model and analyze system reliability, unlike our approach, dynamic system reliability properties, such as state-based dependency, were not concerned. Furthermore, our approach has a major difference from their approaches: instead of providing quantitative analysis of system reliability using Petri nets, we use colored Petri nets for verification of the correctness of a reliability model (i.e., a DRBD model). Such verification is necessary because when dynamic reliability properties are involved, a DRBD model of a large and complex system becomes very complicated. 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

3.1 A Motivating Example

Consider a cluster of sensor nodes in a clustered wireless sensor network system as shown in Fig. 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 Fig. 1) that is activated when the primary cluster head fails. All the sensor nodes, except the cluster heads, within the cluster are divided into two mutually exclusive subsets ($S1$, $S2$). We assume each subset can provide desired sensing coverage. Initially, sensor nodes in $S1$ are operational; while sensor nodes in $S2$ are in a sleeping mode. To preserve the limited energy of sensor nodes, the duty cycle of sensor nodes will be adjusted. At certain point of time, sensor nodes in $S1$ will be put into a sleeping mode, and sensor nodes in $S2$ will be activated to maintain the desired sensing coverage. The entire cluster is considered to be operational if at least one of the two subsets of sensor nodes is operational and one cluster head is functioning. One of the major tasks in development of a reliability model for the above example is to model the state dependency between difference components, for example, the `Deactivation -> Activation` state dependency between node set $S1$ to node set $S2$. However, most of the existing reliability modelling tools (e.g., RBD) cannot capture the state dependency between components. Other tools, such as DFT, may support modelling a functional dependency, where the failure of a component causes some other dependent components to become inaccessible or unusable; they cannot capture the `Deactivation -> Activation` state dependency among components. In order to properly model various state dependency as well as other types of dependency relationships, e.g., sparing relationship and load sharing relationship, we propose a set of new dynamic reliability block diagram (DRBD) constructs as an extension to the existing RBD modelling tool in the following section.

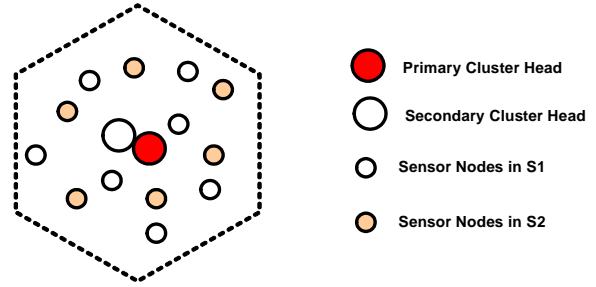


Figure 1. A clustered wireless sensor network system

3.2 DRBD Constructs

To model the state dependency between the two subsets of sensor nodes ($S1$, $S2$), we define a new DRBD controller component called SDEP (state dependency) block. The SDEP block can be used to model the state dependency relationship between $S1$ and $S2$, where the deactivation (or sleeping) of one subset of sensor nodes leads to the activation (or wake-up) of the other subset of sensor nodes. Fig. 2 (a) illustrates the general structure of this block, where A stands for an activation event occurred on a

component that leads to an *Active* state of that component, *D* stands for a deactivation event occurred on a component that leads to a *Standby* state of that component, and *F* stands for a failure event occurred on a component that leads to a *Failed* state of that component. The occurrence of a trigger event (*A*, *D* or *F*) will force all dependent events (*A*, *D*, or *F*) to happen. In other words, our proposed SDEP block can be used to model nine types of dependency relationships among system components, namely (*A*, *A*), (*A*, *D*), (*A*, *F*), (*D*, *A*), (*D*, *D*), (*D*, *F*), (*F*, *A*), (*F*, *D*), and (*F*, *F*). Since the DFT approach can only be used to model a (*F*, *D*) state dependency, DRBD is more powerful in modelling state-based dependent behaviors.

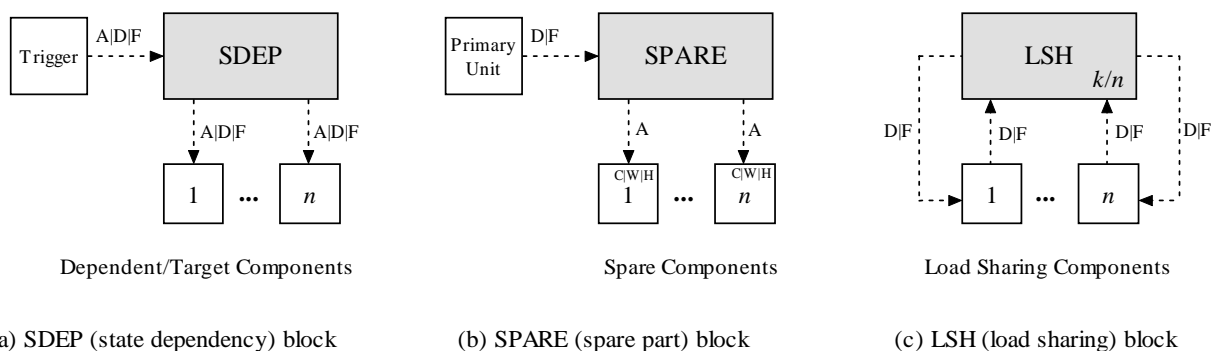


Figure 2. DRBD controller component blocks

To model the cold standby sparing cluster head subsystem, we define a new DRBD controller component called SPARE (spare part) block. Fig. 2 (b) illustrates the general structure of this block, where *C|W|H* stands for *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 the first spare component; the deactivation or failure of the first spare component will lead to the activation of the second spare component, and so on. All the spare units could be in *cold*, *warm*, or *hot* standby state, and must be used in the order from 1 to *n*.

Fig. 3 illustrates the DRBD model of the wireless sensor network system as described in Fig. 1. The model is developed using SDEP and SPARE blocks. The components labeled *S1* and *S2* represent the sets of sensor nodes in series structures, which constitute the corresponding subsets or sensor nodes. From the figure, we can see that the failure of the primary cluster head will automatically activate the secondary cluster head; while the deactivation of either *S1* or *S2* will automatically activate the other one. Besides the capability of modelling the state

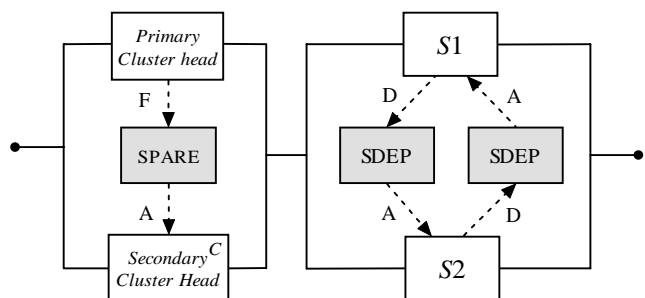


Figure 3. The DRBD model of the example system

dependencies and the various sparing behaviors, more new DRBD blocks and concepts have been proposed to model other dynamic relationships. An example of such new DRBD blocks is the LSH (load sharing) block as illustrated in Fig. 2 (c). As shown in the figure, a LSH block annotated with “ k/n ” refers to a load sharing controller component with n load sharing components, among which at least k components must be active or functioning. If currently there are exactly k active components, and one of them is deactivated or fails, all other active components will be deactivated, and thus enter *Standby* states. However, it is possible that when one of the k active components fails, some other active components may also fail due to overloading if they are not deactivated timely. These possible results are represented by the label “ $D|F$ ” in Fig. 2 (c). Other important DRBD constructs include SEQ (sequence dependency) block and PAND (priority AND) block. Sequence dependence enforces the occurrence order of state change events. For example, a SEQ block connecting two components $C1$ and $C2$ with a (F, F) sequence dependency can be used to specify that $C1$ must fail before $C2$ fails; thus, it precludes the case that $C2$ fails before $C1$ fails. Similarly, a priority AND or PAND block can be used to detect the occurrence order of certain events. A classic example of using a PAND controller component is a fault-tolerant system that consists of a primary component ($C1$) and a standby spare ($C2$) connected to a switch controller ($C3$) [14]. If the switch controller fails after the primary component fails, the standby component can be successfully switched into an *Active* state; thus the system can continue to operate. Otherwise, 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. In a DRBD model with a PAND block that connects $C1$, $C2$ and $C3$, the PAND block can automatically detect the failure order of $C1$ and $C3$; if it detects that $C3$ fails first, it will prohibit the activation of $C2$ when $C1$ fails. Note that both SEQ block and PAND block are defined as DRBD constructs in our proposed approach; however, due to space limitation, in this paper, we only illustrate those controller blocks that are used in our examples.

4. Conversion of DRBD Constructs to Colored Petri Nets

The introduction of new DRBD constructs as an extension to the RBD model can greatly enhance the modelling power for system reliability behaviors. 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 develop a standard RBD model, because it only contains a few static modelling constructs. However, when we design a DRBD model that involves new dynamic conceptual modelling constructs, engineers are more likely to bring design errors into the model due to the complexity of the newly introduced dynamic modelling

constructs, e.g., a SDEP block. Such design errors could be very subtle and difficult to detect when the model is not trivial, and it may result in an incorrect reliability model that leads to inaccurate results when the model is evaluated. Traditional simulation approaches to model testing is not suitable for verifying 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 a DRBD 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 [15] to specify the following system property of a computer system, “if component *A* fails, component *B* and *C* will also fail, which will lead to the failure of the whole system *S*.” The temporal formula in LTL (Linear Temporal Logic) can be written as $\Box (\neg A \rightarrow (\neg B \wedge \neg C) \wedge \Diamond \neg S)$, where the box \Box and the diamond \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 [16], where a “true” result indicates that the reliability model developed for system *S* does have this property; while a “false” result indicates 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.

Although the semantics of DRBD components and constructs can be formally defined in Object-Z as we did in our previous work [6], 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. 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 the advantages of being user friendly based on its graphical notations, and the powerful, but intuitive rules for defining structure and dynamic behaviors [7]. Petri nets provide a graphically defined intuitive way to model conditions, events, and their relationships, as well as essential characteristics like nondeterminism and concurrency. In a colored Petri net model, there are places and colored tokens that represent conditions, and transitions that represent events. A colored token, residing in a place, has a value, which can be tested against certain guards associated with transitions. When some specified guards become true, events can occur – this is depicted in the net model by the “firing” of transitions, which changes the distribution of tokens in the net, and also modifies the system state. So, a Petri net provides an executable model that directly defines the concept of a system’s state space. Based on our significant experience with colored Petri nets for many years, the Petri net formalism can achieve an effective balance between theoretical concepts and practical modelling techniques.

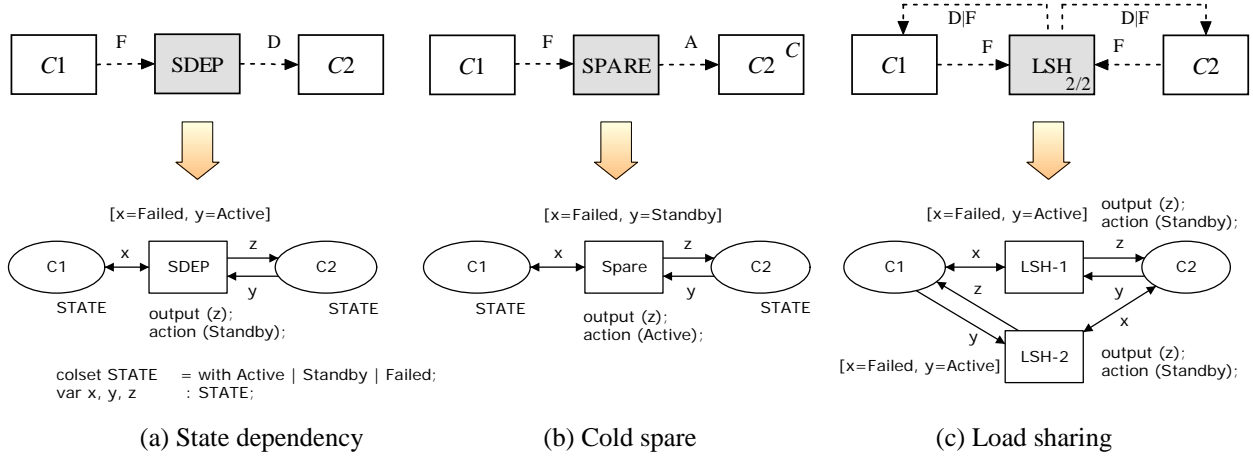


Figure 4. Conversion of DRBD constructs into colored Petri nets (CPN)

Fig. 4 demonstrates how to convert some key DRBD constructs into colored Petri nets. In Fig. 4 (a), component $C1$ and $C2$ are connected by a SDEP block with a (F, D) state dependency, i.e., the failure of $C1$ will lead to the deactivation of $C2$. In its corresponding colored Petri net, the two components are represented by two places $C1$ and $C2$, where the place types are defined by the color set `STATE` with three different colors, namely “Active”, “Standby” and “Failed”. When the transition guard $[x=Failed, y=Active]$ evaluates to true, i.e., place $C1$ contains a “Failed” token and place $C2$ contains an “Active” token, the transition SDEP becomes enabled and can fire. The firing of the transition will remove the tokens in place $C1$ and $C2$, and return a “Failed” token and a “Standby” token to place $C1$ and $C2$, respectively. Note that place $C2$ now contains a “Standby” token, which indicates that $C2$ enters a *Standby* state due to the failure of component $C1$. Similarly, Fig. 4 (b) illustrates how to convert a SPARE block into a colored Petri net. The colored Petri net corresponding to a SPARE block shows that when $C1$ becomes failed, and $C2$ is in a *Standby* state, $C2$ will be activated. In Fig. 4 (c), the LSH block annotated with “2/2” indicates that the LSH block is functioning only when both of the load sharing components $C1$ and $C2$ are functioning. When either of the components fails, the other component will be deactivated accordingly, and the whole LSH block becomes no longer functioning. In its corresponding colored Petri net, we use two transitions LSH-1 and LSH-2 to model the load sharing behaviors. When component $C1$ fails and $C2$ is in an *Active* state, the transition LSH-1 may fire, and its firing will force $C2$ to enter a *Standby* state. Similarly, when $C2$ fails and $C1$ is in an *Active* state, the transition LSH-2 may fire, and its firing will force $C1$ to enter a *Standby* state. Note that both the transition LSH-1 and LSH-2 can fire only when their associated transition guards evaluate to true. Thus, if component $C1$ and $C2$ fail at the same time, no transition (LSH-1 or LSH-2) can fire.

5. Case Study: Development and Verification of DRBD Models

5.1 DRBD Models of Load Sharing Servers with Routers

Load sharing across multiple servers would enhance responsiveness and scale well with session load; while a single server might not be able to cope with increasing demand for multiple sessions simultaneously. Now consider two load sharing server computers connected to a router. When a client attempts to access a server through the router, the router will automatically select a node in the server pool based on a load sharing algorithm, and redirect the request to that node.

Fig. 5 (a) shows the DRBD model of the above system that consists of two load sharing servers with a router. As shown in the figure, a router (component $R1$) is connected to two server computers (component $C1$ and $C2$). The two computers are connected in a parallel structure and controlled by a LSH block. The annotation “2/2” indicates that the parallel structure is functioning only when both $C1$ and $C2$ are in their *Active* states. Since clients can only access the server computers through the router, an (F, D) state dependency exists from component $R1$ to both $C1$ and $C2$, i.e., when the router fails, the computers will be deactivated for their network connections, and enter *Standby* states. Note that to simplify our conversion process, in the DRBD model shown in Fig. 5 (a), we used two SDEP blocks, which only involve one dependent/target component; however, the two SDEP blocks are equivalent to a single SDEP block with two dependent/target components $C1$ and $C2$, which follows the general form of a SDEP block defined in Fig. 2 (a).

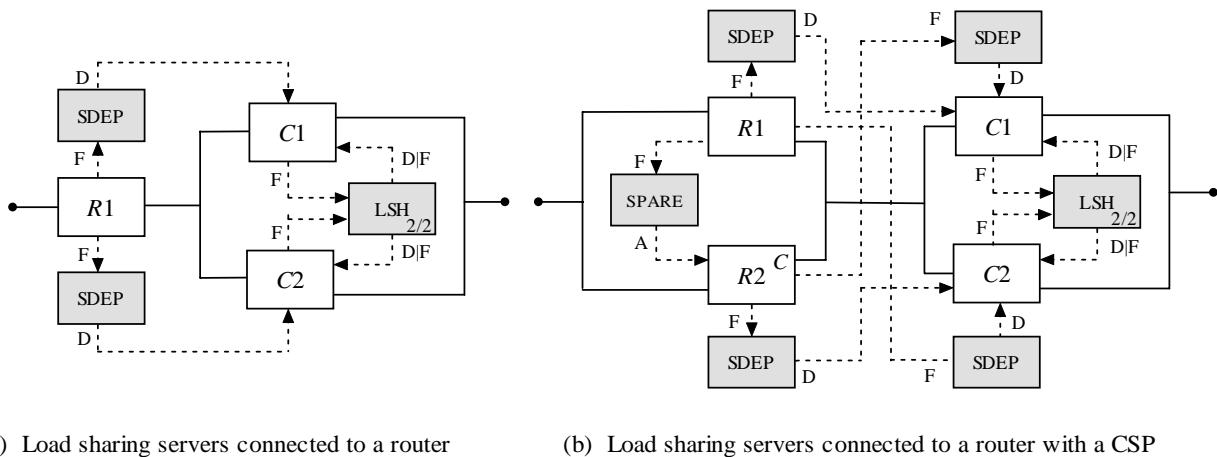


Figure 5. DRBD models of load sharing servers with router(s)

In order to make the system more reliable, we introduce a cold spare (CSP) for the router, which is represented by component $R2$ annotated with letter “C” in Fig. 5 (b). The DRBD model in Fig. 5 (b) describes when component $R1$ fails, component $R2$ will automatically be activated and continues to provide network access for component $C1$ and $C2$. When component $R2$ also fails, component $C1$ and $C2$ will be deactivated, and thus become inaccessible.

5.2 Conversion of DRBD Model to Colored Petri Nets

The DRBD model in Fig. 5 (b) can be converted into a colored Petri net as shown in Fig. 6. The four components in Fig. 5 (b) are modeled by the four places $R1$, $R2$, $C1$ and $C2$, and each place can contain a colored token with one of the three different colors, i.e., “Active”, “Standby” and “Failed”. An active component may fail when its corresponding $Fail_Ri$ or $Fail_Ci$ ($i = 1..2$) 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 four component places $R1$, $R2$, $C1$ and $C2$ contain “Active”, “Standby”, “Active” and “Active” colored token, respectively. According to the DRBD model in Fig. 5 (b), when either $R1$ or $R2$ is active, the parallel structure containing $R1$ and $R2$ is functioning. This is represented by firing either

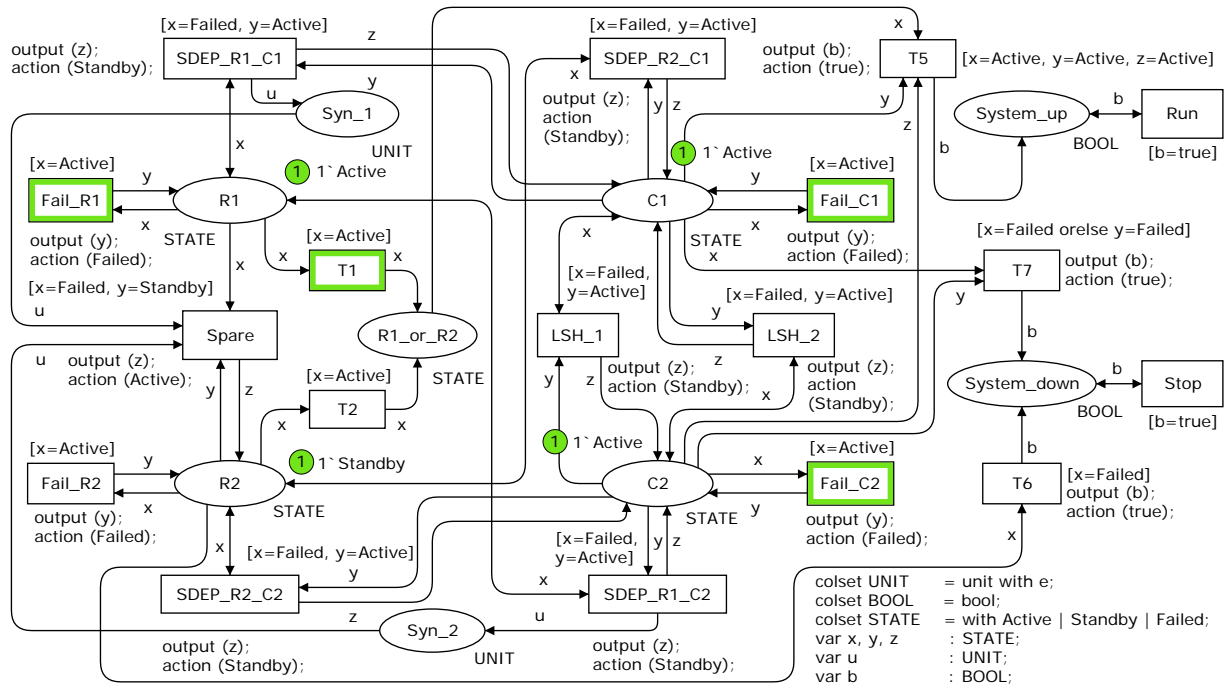


Figure 6. The colored Petri net model converted from the DRBD model in Fig. 5

transition $T1$ or $T2$ when either the place $R1$ or $R2$ contains an “Active” token, thus an “Active” token can be deposited into place $R1_or_R2$. When both component $C1$ and $C2$ are also active, the transition $T5$ may fire, and a Boolean token “true” is deposited into place $System_up$, which enables the Run transition. This scenario shows that when component $R1$ (or $R2$) and component $C1$ and $C2$ are functioning, the system must be functioning. On the other hand, when component $R2$ fails, the transition $T6$ may fire, and the firing of transition $T6$ will deposit a Boolean token “true” into place $System_down$. Similarly, when either component $C1$ or $C2$ fails (denoted by guard $[x=Failed \text{ or } y=Failed]$ for transition $T7$), the transition $T7$ may fire and deposit a Boolean token “true” into place $System_down$. A Boolean token “true” in place $System_down$ enables the $Stop$ transition, which indicates that the system cannot be functioning. Finally, if component $R1$ fails, it will activate component $R2$ due to the spare part redundancy; thus, it should not lead to the failure of the whole system.

The next step to convert the DRBD model in Fig. 5 (b) into a colored Petri net model is to convert the DRBD controller blocks into colored Petri nets according to the conversion methods described in Fig. 4. As shown in Fig. 6, the four transitions $SDEP_Ri_Cj$ ($i, j = 1..2$) represent the four SDEP blocks in Fig. 5 (b), which define the state dependency between a router and a server computer. The $Spare$ transition corresponds to the SPARE block in Fig. 5 (b), which activates component $R2$ when $R1$ fails. The transitions LSH_1 and LSH_2 are used to model the LSH block for the two load sharing components $C1$ and $C2$, which deactivates the other component when one of them fails. In addition, there are two synchronization places Syn_1 and Syn_2 , which are used to synchronize the transition $SDEP_R1_C1$ and $Spare$, and $SDEP_R1_C2$ and $Spare$, respectively. When transition $SDEP_R1_C1$ ($SDEP_R1_C2$) fires, a unit token is deposited into the synchronization place Syn_1 (Syn_2) to ensure that the firing of transition $SDEP_R1_C1$ ($SDEP_R1_C2$) precedes the firing of transition $Spare$, so the transition $SDEP_R1_C1$ ($SDEP_R1_C2$) will not accidentally become disabled when the “Failed” token in place $R1$ ($R2$) is removed due to the firing of transition $Spare$. When the transition $Spare$ fires, it deposits an “Active” token into place $R2$ in order to activate the spare part. This should lead to the continuous functioning of the whole system.

5.3 State Space Analysis and Model Checking of Colored Petri Nets

We now use an existing Petri net tool, called CPN Tools [17], to analyze our colored Petri net model. CPN Tools is a program that supports editing, simulating, and analyzing colored Petri nets, which include a state space analysis engine that can generate a full or partial state space, and produce a standard state space report containing information such as boundedness, liveness, and deadlock-freeness properties [18].

Table 1. Results from state space analysis tool

Result-1	Result-2	Result-3
Statistics	DeadMarking(32)	Statistics
-----	-----	-----
State Space	val it = true : bool	State Space
Nodes: 33		Nodes: 67
Arcs: 69	print(NodeDescriptor 32)	Arcs: 162
Secs: 0	-----	Secs: 0
Status: Full	32:	Status: Full
	C1 1: 1`Standby	
	C2 1: 1`Standby	
Scg Graph	R1 1: empty	Scg Graph
Nodes: 33	R2 1: empty	Nodes: 67
Arcs: 62	R1_or_R2 1: 1`Active	Arcs: 141
Secs: 0	Syn_1 1: empty	Secs: 0
	Syn_2 1: empty	
Liveness Properties	System_down 1: empty	Liveness Properties
-----	System_up 1: empty	-----
Dead Markings [32]	val it = () : unit	Dead Markings
		None
Dead Transition Instances	Reachable'(1, 32)	Dead Transition Instances
Router'SDEP_R2_C1 1	-----	None
Router'SDEP_R2_C2 1	A path from node 1 to 32:	
	[1, 3, 11, 25, 30, 32]	
Live Transition Instances	val it = true : bool	Live Transition Instances
None		None

When running the analysis tool in CPN Tools for our colored Petri net model, we get Result-1 as shown in Table 1. The result shows that there is a dead marking (state 32) in the reachability graph of the Petri net model, which can also be verified by executing the command `DeadMarking(32)` as shown in Table 1, Result-2. By executing the command `print(NodeDescriptor 32)`, it prints out the marking for state 32 as shown in Table 1, Result-2. From the result, we can see that in the dead marking S_{32} , both the places *System_up* and *System_down* contains no token (i.e., empty). This implies that the Petri net model must contain a deadlock, where none of the transitions *Run* and *Stop* is enabled. By executing the command `Reachable'(1, 32)`, we find a path from node 1 to node 32, i.e., [1, 3, 11, 25, 30, 32]. When we further tracing the dead marking state using the CPN Tools, we can calculate the firing sequence that leads to the dead marking, which is: $S_1, Fail_{R1}, S_3, SDEP_{R1_C1}, S_{11}, SDEP_{R1_C2}, S_{25}, Spare, S_{30}, T_2, S_{32}$. From Fig. 6, it is easy to see that the dead marking is due to the firing of transition *SDEP_R1_C1* (*SDEP_R1_C2*) that deposits a “Standby” token in place *C1* (*C2*) when component *R1* fails. When component *R2* is activated, component *C1* and *C2* should also be activated accordingly. However, such state dependency from component *R2* to *C1* (*C2*) is not presented in the DRBD model in Fig. 5 (b). This design error can be fixed by adding an (A, A) state dependency from *R2* to *C1* and *C2* in Fig. 5 (b). Accordingly, we need to revise the colored Petri net model in Fig. 6 as follows: (1) add a new transition *SDEP_R2_C12* with places *R2*, *C1* and *C2* as both the input places and output places; (2) add a new synchronization place *Syn_3* with *SDEP_R2_C12* as input transition and *T2* as output transition; (3) set

the guard of transition *SDEP_R2_C12* such that *R2* contains an “Active” token and both *C1* and *C2* contain a “Standby” token, i.e., $[x=Active, y=Standby, z=Standby]$; and (4) set the output of transition *SDEP_R2_C12* as “Active” tokens deposited into both *C1* and *C2* place, i.e., output (s, t) ; action $(Active, Active)$. We analyze the revised colored Petri net again using the CPN Tools, and now we get the results as shown in Table 1, Result-3. 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.

Furthermore, we can use model checking technique to verify properties of our colored Petri net models. CPN Tools facilitate analysis of state spaces by means of CTL-like temporal logic, called ASK-CTL, for formulating queries about states, as well as queries about state changes (e.g., the occurrence of certain transitions) [19]. For example, the *Formula_1* listed in Table 2, which is written in ML language, defines a CTL-like temporal formula $EXIST_UNTIL(TT, NOT(MODAL(TT)))$. In this formula, *TT* represents the constant “true” value, and the operator $MODAL(A)$, as a state formula, is true if starting from the current state, there exists an immediate transition from which the argument *A* is true. Since in the above formula, the argument *A* is constant true, $NOT(MODAL(TT))$ specifies a state that has no immediate transition that can fire, i.e., a deadlock marking. The operator $EXIST_UNTIL(A_1, A_2)$ is true if there exists a path, starting from the current state, A_1 is true for each state along the path until A_2 becomes true. Thus, the complete formula $EXIST_UNTIL(TT, NOT(MODAL(TT)))$ specifies whether there exists a path that involves a deadlock marking. In Table 2, the columns denoted as “After Rev” and “Before Rev” present the model checking results for the Petri net models after and before revision, respectively. From the model checking results for *Formula_1*, we can see that the original model (before revision) contains a deadlock marking; while the revised one has no deadlock marking.

Table 2. Model checking results for the Petri net models before and after revision

Formulas	ASK-CTL in ML	After Rev	Before Rev
Formula_1	<pre>val myASKCTLformula = EXIST_UNTIL(TT, NOT(MODAL(TT))); eval_node myASKCTLformula InitNode;</pre>	false	true
Functions	<pre>fun R1_Failed n = (Mark.R1 1 n = 1`Failed); fun R2_Failed n = (Mark.R2 1 n = 1`Failed); fun SystemFailed n = (Mark.System_down 1 n = 1`true);</pre>	-	-
Formula_2	<pre>val isFailed = FORALL_UNTIL(TT, NF("", SystemFailed)); val system = OR(NOT(NF("", R2_Failed)), isFailed); val myASKCTLformula = INV(system); eval_node myASKCTLformula InitNode</pre>	true	true
Formula_3	<pre>val isFailed = FORALL_UNTIL(TT, NF("", SystemFailed)); val system = OR(NOT(NF("", R1_Failed)), isFailed); val myASKCTLformula = INV(system); eval_node myASKCTLformula InitNode;</pre>	false	true

In Table 2, we further define three functions: *R1_Failed*, *R2_Failed* and *SystemFailed*. The function *R1_Failed* (or *R2_Failed*) returns *true* if place *R1* (or *R2*) contains a colored token “Failed”, which indicates that the corresponding component fails. Similarly, the function *SystemFailed* returns *true* when place *System_down* contains one colored token “true”, which indicates that system is not functioning. In *Formula_2* from Table 2, we define a temporal formula $isFailed = \text{FORALL_UNTIL}(TT, \text{NF}("", \text{SystemFailed}))$, where the operator $\text{FORALL_UNTIL}(A_1, A_2)$ is true if from the current state, 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 specifies that from the current state, whether the system will eventually become *not* functioning. We now consider the relationship between the failure of component *R2* and the failure of the whole system. At any time (state), the system model should satisfy the following property: $(R2_Failed \rightarrow \langle \rangle \text{SystemFailed})$, which is equivalent to $(\neg R2_Failed \vee \langle \rangle \text{SysFailed})$. This property can be specified by the temporal formula $\text{system} = \text{OR}(\text{NOT}(\text{NF}("", R2_Failed)), isFailed)$. Since this property should be satisfied at any time, the formula $\text{INV}(\text{system})$ must be true for our Petri net models, where $\text{INV}(\text{system})$ is true if from the current state, the argument *system* is true for all reachable states. The model checking results show that both Petri net models (before and after revision) satisfy this property. Similarly, we may also verify the relationship between the failure of component *R1* and the failure of the whole system. Since the failure of component *R1* will activate component *R2* due to the spare part redundancy; thus, it should not lead to the failure of the whole system. The model checking result of *Formula_3* for our revised Petri net model is *false*, which verifies that the revised model does not satisfy the property (if written in LTL): $[\](R1_Failed \rightarrow \langle \rangle \text{SystemFailed})$, i.e., the failure of *R1* will not always lead to the failure of the whole system. On the other hand, the model checking result of *Formula_3* for the original Petri net model (with deadlock marking) is *true*, which indicates that the above property is satisfied. In other words, it is true that the failure of component *R1* will always lead to the failure of the whole system. However, we notice that along the path that leads to the deadlock marking, it is impossible to deposit a colored token “true” into place *System_down*; thus, it seems that the model checking result should also be *false*. The reason it returns a *true* result is because the path that leads to the deadlock marking is a finite sequence, and in this case, the CPN Tools will evaluate the temporal formula $\text{FORALL_UNTIL}(TT, \text{NF}("", \text{SystemFailed}))$ to true, although *SystemFailed* does not eventually become true along that path.

6. Conclusions and Future Work

Existing system reliability modelling approaches cannot fully capture dynamic relationships between components, such as state dependency and redundancy. In this paper, we propose a new modelling

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 reliability modelling tool for complex and large computer-based systems. The methods we proposed for conversion of DRBD constructs to colored Petri nets provide a potential solution for automated conversion of a DRBD model to colored Petri nets, and automated verification of DRBD models, which is demonstrated in our recent work [20]. The case study illustrates how a DRBD model of a computer-based system can be developed, and how formal verification approach can be used to ensure a correct design of the DRBD model. In our future work, we will develop a software tool that can automatically translate DRBD models into colored Petri nets for formal verification. We also plan to develop efficient evaluation methods for DRBD models in order to analyze and predict system reliability performance. A comprehensive system reliability modelling tool that supports editing, formal verification, and evaluation of DRBD models for complex and large-scale systems is envisioned as our future, more ambitious research direction.

Acknowledgement

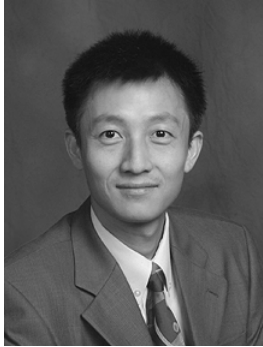
This material is based upon work partially supported by the Research Seed Initiative Fund (RSIF), College of Engineering, UMass Dartmouth. We thank all anonymous referees for the careful review of this paper and the suggestions for improvements they provided.

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, & 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] BlockSim, *System reliability analysis software using an RBD or fault tree approach*, ReliaSoft Corporation, <http://www.reliasoft.com/BlockSim/>, accessed on June 12, 2007.
- [4] M. Rausand & A. Høyland, *System reliability theory: models, statistical methods, and applications* (New York, USA, Wiley-Interscience, 2003).
- [5] W. Wang, J. M. Loman, R. G. Arno, P. Vassiliou, E. R. Furlong, & 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.
- [6] H. Xu & L. Xing, Formal semantics and verification of dynamic reliability block diagrams for system reliability modelling, In *Proc. of the 11th International Conference on Software Engineering and Applications (SEA 2007)*, November 19-21, 2007, Cambridge, Massachusetts, USA.

- [7] T. Murata, Petri nets: properties, analysis and applications, *Proc. of the IEEE*, Vol. 77, No. 4, April 1989, pp. 541-580.
- [8] K. Jensen, *Colored Petri nets: basic concepts, analysis methods and practical use, Volume 2, analysis methods* (Monographs in Theoretical Computer Science, Springer-Verlag, 1997).
- [9] S. Distefano & L. Xing, A new approach to modelling the system reliability: dynamic reliability block diagrams, *Proc. of the 52nd Annual Reliability & Maintainability Symposium (RAMS'06)*, Newport Beach, CA, January 2006, pp. 189-195.
- [10] D. Coppit, K. J. Sullivan, & 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 & 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 & H. Kumamoto, *Probabilistic Risk Assessment: Reliability Engineering, Design, and Analysis* (IEEE Press, 1992).
- [15] Z. Manna & A. Pnueli, *The temporal logic of reactive and concurrent systems - specification* (Springer-Verlag New York, Inc, 1992).
- [16] E. M. Clarke, O. Grumberg, & D. A. Peled, *Model Checking* (MIT Press, 2001).
- [17] Jensen, K., Kristensen, L. M., & Wells, L., Coloured Petri nets and CPN Tools for modelling and validation of concurrent systems, *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, 2006.
- [18] 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.
- [19] A. Cheng, S. Christensen, & K. H. Mortensen, Model Checking Coloured Petri Nets Exploiting Strongly Connected Components, In *Proc. of the International Workshop on Discrete Event Systems*, Edinburg, Scotland, UK, pages 169-177. August 1996.
- [20] R. Robidoux, *Automated verification of a computer system reliability model*, Master's Thesis, Computer and Information Science Department, University of Massachusetts Dartmouth, July 2007.

Biographies



Haiping Xu received the B.S. degree in Electrical Engineering from Zhejiang University, Hangzhou, China, in 1989, the M.S. degree in Computer Science from Wright State University, Dayton, OH, in 1996, and the Ph.D. degree in Computer Science from the University of Illinois at Chicago, IL, in 2003. Since 2003, he has been with the University of Massachusetts Dartmouth, where he is currently an Assistant Professor at the Computer and Information Science Department, and a Co-Director of the Concurrent Software Engineering Laboratory. His research interests include distributed software engineering, formal methods, Internet security, multi-agent systems, and service-oriented systems. He is a senior member of the IEEE Computer Society and a professional member of the ACM.



Liudong Xing received the B.E. degree in Computer Science from Zhengzhou University, China, in 1996, the M.S. degree and the Ph.D. degree in Electrical Engineering from the University of Virginia, in 2000 and 2002, respectively. Her major field of study is reliability engineering and fault-tolerant computing. She is currently an Associate Professor with the Electrical and Computer Engineering Department, University of Massachusetts Dartmouth. Her current research interests include dependable computing and networking, hardware and software reliability engineering, fault-intrusion tolerant computing, and wireless sensor networks.



Ryan Robidoux received the B.S. degree and the M.S. degree in Computer Science from the University of Massachusetts Dartmouth, MA, in 2005 and 2007, respectively. His major research interests include neural network, software engineering, formal methods, and web services. He is currently a lecturer in the Computer and Information Science Department at the University of Massachusetts Dartmouth, and a research associate and a software developer at the Kaput Center for Research and Innovation in Mathematics Education.