NASA/CR–2011-217090

Probabilistic Analysis of Distributed Fault-Tolerant Systems

Bruno Dutertre
Computer Science Laboratory, SRI International
Menlo Park, California

May 2011

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2011-217090

Probabilistic Analysis of Distributed Fault-Tolerant Systems

Bruno Dutertre
Computer Science Laboratory, SRI International
Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

May 2011

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

An approach is documented for analyzing probabilistic properties of fault-tolerant distributed systems using the PRISM model checker.

Contents

1 Introduction

A method for probabilistic analysis of fault-tolerant distributed systems using a probabilistic model checker is presented. This description includes several example analyses based on the Scalable Processor-Independent Design for Enhanced Reliability (SPIDER) architecture.

A long history exists in applying probabilistic models to estimate the reliability of redundant systems. Methods based on Markov chain analysis are prevalent in this context as they enable modeling of state-dependent stochastic phenomena, which is necessary to represent dynamic system features such as reconfiguration and component repair. The general principles behind these methods, and many examples of Markov (and semi-Markov) models of fault-tolerant systems, are presented by Butler and Johnson in [5].

Different software tools have been developed to solve the Markov or semi-Markov chains encountered when modeling fault-tolerant systems. Geist and Trivedi survey several tools available in the early 1990s and present the solution techniques that they employ [10]. All these tools attempt to address two fundamental issues. First, Markov models of fault-tolerant systems can have very large state spaces. This large size is caused both by the number of components in such systems and by the complex reconfiguration and diagnosis strategies fault tolerance often requires. Second, the transition rates encountered in the Markov models can vary widely in magnitude—since fault-arrival rates are slow but system reconfiguration is fast—which leads to difficulties for numerical algorithms. The Semi-Markov Unreliability Range Evaluator (SURE) [4] is dedicated to the analysis of fault-tolerant systems that exhibit low fault rates and fast reconfiguration. It relies on approximation theorems to give lower and upper bounds on system reliability. The Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [27] uses hierarchical modeling to mitigate the state-explosion problem. Simulation-based methods that rely on importance sampling have also been proposed [9, 30].

More recent tools such as Möbius [8] or the Symbolic Model checking Analyzer for Reliability and Timing (SMART) [6] are general frameworks for assessing reliability, performance, and other properties of complex computer systems. They employ sophisticated techniques for compactly representing the model state space and they use decomposition methods for reducing the analysis cost. These tools are generic and have been applied to a variety of computer systems and networks. Example applications to fault-tolerant systems include [3] and [31]. A more complete survey of recent modeling formalisms and analysis tools can be found in [28].

Typically, the probabilistic analysis of Markovian models focuses on computing transient or steady-state distributions of a Markov chain. Thus, the previous tools can typically answer questions of the form "what is the probability of reaching a set of states S?" or "what is the steady-state probability of being in state S?". Probabilistic model checkers are a related class of tools that provide a richer language for querying models, which allows one to consider not just reachability properties but also properties of execution paths in an underlying Markov model [17]. Properties are expressed in probabilistic temporal logics, most of which are extensions of the Probabilistic Computational Tree Logic (PCTL) or the Continuous Stochastic Logic (CSL) introduced in [12] and [2], respectively. As the name indicates, probabilistic model checking is an extension of traditional, nonprobabilistic model checking to probabilistic and, more generally, quantitative system properties. For example, a probabilistic model checker can be used to formally verify that a system model, usually given as a finite Markov chain or Markov decision process, satisfies properties such as "the failure probability is less than 10^{-k}" or "the probability that component A will fail before component B is more than p".

Several probabilistic model checkers are available, including PRISM [15], the Markov Reward Model Checker (MRMC) [16], the Erlangen-Twente Markov Chain Checker (ETMCC) [13], Ymer [34], and VESTA [29]. Among these, PRISM is recognized as the leading software tool. It has been continuously

developed and maintained since 2000, and it has been applied to a wide range of case studies. It can be downloaded for free at `http://www.prismmodelchecker.org/`. In this report, we investigate the use of PRISM to the analysis of high-reliability, fault-tolerant systems.

We give an overview of the PRISM modeling language and logic. We then present several PRISM models and analysis of a representative high-reliability system—namely, the SPIDER fault-tolerant architecture [21, 33].

2 PRISM Overview

PRISM is a tool for the formal modeling and verification of stochastic systems. The current release (i.e., PRISM 3.3.1) supports three types of probabilistic models: *discrete-time Markov chains (DTMCs)*, *continuous-time Markov chains (CTMCs)*, and *discrete-time Markov decision processes (MDPs)*. Support for *probabilistic timed automata (PTAs)* has been announced for the next release (i.e., PRISM 4.0) [18]. PTAs are an extension of MDPs with special real-valued variables called *clocks* that measure time progress. PTAs are then intended for modeling and analysis of real-time, probabilistic systems. A preliminary version of PRISM 4.0 is available as a beta release, but all the work and experiments described in this report were performed using PRISM 3.3.1.

2.1 Modules

A PRISM system is specified as the composition of one or more elementary systems called *modules*. Each module is defined by its state variables, which must all have a finite type, and by a set of guarded commands that describes state transitions and the associated probabilities.

Figure 1 shows a PRISM specification that defines a simple fault model. The specification starts with the keyword `ctmc` to indicate that the model is a continuous-time Markov chain. Alternatively, one can use the keywords `dtmc` or `mdp` for a discrete-time Markov chain or Markov decision process, respectively. The model then declares three parameters named `lambda1`, `lambda2`, and `lambda3` that define three transition rates. In this particular example, `lambda1` is intended to be the transient-fault rate; `lambda2` is the permanent-fault rate; and `lambda3` is the recovery rate from transient faults.

The module has a single state variable called `state` whose value is an integer in the set $\{0, 1, 2\}$. The intended interpretation is that 0 is the fault-free state of the system, 1 is the state after a transient fault, and 2 is the state after a permanent fault. Initially, the system is fault free: `state` is initialized to 0.

The rest of the module consists of three guarded commands, which define the transition rates of the underlying Markov chain. The first command is

```
[fault] (state = 0) -> lambda1: (state'=1) + lambda2: (state'=2);
```

It defines the transitions originating from state 0: with rate `lambda1` the system moves to state 1 and with rate `lambda2` the system enters state 2. This command is labeled by the identifier `fault`, which is used later in specifying rewards. The other two guarded commands in Figure 1 do not have a label. The transition rates such as `lambda` are optional. If they are not given in the guarded command, then PRISM will assume a default rate of 1 (as in the transition from state 2). The Markov chain corresponding to module `c1` of Figure 1 is depicted in Figure 2.

The other parts of the PRISM model in Figure 1 include the definition of a set of states, identified by the label `failed`, that corresponds to the faulty states of the system (i.e., when `state` is different from 0). Also, three *rewards* are attached to the model. In general, a reward is a numerical quantity that is assigned to particular states or transitions of the system. In this example, the reward `functional` is equal to 1 when `state=0` and 0 otherwise. The PRISM logic includes several constructs to express properties of these

```
ctmc

const double lambda1; // transient faults
const double lambda2; // permanent faults
const double lambda3; // recovery from transient faults

module c1
  state: [0..2] init 0;

  [fault] (state = 0) -> lambda1: (state'=1) + lambda2: (state'=2);
  []   (state = 1) -> lambda3: (state'=0);

  []   (state=2) -> (state'=2);
endmodule

label "failed" = (state=1) | (state=2);

// Rewards: 1 for state 0, 0 in all other states
rewards "functional"
  state = 0 : 1;
  state != 0: 0;
endrewards

// For computing expected times
rewards "total_time"
  true: 1;
endrewards

// For computing number of faults
rewards "num_failures"
  [fault] true: 1;
endrewards
```

Figure 1. Example PRISM Model

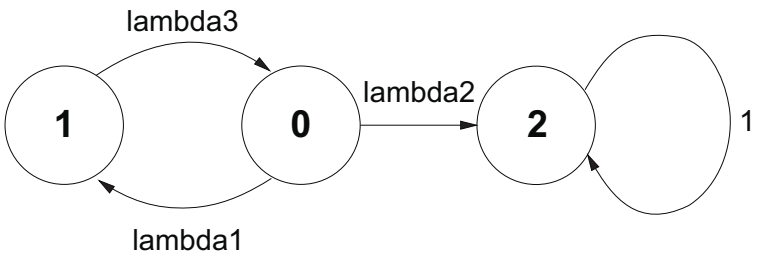

Figure 2. Markov Chain Corresponding to the PRISM Model in Figure 1

reward functions—for example, the expected cumulative reward in a bounded interval $[0, t]$. In the case of the `functional` reward, this is the average fraction of time during which the system is fault free. The reward function `total_time` can be used in a similar fashion—for example, to define the expected time to the first failure. As illustrated by `num_failures`, rewards can also be attached to state transitions. In this case, `num_failures` is equal to 1 every time the transition `fault` is taken, and to 0 on other transitions.

2.2 Property Specification

PRISM uses an extension of PCTL for expressing properties of Markov chains. The logic includes the traditional Computational Tree Logic (CTL) [7] operators that describe possible execution trees of a state-transition system and new operators that express probability constraints on these execution trees. The underlying semantics is then based on a probability space defined over the execution traces of a Markov chain [17, 25, 26].

Figure 3 shows example logical formulas related to the PRISM model defined previously and their informal interpretation. The first five examples illustrate a very common pattern of queries one makes about PRISM models. These queries are all of the form

```
P =? [ path-property ]
```

Such formulas ask for the probability that the Markov chain's execution path satisfies the given path property. The path properties are themselves written using the CTL operators `G` (always), `F` (eventually), and `U` (until). For example, the path property `F state=1` means "eventually the system reaches state 1". It is satisfied by all paths that pass through state 1, and the formula `P=? [ F state=1]` asks for the probability of this set of paths.

Formula `P=? [G<=10.0 !"failed"]` uses a bounded form of the CTL operator `G` to make a query about failure probability for a bounded mission time. It uses the label `failed` defined in Figure 1 to refer to the faulty states. Property `G<=10 !"failed"` is satisfied by all paths that spend at least 10 time units in the fault-free state—that is, all paths where the first fault occurs after time 10. Hence, formula `P=? [G<=10.0 !"failed]` asks for the probability that no fault occurs in the first 10 time units of the system behavior.

Rather than asking PRISM to compute probabilities of interest, one can also assert bounds on the probability of certain sets of paths and verify with PRISM whether these bounds actually hold. This is illustrated by the sixth property of Figure 3. The formula `P>0.9 [ G<=10.0 (state!=2) ]` is true in a state s of the Markov chain if the set of paths that start from s and do not reach state 2 in the first 10 time units has probability at least 0.9. Then property

```
"init" => P>0.9 [ G<=10.0 (state!=2) ]
```

holds for the system if all initial states satisfy the probability constraint.

```
// What's the probability of eventually experiencing a transient fault?
P=? [ F state=1 ]

// What's the probability of a eventually experiencing a permanent fault?
P=? [ F state=2 ]

// What's the probability that the first fault is transient?
P=? [ state=0 U state=1 ]

// What's the probability that the first fault is permanent?
P=? [ state=0 U state=2 ]

// What's the probability that the system doesn't fail in a 10 hour run?
P=? [ G<=10.0 !"failed" ]

// Is the probability of no permanent failure at least 0.9
// (in a 10 hour run)?
"init" => P>0.9 [ G<=10.0 (state!=2) ]

// If we run for 10 hours, what's the expected amount of functional time
R{"functional"} =? [ C<=10.0 ]

// What's the expected number of failures
R{"num_failures"} =? [ C<=10.0 ]

// What's the expected time to the first failure
R{"total_time"} =? [ F "failed" ]
```

Figure 3. Example PRISM Formulas for the Model in Figure 1

Property	Result
`P=? [ F state=1 ]`	0.90909
`P=? [ F state=2 ]`	1.00000
`P=? [ state=0 U state=1 ]`	0.90909
`P=? [ state=0 U state=2 ]`	0.09091
`P=? [ G<=10.0 !"failed" ]`	0.99989
`"init" => P>0.9 [ G<=10.0 (state!=2) ]`	true
`R{"functional"} =? [ C<=10.0 ]`	9.99994
`R{"num_failures"} =? [ C<=10.0 ]`	1.098×10^{-4}
`R{"total_time"} =? [ F "failed" ]`	90909.1

Parameters: `lambda1` $= 10^{-5}$, `lambda2` $= 10^{-6}$, `lambda3` $= 10$

Table 1. Model Checking Results for the Properties in Figure 3

Finally, Figure 3 shows how PRISM can express quantitative properties using reward structures. For example, formula `R{"functional"} =? [ C<= 10.0]` asks for the expected cumulated value of the reward function `functional` over the first 10 time units of a run. More examples of queries and formulas, and the exact semantics of the PRISM logical operators, are given in the PRISM manual [25].

2.3 The PRISM Tools

To answer the previous queries or check formulas, PRISM can be used either as a command-line tool or via a graphical user interface. For example, one can check the properties listed in Figure 3 by using the following command.

```
prism simple-fault-model.sm simple-continuous.pctl
    -const 'lambda1=1e-5,lambda2=1e-6,lambda3=10'
```

This invokes the `prism` model checker on the model defined in file `simple-fault-model.sm`, for the set of properties stored in file `simple-continuous.pctl`, and with specific values assigned to the model parameters `lambda1`, `lambda2`, and `lambda3`.

For each property it evaluates, the tool prints statistics and results, as follows

```
Model checking: P=? [ F state=1 ]
Model constants: lambda1=0.00001, lambda2=0.000001, lambda3=10
...
Starting iterations...

Jacobi: 2 iterations in 0.00 seconds (average 0.000000, setup 0.00)

Time for model checking: 0.0020 seconds.

Result (probability): 0.9090909090909091
```

This shows the probability of experiencing a transient failure for the Markov model of Figure 1. Table 1 summarizes the `prism` results for the properties of Figure 3.

PRISM also provides a graphical user interface (cf. Figure 4). This tool can be used for editing models, and specifying and verifying properties. More information on this tool and the `prism` command-line tool can be found in the PRISM manual [25].

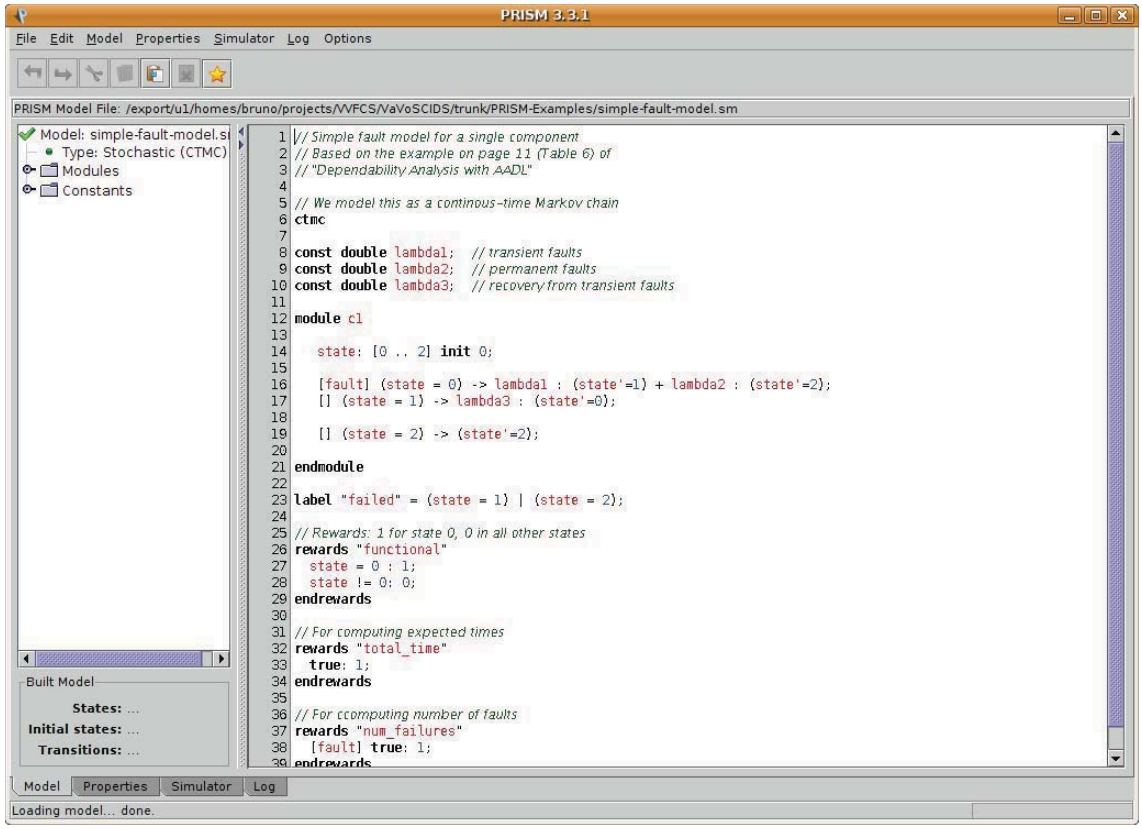

Figure 4. PRISM Graphical User Interface

2.4 Other Features of PRISM

In the example so far, we have considered only a PRISM model that consists of a single module. It is much more common to build PRISM systems by composing several modules. PRISM provides several module-combination operators, but the default parallel composition appears to be the most common: the modules perform transitions independently of each other, except that transitions having the same labels must be synchronized. This is a standard form of asynchronous product of labeled transition systems. The PRISM semantics document [26] explains how the transition probabilities are combined in such products.

Another useful feature is the ability to define identical modules by renaming. This allows one to define a module B as a copy of another module A, while optionally changing the names of variables, parameters, or transition labels.

In addition to building systems by combining modules, PRISM allows systems to include global state variables that can be read and written by all the modules. Some restrictions on the use of such variables ensure that only one module writes to a global variable at a time.

3 Probabilistic Analysis of SPIDER Using PRISM

We present applications of PRISM to the analysis of the SPIDER fault-tolerant distributed system. The standard reliability-estimation method for such a system starts with identifying a set of assumptions about the number and type of faulty components the system is designed to tolerate. Then, overall system reliability can be established in two steps: prove that the system actually remains functional under the given fault assumptions and estimate the probability that these assumptions hold.

We first describe how this technique can be supported by the PRISM tool, and then we investigate the application of PRISM to other analysis of SPIDER, including protocol properties.

3.1 SPIDER

SPIDER [22,33] is a distributed computer architecture that provides a communication service for integrated modular avionics. The system relies on hardware replication, Byzantine-fault-tolerant protocols, and diagnosis and reconfiguration to achieve very high reliability. The core of SPIDER is a Reliable Optical Bus (ROBUS) whose architecture is shown in Figure 5. ROBUS consists of N bus interface units (BIUs) and M redundancy management units (RMUs). Each BIU is connected to each RMU via a bidirectional communication link. Each BIU is also connected to a single processing element (PE). The goal of ROBUS is to provide a highly reliable communication medium to the PEs.

More precisely, ROBUS implements a fault-tolerant clock synchronization algorithm to ensure that all the fault-free BIUs and RMUs are synchronized. This enables the PEs to access the bus by following the time-division multiple access (TDMA) strategy. All components in the system agree and enforce a system-wide communication schedule that prevents bus collisions. The redundant RMUs and BIUs provide a fault-tolerant *consistent broadcast* service by means of a Byzantine fault-tolerant voting algorithm. In addition, BIUs and RMUs implement an online diagnosis protocol to detect and exclude faulty components. Other important protocols used by ROBUS include a fault-tolerant startup and a reintegration protocol that enables nodes that are subject to transient failures to rejoin the system. The ROBUS protocols are described in detail in [33] and [11]. Formal modeling and verification of these protocols are presented in a series of publications, including [21, 23, 24].

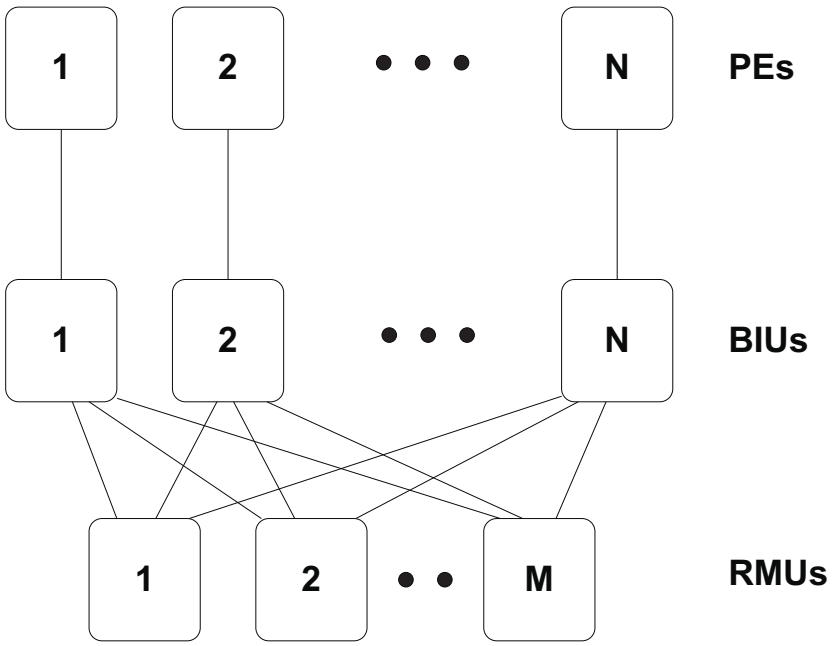

Figure 5. ROBUS Architecture

3.2 Maximal Fault Assumption

ROBUS provides correct service to the PEs under a specific set of assumptions about the number and severity of faulty components in the system. Following the hybrid fault model introduced in [32] (see also [23]), we assume that a BIU or RMU can be in one of the following four states:

- *Good:* A node is good if it is free of faults.
- *Benign:* A node is benign if it sends only messages that are manifestly wrong. This means that the recipients of such messages can accurately recognize that the message is incorrect. Since SPIDER components can detect that a message is missing, benign nodes includes fail-silent nodes.
- *Symmetric:* A node is symmetric if it sends the same possibly incorrect message to all recipients, and the recipients may not detect that the message is incorrect.
- *Asymmetric:* A node is asymmetric if it may send different, possibly incorrect, messages to each recipient. As previously, the recipients may not recognize that the message is incorrect.

More refined fault classifications are possible (e.g., [1]) but we follow this taxonomy since it is used in the majority of the publications on SPIDER.

Based on this taxonomy, we can partition the BIUs and the RMUs into four sets. Let GB, BB, SB, and AB denote the sets of good, benign, symmetric, and asymmetric BIUs, respectively. Similarly, let GR, BR, SR, and AR denote the sets of good, benign, symmetric, and asymmetric RMUs. Then, the SPIDER protocols provide correct services to the PEs provided the following constraints are satisfied [21]:

1. $|GB| > |AB| + |SB|$
2. $|GR| > |AR| + |SR|$
3. $|AR| = 0$ or $|AB| = 0$

Less formally, SPIDER can tolerate asymmetric components as long as they are all BIUs or all RMUs, and the system remains functional as long as there are enough good BIUs and RMUs to outvote the faulty components.

The three conditions above are SPIDER's *maximal fault assumption*. They describe the maximal number and types of faults that SPIDER has been proven to tolerate.

3.3 Reliability Models in PRISM

Our first reliability model of SPIDER follows the system architecture. It is given in the appendix (A.1). The particular instance of SPIDER used in this study consists of three BIUs and three RMUs (i.e., $N = M = 3$). We then developed a more general model in which the numbers N and M are parameters. This second model is also given in the appendix (A.2).

3.3.1 Baseline Model

Our baseline model is a continuous-time Markov chain that closely follows the SPIDER architecture. This model includes three BIUs and three RMUs, each of them represented as a separate PRISM module. The PRISM specification of BIU1 is described in Figure 6. The modeling follows the fault classification presented previously. It also distinguishes between permanent and transient faults.

The state of a BIU is encoded as two variables: A variable specifies whether the BIU is good, benign, symmetric, or asymmetric, and a Boolean variable records whether the current fault is permanent or transient. These two variables are named `biu1_state` and `biu1_permanent` in Figure 6. As indicated in the figure, the possible error states are represented by integers between 0 and 3.

To model fault occurrences, we use two sets of parameters:

- *Transition rates:* Two parameters define the fault-arrival rates for transient and permanent faults. They are called `trans_lambda` and `perm_lambda`, respectively. A third parameter is the rate of recovery from transient faults (called `repair_lambda`). All rates are assumed to be measured in number of faults per hour.
- *Probabilities:* Three parameters define the probability that a fault is benign, symmetric, or asymmetric. The first two probabilities are given by parameters `benign_ratio` and `symmetric_ratio` in PRISM, with the third probability defined as

  ```
  const double asymmetric_ratio = (1 - benign_ratio - symmetric_ratio);
  ```

  Obviously, the actual values of `benign_ratio` and `symmetric_ratio` must be between 0 and 1, and their sum must also be in the interval $[0, 1]$. PRISM cannot enforce these constraints, since it does not know the parameters' interpretation.[1]

  We have assumed that these probabilities are independent of whether the fault is transient or permanent, but it is easy to adjust the model to a different hypothesis.

The PRISM module then specifies in a straightforward fashion the transitions from fault free to one of the possible error states. For example, a benign, transient fault occurs with a rate equal to `trans_lambda * benign_ratio`. If a fault is transient, then the BIU returns to the good state with a rate equal to `repair_lambda`.

[1] PRISM does not currently support assertions on system parameters, so there is no easy way to encode these constraints in PRISM.

```
// Error states of each component
const int good = 0;
const int benign = 1;
const int symmetric = 2;
const int asymmetric = 3;

// Parameters: fault arrival rates for each component
const double perm_lambda;   // permanent faults   (e.g., 10^-6)
const double trans_lambda;  // transient faults   (e.g., 10^-5)
const double repair_lambda; // recovery from transient faults (e.g., 10)

// Fractions of benign/symmetric/asymmetric faults
const double benign_ratio;          // 0.5
const double symmetric_ratio;       // 0.3
const double asymmetric_ratio = (1 - benign_ratio - symmetric_ratio);

// Fault arrival rates for the BIUs
const double biu_perm_lambda = perm_lambda;
const double biu_trans_lambda = trans_lambda;
const double biu_repair_lambda = repair_lambda;

// Fractions
const double biu_benign = benign_ratio;
const double biu_symmetric = symmetric_ratio;
const double biu_asymmetric = asymmetric_ratio;

// Error model for BIU1
module biu1
  biu1_state: [0..3] init good;
  biu1_permanent: bool init false;

  [] biu1_state = good ->
       (biu_perm_lambda * biu_benign): (biu1_state'=benign) &
                                       (biu1_permanent'=true)
     + (biu_perm_lambda * biu_symmetric): (biu1_state'=symmetric) &
                                          (biu1_permanent'=true)
     + (biu_perm_lambda * biu_asymmetric): (biu1_state'=asymmetric) &
                                           (biu1_permanent'=true)

     + (biu_trans_lambda * biu_benign): (biu1_state'=benign) &
                                        (biu1_permanent'=false)
     + (biu_trans_lambda * biu_symmetric): (biu1_state'=symmetric) &
                                           (biu1_permanent'=false)
     + (biu_trans_lambda * biu_asymmetric): (biu1_state'=asymmetric) &
                                            (biu1_permanent'=false);

  // recovery from transient faults
  [] (!biu1_permanent & biu1_state != good) ->
        biu_repair_lambda: (biu1_state'=good) & (biu1_permanent'=false);

endmodule
```

Figure 6. Baseline PRISM Model: A BIU

The other components of ROBUS—that is, the other BIUs and the RMUs—are modeled in the exact same way as BIU1. Their PRISM specification is identical to module `biu1` of Figure 6, with state variables and parameters renamed appropriately. The details are shown in the appendix (A.1).

3.3.2 Properties

The remainder of the PRISM model introduces various abbreviations and definitions to describe SPIDER's maximal fault assumption. The model includes three BIUs, so part (1) of the fault assumption states that either at least two of them are good, or that one of them is good and the other two are benign. It is straightforward to express these two conditions in PRISM:

```
formula two_good_bius =
    (biu1_state=good & biu2_state=good)
  | (biu1_state=good & biu3_state=good)
  | (biu2_state=good & biu3_state=good);

formula one_good_two_benign_bius =
    (biu1_state=good & biu2_state=benign & biu3_state=benign)
  | (biu1_state=benign & biu2_state=good & biu3_state=benign)
  | (biu1_state=benign & biu2_state=benign & biu3_state=good);
```

Clauses (2) and (3) of the fault assumptions are specified in a similar way, and the set of states that satisfy the maximal fault assumption is defined by

```
label "functional" =
    (two_good_bius | one_good_two_benign_bius)
  & (two_good_rmus | one_good_two_benign_rmus)
  & (no_asymmetric_bius | no_asymmetric_rmus);
```

Then, the properties we examined include

P1: probability that the system is always functional for a given mission time T. For example, for $T = 10$ hours, this is written

```
P=? [ G<=10 "functional" ]
```

This PRISM query asks for the probability that the maximal fault assumption is satisfied at all times in the interval $[0, T]$.

P2: The probability of system failure for a mission time of 1 hour is less than a bound p_0. For example, if p_0 is 10^{-9}, this is written

```
"all_good" => P<=1e-9 [ F<=1 !"functional" ]
```

In this formula, label `"all_good"` denotes the set of system states where all components are good.

P3: The probability of system failure within one hour after a component fails symmetric is less than a bound p_1. For example, with $p_1 = 10^{-6}$,

```
"one_fault"=>P<=1e-6 [ F<=1 !"functional" ]
```

This last property allows us to investigate the reliability of a 3×3 ROBUS instance after the symmetric failure of a single BIU or RMU.[2]

The complete list of properties that we evaluated with PRISM is given in the appendix (A.1).

[2]The PRISM model described here does not include the diagnosis and reconfiguration features of ROBUS, which are intended to help survive such a component failure. However, the PRISM model can be extended to take diagnosis into account, as done in [22] or [19].

Property	Parameter	Result	Run time
P1	$T = 5$	0.99999999814	0.41 s
P1	$T = 10$	0.99999999359	0.59 s
P1	$T = 100$	0.99999945515	4.15 s
P2	$p_0 = 10^{-8}$	true	0.28 s
P2	$p_0 = 10^{-9}$	true	0.29 s
P2	$p_0 = 10^{-10}$	false	0.29 s
P3	$p_1 = 10^{-3}$	true	0.29 s
P3	$p_1 = 10^{-4}$	true	0.29 s
P3	$p_1 = 10^{-5}$	false	0.28 s
P3	$p_1 = 10^{-6}$	false	0.28 s
P3	$p_1 = 10^{-7}$	false	0.31 s

Table 2. Example Analysis Results (3 BIUs, 3 RMUs)

3.3.3 Analysis Results

Table 2 shows the results and the runtime of the PRISM model checker on different instances of properties **P1** to **P3**. This experiment was run on a 64bit Intel Core 2 computer, with 4 GB RAM, and running Linux 2.6 (Ubuntu 8.04). All the analysis was done using PRISM's default settings. Memory consumption was not very large in this example, so we do not include memory usage data in the table. We used the following model parameters:

- *Rates:* `trans_lambda` $= 10^{-5}$, `perm_lambda` $= 10^{-5}$, `repair_lambda` $= 10$
- *Probabilities:* `benign_ratio` $= 0.5$, `symmetric_ratio` $= 0.3$

As can be seen from the table, the model checker is fast in this example. The results are as one would expect. There is enough redundancy in this ROBUS instance to achieve very high reliability, of the order of 10^{-9} system failures per hour. Reliability decreases significantly if one RMU or BIU suffers a (nondiagnosed) symmetric fault.

It is easy to extend the PRISM model to larger versions of ROBUS by adding more copies of the BIU and RMU modules. Tables 3 and 4 show experimental results for a 4×3 instance (4 BIUs and 3 RMUs), and for a 4×4 instance. The rate and probability parameters were set as in the previous experiment. On these examples, one observes a significant increase in the model checker's runtime. This is caused by the well-known state-explosion problem, as the number of states in the PRISM model (and the Markov chain) grows exponentially with the number of system modules. PRISM employs data structures based on decision diagrams [14] and sparse matrix representations, to attempt to store the Markov chain compactly, but scalability to systems with a large number of modules remains an issue.

The model checker also uses iterative methods to compute transient probability distributions, and one observes that the number of iterations required increases with the number of modules. An increase in the number of iterations also explains why evaluating property **P1** for $T = 100$ takes much longer than for $T = 10$ or $T = 5$.

Property	Parameter	Result	Run time
P1	$T = 5$	0.99999999838	8.99 s
P1	$T = 10$	0.99999999443	14.56 s
P1	$T = 100$	0.99999952577	98.90 s
P2	$p_0 = 10^{-8}$	true	5.318 s
P2	$p_0 = 10^{-9}$	true	5.314 s
P2	$p_0 = 10^{-10}$	false	5.298 s
P3	$p_1 = 10^{-3}$	true	5.315 s
P3	$p_1 = 10^{-4}$	true	5.32 s
P3	$p_1 = 10^{-5}$	false	5.317 s
P3	$p_1 = 10^{-6}$	false	5.34 s
P3	$p_1 = 10^{-7}$	false	5.318 s

Table 3. Experiment: 4 BIUs and 3 RMUs

Property	Parameter	Result	Run time
P1	$T = 5$	0.99999999861	59.09 s
P1	$T = 10$	0.99999999520	103.03 s
P1	$T = 100$	0.99999959193	732.9148 s
P2	$p_0 = 10^{-8}$	true	33.057 s
P2	$p_0 = 10^{-9}$	true	33.052 s
P2	$p_0 = 10^{-10}$	false	33.037 s
P3	$p_1 = 10^{-3}$	true	33.042 s
P3	$p_1 = 10^{-4}$	true	33.074 s
P3	$p_1 = 10^{-5}$	false	33.045 s
P3	$p_1 = 10^{-6}$	false	33.048 s
P3	$p_1 = 10^{-7}$	false	33.077 s

Table 4. Experiment: 4 BIUs and 4 RMUs

```
module spider_bius
  good_bius: [0..N] init N;
  benign_transient_bius: [0..N] init 0;
  symmetric_transient_bius: [0..N] init 0;
  asymmetric_transient_bius: [0..N] init 0;
  benign_permanent_bius: [0..N] init 0;
  symmetric_permanent_bius: [0..N] init 0;
  asymmetric_permanent_bius: [0..N] init 0;

  [] good_bius>0 &
     benign_transient_bius<N & benign_permanent_bius<N &
     symmetric_transient_bius<N & symmetric_permanent_bius<N &
     asymmetric_transient_bius<N & asymmetric_permanent_bius<N
   ->
       (biu_trans_lambda * good_bius * biu_benign) :
            (good_bius'=good_bius-1) &
            (benign_transient_bius'= benign_transient_bius+1)
     + (biu_trans_lambda * good_bius * biu_symmetric) :
            (good_bius'=good_bius-1) &
            (symmetric_transient_bius'= symmetric_transient_bius+1)
     ...

     + (biu_perm_lambda * good_bius * biu_asymmetric) :
            (good_bius'=good_bius-1) &
            (asymmetric_permanent_bius'=asymmetric_permanent_bius+1);

  [] (benign_transient_bius>0 & good_bius<N) ->
       (biu_repair_lambda * benign_transient_bius) :
          (good_bius'=good_bius+1) &
          (benign_transient_bius'=benign_transient_bius-1);

   ...
endmodule
```

Figure 7. Revised Model for N BIUs

3.3.4 Revised Model

Rather than representing each component of the system as a separate PRISM module, a more traditional approach is to count the number of BIUs and RMUs in each fault class (cf. [5] for example). This modeling method takes advantage of the symmetries present in SPIDER to reduce the number of states in the Markov chain and increase scalability. We describe how such a model can be constructed and analyzed with PRISM.

As previously, the PRISM system is parameterized by fault rates and by the probabilities of each type of fault. In addition, this new model is parameterized by N and M—that is, the number of BIUs and RMUs, respectively. The system consists of two similar modules; one describes the set of BIUs and the other models the RMU faults. The BIU model is sketched in Figure 7.

This module has seven state variables with values in the interval $[0, N]$. Variable `good_bius` is the number of good BIUs, `benign_transient_bius` counts the number of BIUs that have suffered a benign, transient fault, and so forth. Then, a fault can occur as long as the number of good BIUs is non-zero, and the effect of a fault is to decrement `good_bius` and increment one of the other counters. The rate for a fault

type is the product of the fault arrival rate, the probability of the fault type, and the number of good BIUs. For example, the rate of asymmetric, permanent faults is given by

```
biu_perm_lambda * good_bius * biu_asymmetric
```

as shown in Figure 7. The PRISM model then includes one probabilistic transition to model the occurrence of faults. For technical reasons, the guard of this transition is the conjunction

```
good_bius>0 &
benign_transient_bius<N & benign_permanent_bius<N &
symmetric_transient_bius<N & symmetric_permanent_bius<N &
asymmetric_transient_bius<N & asymmetric_permanent_bius<N
```

Intuitively, the constraint `good_bius>0` should be sufficient: faults can occur as long as there are non-faulty BIUs. But PRISM requires additional constraints such as `benign_transient_bius<N` for type checking. Without this constraint, it is not possible to increment `benign_transient_bius` since this variable's value must be in the interval $[0, N]$. Adding such constraints does not change the underlying Markov chain, as all states of the chain satisfy the following invariant

```
good_bius + benign_transient_bius + benign_permanent_bius +
 symmetric_transient_bius + symmetric_permanent_bius +
  asymmetric_transient_bius + asymmetric_permanent_bius = N.
```

Recovery from transient faults is modeled in a similar fashion, as shown in Figure 7.

The fault model for the set of RMUs is identical to the BIU model (cf. the appendix [A.2]), and the maximal fault assumption is specified as shown in Figure 8.

We analyzed the same properties as previously using this revised model. The exact PRISM formulas used in this study are listed in the appendix (A.2). Since the model is now parameterized by N and M, we can ask PRISM to check these properties for different instances of ROBUS, by giving ranges of values for N and M (see [25] for details).

Table 5 shows the runtimes and results of PRISM verification of property **P1** for different ROBUS instances, where N varies from 3 to 15 and M from 3 to 5, and for different values of the mission time T. In this experiment, the rates and fault probability parameters were set as in the previous section:

- *Rates:* `trans_lambda` $= 10^{-5}$, `perm_lambda` $= 10^{-5}$, `repair_lambda` $= 10$
- *Probabilities:* `benign_ratio` $= 0.5$, `symmetric_ratio` $= 0.3$

As expected, this revised model is much more scalable than the previous model and can be used to analyze much larger instances of SPIDER. One can also observe that for the chosen fault rates, this model predicts that maximal reliability is achieved with about five RMUs and BIUs. With more components, the system reliability decreases slightly. This can be explained by the fact that SPIDER cannot tolerate Byzantine failures in both RMUs and BIUs, and with more system components, the probability that a least one RMU and one BIU are asymmetric increases.[3]

3.4 Protocol Models in PRISM

A tool such as PRISM has been applied to the verification of randomized leader-election algorithms and other distributed protocols. Here, we describe an attempt to model one of SPIDER's protocols and analyze it with PRISM.

[3]This phenomenon may be more pronounced in this model than in real SPIDER implementations, since component diagnosis and reconfiguration are not modeled.

```
formula symmetric_bius =
   symmetric_transient_bius+symmetric_permanent_bius;

formula asymmetric_bius =
   asymmetric_transient_bius+asymmetric_permanent_bius;

formula bad_bius = symmetric_bius + asymmetric_bius;

formula symmetric_rmus =
   symmetric_transient_rmus+symmetric_permanent_rmus;

formula asymmetric_rmus =
   asymmetric_transient_rmus+asymmetric_permanent_rmus;

formula bad_rmus = symmetric_rmus + asymmetric_rmus;

//
// Maximal fault assumption:
// - number of good BIUs > number of bad BIUs
// - number of good RMUs > number of bad RMUs
// - no asymmetric BIU or no asymmetric RMU
//
label "functional" =
   (good_bius > bad_bius) & (good_rmus > bad_rmus)
 & (asymmetric_bius = 0 | asymmetric_rmus = 0);
```

Figure 8. Maximal Fault Assumption in PRISM

		$T = 5$		$T = 10$		$T = 100$	
M	N	Result	Time	Result	Time	Result	Time
3	3	0.999999994380	0.023 s	0.999999985032	0.021 s	0.999999360021	0.047 s
3	4	0.999999995108	0.051 s	0.999999986972	0.052 s	0.999999442968	0.135 s
3	5	0.999999996704	0.107 s	0.999999991222	0.114 s	0.999999624638	0.334 s
3	6	0.999999996565	0.205 s	0.999999990853	0.223 s	0.999999608883	0.731 s
3	7	0.999999996427	0.396 s	0.999999990484	0.444 s	0.999999593130	1.543 s
3	8	0.999999996288	0.886 s	0.999999990114	1.006 s	0.999999577341	3.561 s
3	9	0.999999996149	1.956 s	0.999999989745	2.281 s	0.999999561553	9.076 s
3	10	0.999999996010	3.075 s	0.999999989375	3.693 s	0.999999545767	15.215 s
3	11	0.999999995872	4.539 s	0.999999989005	5.521 s	0.999999529981	23.911 s
3	12	0.999999995733	6.858 s	0.999999988636	8.422 s	0.999999514197	37.450 s
3	13	0.999999995594	10.021 s	0.999999988266	12.343 s	0.999999498414	57.240 s
3	14	0.999999995455	14.194 s	0.999999987897	17.565 s	0.999999482633	83.052 s
3	15	0.999999995316	19.949 s	0.999999987527	24.896 s	0.999999466852	121.887 s
4	3	0.999999995108	0.067 s	0.999999986973	0.055 s	0.999999442968	0.137 s
4	4	0.999999995791	0.142 s	0.999999988790	0.129 s	0.999999520647	0.377 s
4	5	0.999999997340	0.290 s	0.999999992916	0.290 s	0.999999697038	0.942 s
4	6	0.999999997155	0.620 s	0.999999992424	0.677 s	0.999999676016	2.266 s
4	7	0.999999996970	1.418 s	0.999999991931	1.582 s	0.999999654995	5.688 s
4	8	0.999999996785	2.826 s	0.999999991438	3.259 s	0.999999633939	12.819 s
4	9	0.999999996600	4.510 s	0.999999990946	5.430 s	0.999999612886	21.825 s
4	10	0.999999996415	7.003 s	0.999999990453	8.329 s	0.999999591833	36.572 s
4	11	0.999999996230	10.813 s	0.999999989960	13.298 s	0.999999570783	58.089 s
4	12	0.999999996045	16.394 s	0.999999989467	20.056 s	0.999999549734	92.057 s
4	13	0.999999995860	24.324 s	0.999999988975	29.980 s	0.999999528687	140.312 s
4	14	0.999999995675	34.667 s	0.999999988482	43.105 s	0.999999507641	206.737 s
4	15	0.999999995490	48.679 s	0.999999987989	61.488 s	0.999999486597	302.655 s
5	3	0.999999996704	0.112 s	0.999999991222	0.120 s	0.999999624638	0.347 s
5	4	0.999999997340	0.267 s	0.999999992916	0.289 s	0.999999697038	0.930 s
5	5	0.999999998843	0.713 s	0.999999996919	0.805 s	0.999999868138	2.680 s
5	6	0.999999998612	1.947 s	0.999999996304	2.179 s	0.999999841838	8.154 s
5	7	0.999999998381	3.298 s	0.999999995688	3.860 s	0.999999815539	15.385 s
5	8	0.999999998150	5.613 s	0.999999995072	6.633 s	0.999999789207	27.815 s
5	9	0.999999997918	9.382 s	0.999999994456	11.140 s	0.999999762876	47.858 s
5	10	0.999999997687	14.847 s	0.999999993840	17.993 s	0.999999736547	79.659 s
5	11	0.999999997456	23.451 s	0.999999993224	28.736 s	0.999999710220	131.086 s
5	12	0.999999997225	35.189 s	0.999999992608	43.689 s	0.999999683896	206.533 s
5	13	0.999999996993	51.936 s	0.999999991992	65.307 s	0.999999657573	313.023 s
5	14	0.999999996762	74.964 s	0.999999991376	94.072 s	0.999999631253	463.504 s
5	15	0.999999996531	106.132 s	0.999999990760	134.989 s	0.999999604934	677.744 s

Table 5. Example Analysis Results for the Revised SPIDER Model

3.4.1 SPIDER's Broadcast Service

SPIDER provides a broadcast service to the PEs by means of a two-round interactive consistency protocol described in [33]. If SPIDER's maximal fault assumption is satisfied, then this protocol is guaranteed to achieve *consistent broadcast:*

- If at the end of the protocol, a fault-free PE receives message m, then all fault-free PEs receive the exact same m.
- If the BIU and originating PE are fault free, then m is identical to the message broadcast.

We consider a simplified version of this protocol that does not take diagnosis and group membership into account. This simplified protocol works as follows. When a PE is scheduled to transmit data, it sends a message m to its BIU. This BIU checks that the message is well-formed (e.g., whether the CRC is valid). If the message m is valid, the BIU forwards it to all the RMUs; otherwise, the BIU sends the special message `PE_ERROR`. Every RMU checks the validity of the message it receives. If the message is valid, the RMU resends it to all BIUs. Otherwise, the RMU sends `SOURCE_ERROR` to all BIUs. At this point, each BIU receives M messages $m_1, \ldots, m_M$ and it performs a vote on this set of messages. If more than half of these messages are identical, then the majority copy is selected by the BIU and is relayed to the attached PE. Otherwise, the BIU forwards a special message `NO_MAJORITY` to the PE.

Our goal is now to examine the probability that this protocol achieves consistent broadcast, under some assumptions about the rates and types of failure of BIUs and RMUs.

3.4.2 PRISM Model

We have developed a PRISM model of the interactive consistency protocol, for a ROBUS instance with three BIUs and three RMUs. The PRISM model is actually even more simplified: it describes a single execution of the protocol, where PE_1 is the broadcast source. The full model is given in the appendix (B).

Unlike all the PRISM examples described previously, this model is a discrete-time Markov decision process. It mixes probabilistic state transitions that model failure probabilities, and nondeterministic transitions that model the behavior of faulty components. A nonfaulty component executes the protocol properly, but may become either benign, symmetric, or asymmetric faulty with some probability. Once a component is symmetric or asymmetric, it can exhibit a range of incorrect behaviors. We model the set of possible behaviors as a nondeterministic choice between alternatives, to let the PRISM model checker discover the worst-case scenarios.

The modeling follows a few general principles. Each communication link is modeled as a one-place buffer, represented as a global system variable in PRISM. Protocol components are modeled as state-transition systems (i.e., PRISM modules). The transmission of a message is modeled by writing the message into the global variable representing the appropriate channel. Conversely, the reception of a message is modeled by reading the channel variable. To properly coordinate the read and write operations, the transmitting and receiving modules synchronize using transition labels.

The constant and global variables shown in Figure 9 model the protocol messages and the communication channels. Messages are represented by integers in the interval $[0, 7]$. There are three types of messages: normal data, special control messages, and the special value `bad_message` that represents any manifestly incorrect message. This special value is intended to model missing messages, messages with a wrong CRC, or any other other type of error that can be recognized by the recipient.

Because PRISM requires all types to be finite, we represent the possible valid data by the three constants `data1`, `data2`, and `data3`. At least three good values are necessary to model fault scenarios such as the

```
const int data1 = 0;
const int data2 = 1;
const int data3 = 2;
const int pe_error = 4;
const int source_error = 5;
const int no_majority = 6;
const int bad_message = 7;

// Global variables = communication links
global pe1_biu1: [0..7] init 0;
global pe2_biu2: [0..7] init 0;
global pe3_biu3: [0..7] init 0;

global biu1_rmu1: [0..7] init 0;
global biu1_rmu2: [0..7] init 0;
global biu1_rmu3: [0..7] init 0;
global biu2_rmu1: [0..7] init 0;
global biu2_rmu2: [0..7] init 0;
global biu2_rmu3: [0..7] init 0;
global biu3_rmu1: [0..7] init 0;
global biu3_rmu2: [0..7] init 0;
global biu3_rmu3: [0..7] init 0;

// Source data that PE1 will broadcast
const int source = data3;
```

Figure 9. Messages and Communication Channels

transmission of distinct messages by an asymmetric BIU to the three RMUs, without any of these messages being recognized as incorrect by the recipient.[4]

The global variables such as `pe1_biu1` represent bidirectional channels between two components (e.g., between PE1 and BIU1).

The protocol processes are modeled as illustrated in Figure 10. The figure shows a fragment of the PE1 model, which consists of two modules: `pe1_error` is the fault model for PE1, and `pe1` models protocol execution.

Module `pe1_error` specifies that PE1 may suffer a single fault at an arbitrary time during protocol execution. The time of occurrence is nondeterministic, but the fault type is randomly selected according to the probability distribution defined by the parameters `pe_benign` and `pe_symmetric`.

Module `pe1` has a main control variable `pe1_ctrl` that specifies successive steps of the protocol. For example, from state 0, the process initiates a broadcast by transmitting the `source` message to BIU1 and then transitions to the next step, where `pe1_ctrl=1`. The actual message transmitted depends on PE1's current fault status. If PE1 is fault free, then the source data is sent; if PE1 is benign, then the `bad_message` is sent (BIU1 will recognize this message as incorrect); otherwise, PE1 has suffered a symmetric fault and it may send arbitrary data to BIU1. We model the latter case, as a nondeterministic choice between three state transitions, as shown in Figure 10.

In all cases, the broadcast action sets `pe1_ctrl` to 1, and the only possible transition of process `pe1`

[4] One could argue that more good data values would allow us to model more general faulty behaviors, such as, an asymmetric RMU sending three distinct messages that are also different from the message it receives. Unfortunately, we were not able to add an extra `data4` value, as this causes PRISM to run out of memory.

```
module pe1_error
  pe1_faulted: bool init false;
  pe1_error: [0..2] init fault_free;

  [] (!pe1_faulted) ->
         pe_benign:    (pe1_error'=benign_faulty) & (pe1_faulted'=true)
       + pe_symmetric: (pe1_error'=symmetric_faulty) & (pe1_faulted'=true)
       + pe_nofault:   (pe1_error'=fault_free) & (pe1_faulted'=true);

endmodule

module pe1
  pe1_ctrl: [0..4] init 0;
  pe1_out: [0..7] init 0;

  // broadcast
  [] (pe1_ctrl=0) & (pe1_error=fault_free) -> (pe1_ctrl'=1) &
                                              (pe1_biu1' = source);
   ...
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
                                                    (pe1_biu1'=data1);
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
                                                    (pe1_biu1'=data2);
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
                                                    (pe1_biu1'=data3);

  [pe1_broadcast] (pe1_ctrl=1) -> (pe1_ctrl'=2);
  ...
endmodule
```

Figure 10. Broadcast Protocol: Process PE1

```
// BIU1
module biu1
  biu1_ctrl: [0..6] init 0;

  // wait for broadcast from pe1
  [pe1_broadcast] (biu1_ctrl=0) -> (biu1_ctrl'=1);

  // Fault-free case: consume the message and forward it to all rmus
  [] (biu1_ctrl=1) & (biu1_error=fault_free) & (pe1_biu1!=bad_message) ->
              (biu1_ctrl'=2) & (biu1_rmu1'=pe1_biu1) &
              (biu1_rmu2'=pe1_biu1) & (biu1_rmu3'=pe1_biu1);
  [] (biu1_ctrl=1) & (biu1_error=fault_free) & (pe1_biu1=bad_message) ->
              (biu1_ctrl'=2) & (biu1_rmu1'=pe_error) &
              (biu1_rmu2'=pe_error) & (biu1_rmu3'=pe_error);
```

Figure 11. Broadcast Protocol: Process BIU1

after the broadcast increments `pe1_ctrl`. This transition has the label `pe1_broadcast`, and its role is to synchronize `pe1` and `biu1`, so that the latter can consume the message just transmitted. Figure 11 shows the corresponding specification fragment for module `biu1`. Initially, `biu1` is in state 0. It stays in that state until it can execute the transition labeled with `pe1_broadcast`, which increments `biu1_ctrl` so that the next transitions from `biu1` are enabled. Since all transitions that share the same label must execute simultaneously, both `pe1` and `biu1` perform this transition: that is, just after `pe1` has transmitted a message, and just before `biu1` reads that message. A similar mechanism, based on synchronized labeled transitions is used throughout the PRISM specification to model all communication.

3.4.3 Protocol Properties

The full protocol model is specified in the appendix (B). For scalability reasons, we have included fault models for only two components—`pe1` and `biu1`. All the other PEs and BIUs, and all the RMUs are assumed to be fault free. Several sets of states are identified and used in property specifications:

```
label "done" = pe1_ctrl=4 & pe2_ctrl=2 & pe3_ctrl=2;

label "all_good" = (pe1_out=source)  & (pe2_out=source)
                 & (pe3_out=source);

label "unanimous" = (pe1_out=pe2_out) & (pe1_out=pe3_out);

label "nonfaulty_agree" = (pe2_out=pe3_out);

label "nonfaulty_valid" = (pe2_out=source) & (pe3_out=source);
```

The set of states called `done` indicates protocol termination. The other sets of states capture two variants of the consistency and validity properties that the protocol is intended to achieve. For example, `all_good` means that all PEs receive the `source` message, whereas `unanimous` is the weaker property that all PEs receive the same thing.

We examined the protocol properties listed in the appendix (B). First, we expect that the protocol always terminates, so the following PRISM property should be true:

```
"init"=>P>=1 [ F "done" ]
```

Then, we investigate the probabilities of agreement and validity. Since the PRISM model is a Markov decision process, the actual probabilities depend on how the nondeterministic choices are resolved. For example, we can consider two variants of the same property:

```
Pmin =? [ G("done" => "all_good") ]

Pmax =? [ G("done" => "all_good") ]
```

In the first line, we ask for the minimal probability that the protocol ends in a state where all PEs receive the broadcast as transmitted. This is the probability of success under the worst possible fault scenario allowed in our model. Conversely, the second line asks for the maximal probability of success (which should be 1).

3.4.4 Analysis Results

Table 6 shows the results and runtime we observed when running the PRISM model checker on the above example. It also shows model-construction time and the amount of memory consumed by the PRISM process. This data was measured on a 64bit Linux 2.6 system, with an Intel Core 2 CPU, and 4 GB of RAM. The results show that it is possible to analyze these example properties in about 8 minutes. However, the performance numbers are disappointing given the size of the example model. Most of the verification time is spent by PRISM in building an internal representation of the MDP. The internal representation is based on multi-terminal binary decision diagrams (MTBDDs), which appear to be a very poor representation for this model. In particular, the amount of memory used to store the MTBDDs and other internal data structures is enormous; it requires more than 3 GB. This seems to be a serious limitation of the MTBDD representation, as the statistics displayed by PRISM indicate that the actual size of the underlying MDP is not that large: it contains 15446 states and 50357 transitions.[5]

It is well-known that the variable ordering used by an MTBDD can have a major impact on the size of this MTBDD. But there is no obvious way to control which ordering PRISM will use. All our efforts at reorganizing the specifications by changing module and declaration order did not have an impact on the analysis time or memory usage. There has been some work on heuristics for obtaining good variable ordering for MTBDDs in PRISM [20], but these heuristics were not implemented in the PRISM version that we used.

4 Conclusion

We have described several applications of the PRISM probabilistic model checker to fault-tolerant distributed systems.

We first investigated the use of PRISM in reliability modeling, as an alternative to other tools such as SURE or Möbius. In this study, we considered two different approaches to modeling the reliability of the SPIDER system. Both approaches relied on continuous-time Markov chains. The main result of this experiment is that PRISM is perfectly adequate as a reliability analysis tool. The PRISM specification language is expressive enough and enables easy modeling. The model checker performance is also good, and PRISM can be used to analyze relatively large and detailed models. There is still a limitation to this approach, since PRISM cannot encode or analyze non-Markovian models, unlike other tools that support semi-Markovian models or more general classes of stochastic systems.

In a second experiment, we examined application of PRISM to the analysis of a more detailed SPIDER protocol, encoded as a state machine. The PRISM model developed for this purpose was a Markov decision

[5] We have observed very high memory usage (gigabytes) on much smaller MDPs with fewer than a hundred states and a few hundred transitions.

Property	Result	Runtime
`"init"=>P>=1 [ F "done" ]`	true	1.801 s
`P>=1 [ G ("done"=>"all_good") ]`	false	3.041 s
`P>=1 [ G ("done"=>"unanimous") ]`	false	1.950 s
`Pmin =? [ G("done" => "unanimous") ]`	0.7	1.620 s
`Pmax =? [G("done" => "unanimous")]`	1.0	0.727 s
`Pmin =? [G ("done" => "nonfaulty_agree")]`	1.0	0.021 s
`Pmax =? [G ("done" => "nonfaulty_agree")]`	1.0	0.060 s
`Pmin =? [G ("done" => "nonfaulty_valid")]`	0.56	2.195 s
`Pmax =? [G ("done" => "nonfaulty_valid")]`	1.0	0.473 s

Model Construction:	450.023 s
Virtual Memory Used:	3676 Mb
Resident Memory Used:	2358 Mb

Table 6. Broadcast Protocol: Analysis Results

process, in which fault occurrence and the behavior of faulty components are modeled via nondeterministic choices, while the type of fault is determined by a probability distribution given a priori. This experiment showed that it is possible to model and analyze a fault-tolerant protocol using PRISM, but it also showed some limits of the PRISM tool. Although the protocol modeled was quite small, PRISM required a very large amount of memory to build and store its internal data structures for this Markov decision process. Scalability to more complex protocols or systems is then questionable.

References

1. M. H. Azadmanesh and R. M. Kieckhafer. Exploiting Omissive Faults in Synchronous Approximate Agreement. *IEEE Transactions on Computers*, 48(10):1031–1042, October 2000.

2. Adnan Aziz, Kumud Sanwak, Vigyan Singhal, and Robert Brayton. Verifying Continuous Time Markov Chains. In *Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 269–276. Springer-Verlag, 1996.

3. Andrea Bondavalli, Manuela Nelli, Luca Simoncini, and Giorgio Mongardi. Hierarchical Modeling of Complex Control Systems: Dependability Analysis of a Railway Interlocking. *Computer Systems Science and Engineering*, 16(4):249–261, July 2001.

4. Ricky W. Butler. The SURE Approach to Reliability Analysis. *IEEE Transactions on Reliability*, 41(2):210–218, June 1992.

5. Ricky W. Butler and Sally C. Johnson. Techniques for Modeling the Reliability of Fault-Tolerant Systems with the Markov State-Space Approach. Reference Publication NASA RP-1348, NASA Langley Research Center, September 1995.

6. C. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and Stochastic Modeling with SMART. *Performance Evaluation*, 63(6):578–608, June 2006.

7. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

8. Tod Courtney, Shravan Gaonkar, Ken Keefe, Eric W. D. Rozier, and William H. Snaders. Möbius 2.3: An Extensible Tool for Dependability, Security, and Performance Evaluation of Large and Complex System Models. In *International Conference on Dependable Systems and Networks (DSN'09)*, pages 353–358, June 2009.

9. R. M. Geist and M. Smotherman. Utlrahigh Reliability Estimates through Simulation. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 350–355, January 1989.

10. Robert Geist and Kishor Trivedi. Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques. *IEEE Computer*, 23(7):52–61, July 1990.

11. Alfons Geser and Paul S. Miner. A New On-Line Diagnosis Protocol for the SPIDER Family of Byzantine Fault Tolerant Architectures. Technical Memorandum NASA/TM-2004-212432, NASA Langley Research Center, December 2004.

12. Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, September 1994.

13. Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A Tool for Model Checking Markov Chains. *International Journal on Software Tools for Technology Transfer*, 4(2):153–172, February 2003.

14. Holger Hermanns, Marta Kwiatkowska, Gethin Norman, David Parker, and Markus Siegle. On the Use of MTBDDs for Performability Analysis and Verification of Stochastic Systems. *Journal of Logic and Algebraic Progamming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, 56(1–2):23–67, June 2003.

15. Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer-Verlag, 2006.

16. Joost-Pieter Kaoten, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *Quantitative Evaluation of Systems (QEST)*, pages 167–176. IEEE Computer Society, 2009.

17. Marta Kwiatkowska, Gethin Norman, and David Parker. Advances and Challenges of Probabilistic Model Checking. In *Proc. 48th Annual Allerton Conference on Communication, Control, and Computing*. IEEE Press, 2010.

18. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer-Aided Verification (CAV'11)*, 2011. To appear. Preprint available at `http://www.prismmodelchecker.org/publications.php`.

19. Elizabeth Latronico, Paul Miner, and Philip Koopman. Quantifying the Reliability of Proven SPIDER Group Membership Service Guarantees. In *International Conference on Dependable Systems and Networks (DSN'04)*, pages 275–284, 2004.

20. Vivien Maisonneuve. Automated Heuristic-Based Generation of MTBDD Variable Orderings for PRISM Models. Internship report, École Normale Supérieure de Cachan, 2009. Available at `http://www.prismmodelchecker.org/`.

21. Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A Unified Fault-Tolerance Protocol. In *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 2004.

22. Paul S. Miner, Mahyar Malekpour, and Wilfredo Torres. A Conceptual Design for a Reliable Optical Bus (ROBUS). In *Digital Avionics Systems Conference (DASC)*, volume 2, pages 13D3–1–13D3–11, October 2002.

23. Lee Pike, Jeffrey Maddalon, Paul Miner, and Alfons Geser. Abstractions for Fault-Tolerant Distributed System Verification. In *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 257–270. Springer-Verlag, 2004.

24. Lee Pike, Paul Miner, and Wilfredo Torres-Pomales. Model-Checking Failed Conjectures in Theorem Proving: A Case Study. Technical Memorandum NASA/TM-2004-213278, NASA Langley Research Center, October 224.

25. *PRISM Manual. Version 3.3.1*, 2009. Available at `http://www.prismmodelchecker.org/manual/`.

26. *The PRISM Language: Semantics*. Available at `http://www.prismmodelchecker.org/doc/semantics.pdf`.

27. Robin A. Sahner and Jishor S. Trivedi. Reliability Modeling using SHARPE. *IEEE Transactions on Reliability*, 36(2):186–193, June 1987.

28. William H. Sanders. Integrated Frameworks for Multi-Level and Multi-Formalism Modeling. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 2–9, September 1999.

29. Koushik Sen, Mahesh Viswanathan, and Gul Agha. VESTA: A Statistical Model Checker and Analyzer for Probabilistic Systems. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 251–252, September 2005.

30. Perwez Shahabuddin. Importance Sampling for the Simulation of Highly Reliable Markovian Systems. *Management Science*, 40(3):333–352, March 1994.

31. Frederick T. Sheldon and Kshamta Jerath. Assessing the Effect of Failure Severity, Coincident Failures and Usage Profiles on the Reliability of Embedded Control Systems. In *ACM Symposium on Applied Computing*, pages 14–17, March 2004.

32. Philip Thambidurai and You-Keun Park. Interactive Consistency with Multiple Failure Modes. In *Seventh Symposium on Reliable Distributed Systems*, pages 93–100, 1988.

33. Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul S. Miner. ROBUS–2: A Fault-Tolerant Broadcast Communication System. Technical Memorandum NASA/TM-2005-213540, NASA Langley Research Center, March 2005.

34. Hakøan L. S. Younes. Ymer: A Statistical Model Checker. In *Computer-Aided Verification (CAV'2005)*, volume 3576 of *Lecture Notes in Computer Scinece*, pages 429–433. Springer-Verlag, 2005.

Appendix A

Reliability Models of SPIDER

A.1 Baseline Model

```
//
// Fault/reliability model for a SPIDER instance
// modeled as a continuous-time Markov chain
//
// This model includes 3 BIUs and 3 RMUs
//
// All components are assumed to have similar fault models parameterized by:
//    perm_lambda: rate of permanent faults
//    trans_lambda: rate of transient faults
//    repair_lambda: recovery rate from transient faults
//
// Three classes of faults are considered: benign, symmetric, and asymmetric.
//
// Two parameters define the probability that a fault is benign or symmetric:
//    benign_ratio: probability that a fault is benign
//    symmetric_ratio: probability that a fault is symmetric
//    asymmetric_ratio: probability that a fault is asymmetric
//              = (1 - benign_ratio - symmetric_ratio)
//

ctmc

// Error states of each component
const int good = 0;
const int benign = 1;
const int symmetric = 2;
const int asymmetric = 3;


// Parameters: arrival rates for each component
const double perm_lambda;   // permanent faults   (e.g., 10^-6)
const double trans_lambda;  // transient faults   (e.g., 10^-5)
const double repair_lambda; // recovery from transient faults (e.g., 10)

// Fractions of benign/symmetric/asymmetric faults
// all must be between 0 and 1 and they must add to 1
const double benign_ratio;          // 0.5
const double symmetric_ratio;       // 0.3
const double asymmetric_ratio = (1 - benign_ratio - symmetric_ratio);

// Fault arrival rates for the RMUs(faults/hour)
const double rmu_perm_lambda = perm_lambda;
const double rmu_trans_lambda = trans_lambda;
const double rmu_repair_lambda = repair_lambda;
```

```
// Fraction of fault types:
const double rmu_benign = benign_ratio;
const double rmu_symmetric = symmetric_ratio;
const double rmu_asymmetric = asymmetric_ratio;


// Fault arrival rates for the BIUs
const double biu_perm_lambda = perm_lambda;
const double biu_trans_lambda = trans_lambda;
const double biu_repair_lambda = repair_lambda;

// Fractions
const double biu_benign = benign_ratio;
const double biu_symmetric = symmetric_ratio;
const double biu_asymmetric = asymmetric_ratio;


// Error model for BIU1
module biu1
  biu1_state: [0..3] init good;
  biu1_permanent: bool init false;

  [] biu1_state = good ->
       (biu_perm_lambda * biu_benign): (biu1_state'=benign) &
                                       (biu1_permanent'=true)
     + (biu_perm_lambda * biu_symmetric): (biu1_state'=symmetric) &
                                          (biu1_permanent'=true)
     + (biu_perm_lambda * biu_asymmetric): (biu1_state'=asymmetric) &
                                           (biu1_permanent'=true)

     + (biu_trans_lambda * biu_benign): (biu1_state'=benign) &
                                        (biu1_permanent'=false)
     + (biu_trans_lambda * biu_symmetric): (biu1_state'=symmetric) &
                                           (biu1_permanent'=false)
     + (biu_trans_lambda * biu_asymmetric): (biu1_state'=asymmetric) &
                                            (biu1_permanent'=false);

  // recovery from transient faults
  [] (!biu1_permanent & biu1_state != good) ->
        biu_repair_lambda: (biu1_state'=good) & (biu1_permanent'=false);

endmodule

// Same model for BIU2 and BIU3, modulo renaming
module biu2 = biu1[biu1_state=biu2_state,
                   biu1_permanent=biu2_permanent]
endmodule

module biu3 = biu1[biu1_state=biu3_state,
                   biu1_permanent=biu3_permanent]
endmodule
```

```
// Error model for RMU1: same as for BIU1 modulo renaming
module rmu1 = biu1[biu1_state=rmu1_state,
                   biu1_permanent=rmu1_permanent,
                   biu_perm_lambda=rmu_perm_lambda,
                   biu_trans_lambda=rmu_trans_lambda,
                   biu_benign=rmu_benign,
                   biu_symmetric=rmu_symmetric,
                   biu_asymmetric=rmu_asymmetric]
endmodule

// Same model for RMU2 and RMU3 too
module rmu2 = rmu1[rmu1_state=rmu2_state,
                   rmu1_permanent=rmu2_permanent]
endmodule

module rmu3 = rmu1[rmu1_state=rmu3_state,
                   rmu1_permanent=rmu3_permanent]
endmodule


//
// Maximum fault assumption for SPIDER
// - at least two good BIUs or one good BIU and two benign BIUs
// - at least two good RMUs or one good RMU and two benign RMUs
// - at most one asymmetric BIU or RMU
//
formula two_good_bius =
    (biu1_state=good & biu2_state=good) | (biu1_state=good & biu3_state=good)
  | (biu2_state=good & biu3_state=good);

formula one_good_two_benign_bius =
    (biu1_state=good & biu2_state=benign & biu3_state=benign)
  | (biu1_state=benign & biu2_state=good & biu3_state=benign)
  | (biu1_state=benign & biu2_state=benign & biu3_state=good);

formula no_asymmetric_bius =
     biu1_state!=asymmetric & biu2_state!=asymmetric & biu3_state!=asymmetric;


formula two_good_rmus =
    (rmu1_state=good & rmu2_state=good) | (rmu1_state=good & rmu3_state=good)
  | (rmu2_state=good & rmu3_state=good);

formula one_good_two_benign_rmus =
    (rmu1_state=good & rmu2_state=benign & rmu3_state=benign)
  | (rmu1_state=benign & rmu2_state=good & rmu3_state=benign)
  | (rmu1_state=benign & rmu2_state=benign & rmu3_state=good);

formula no_asymmetric_rmus =
     rmu1_state!=asymmetric & rmu2_state!=asymmetric & rmu3_state!=asymmetric;


label "functional" =
    (two_good_bius | one_good_two_benign_bius)
```

```
  & (two_good_rmus | one_good_two_benign_rmus)
  & (no_asymmetric_bius | no_asymmetric_rmus);



//
// Other system states used in properties
//

formula all_bius_good =
   (biu1_state=good) & (biu2_state=good) & (biu3_state=good);

formula all_rmus_good =
   (rmu1_state=good) & (rmu2_state=good) & (rmu3_state=good);


// No faulty component
label "all_good" = all_bius_good & all_rmus_good;


//
// To test reliability after a symmetric fault
//
formula one_symmetric_biu =
    (biu1_state=symmetric & biu2_state=good & biu3_state=good)
  | (biu1_state=good & biu2_state=symmetric & biu3_state=good)
  | (biu1_state=good & biu2_state=good & biu3_state=symmetric);

formula one_symmetric_rmu =
    (rmu1_state=symmetric & rmu2_state=good & rmu3_state=good)
  | (rmu1_state=good & rmu2_state=symmetric & rmu3_state=good)
  | (rmu1_state=good & rmu2_state=good & rmu3_state=symmetric);


label "one_fault" =
    (one_symmetric_biu & all_rmus_good) | (one_symmetric_rmu & all_bius_good);


rewards "total_time"
  true: 1;
endrewards
```

Properties for the previous model

```
// Probability that no component fails over a 10hour run
P=? [ G<=10 "all_good" ]

// Expected time to the first component failure
R{"total_time"}=? [ F !"all_good" ]



// Probability that the system does not fail over a 5hour run
P=? [ G<=5 "functional" ]
```

```
// Probability that the system does not fail over a 10hours run
P=? [ G<=10 "functional" ]

// Probability that the system does not fail over a 100hour run
P=? [ G<=100 "functional" ]


// The probability of system failure over a 1hour run is less than 10^-8
"all_good"=>P<=1e-8 [ F<=1 !"functional" ]

// The probability of system failure over a 1hour run is less than 10^-9
"all_good"=>P<=1e-9 [ F<=1 !"functional" ]

// The probability of system failure over a 1hour run is less than 10^-10
"all_good"=>P<=1e-10 [ F<=1 !"functional" ]



// Probability of system failure after a one hour run is less than 1e-3,
// even if we start with a single benign component failure.
"one_fault"=>P<=1e-3 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-4,
// even if we start with a single benign component failure.
"one_fault"=>P<=1e-4 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-5,
// even if we start with a single benign component failure.
"one_fault"=>P<=1e-5 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-6,
// even if we start with a single benign component failure.
"one_fault"=>P<=1e-6 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-7,
// even if we start with a symmetric component.
"one_fault"=>P<=1e-7 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start with a single benign component failure.
"one_fault"=>P<=1e-8 [ F<=1 !"functional" ]
```

A.2 Revised Model

```
//
// Fault/reliability model for SPIDER
// modeled as a continuous-time Markov chain
//
// This model uses counters for the number of good and faulty RMUs and BIUs
// Two parameters are used: N = number of BIUs, M = number of RMUs
//
// All components are assumed to have similar fault models parameterized by:
//    perm_lambda: rate of permanent faults
//    trans_lambda: rate of transient faults
//    repair_lambda: recovery rate from transient faults
//
// Three classes of faults are considered: benign, symmetric, and asymmetric.
//
// Two parameters define the probability that a fault is benign or symmetric:
//    benign_ratio: probability that a fault is benign
//    symmetric_ratio: probability that a fault is symmetric
//    asymmetric_ratio: probability that a fault is asymmetric =
//                        (1 - benign_ratio - symmetric_ratio)
//

ctmc

// Parameters: arrival rates for each component
const double perm_lambda;   // permanent faults   (e.g., 10^-6)
const double trans_lambda;  // transient faults   (e.g., 10^-5)
const double repair_lambda; // recovery from transient faults  (e.g., 10)

// Fractions of benign/symmetric/asymmetric faults
// all must be between 0 and 1 and they must add to 1
const double benign_ratio;                // e.g., 0.5
const double symmetric_ratio;             // e.g., 0.3
const double asymmetric_ratio = (1 - benign_ratio - symmetric_ratio);

// Fault arrival rates for the RMUs(faults/hour)
const double rmu_perm_lambda = perm_lambda;
const double rmu_trans_lambda = trans_lambda;
const double rmu_repair_lambda = repair_lambda;

// Fraction of fault types:
const double rmu_benign = benign_ratio;
const double rmu_symmetric = symmetric_ratio;
const double rmu_asymmetric = asymmetric_ratio;


// Fault arrival rates for the BIUs
const double biu_perm_lambda = perm_lambda;
const double biu_trans_lambda = trans_lambda;
const double biu_repair_lambda = repair_lambda;

// Fractions
const double biu_benign = benign_ratio;
```

```
const double biu_symmetric = symmetric_ratio;
const double biu_asymmetric = asymmetric_ratio;


// Total number of BIUs and RMUs
const int N;  // number of BIUs
const int M;  // number of RMUs

module spider_bius
  good_bius: [0..N] init N;
  benign_transient_bius: [0..N] init 0;
  symmetric_transient_bius: [0..N] init 0;
  asymmetric_transient_bius: [0..N] init 0;
  benign_permanent_bius: [0..N] init 0;
  symmetric_permanent_bius: [0..N] init 0;
  asymmetric_permanent_bius: [0..N] init 0;

  // Fault of one good BIU
  // NOTE: prism complains if we give only 'good_bius>0' as precondition
  [] good_bius>0 &
     benign_transient_bius<N & benign_permanent_bius<N &
     symmetric_transient_bius<N & symmetric_permanent_bius<N &
     asymmetric_transient_bius<N & asymmetric_permanent_bius<N
   ->
        (biu_trans_lambda * good_bius * biu_benign) :
             (good_bius'=good_bius-1) &
             (benign_transient_bius'= benign_transient_bius+1)
     + (biu_trans_lambda * good_bius * biu_symmetric) :
             (good_bius'=good_bius-1) &
             (symmetric_transient_bius'= symmetric_transient_bius+1)
     + (biu_trans_lambda * good_bius * biu_asymmetric) :
             (good_bius'=good_bius-1) &
             (asymmetric_transient_bius'=asymmetric_transient_bius+1)
     + (biu_perm_lambda * good_bius * biu_benign) :
             (good_bius'=good_bius-1) &
             (benign_permanent_bius'= benign_permanent_bius+1)
     + (biu_perm_lambda * good_bius * biu_symmetric) :
             (good_bius'=good_bius-1) &
             (symmetric_permanent_bius'= symmetric_permanent_bius+1)
     + (biu_perm_lambda * good_bius * biu_asymmetric) :
             (good_bius'=good_bius-1) &
             (asymmetric_permanent_bius'=asymmetric_permanent_bius+1);

  // recovery from transient faults
  [] (benign_transient_bius>0 & good_bius<N) ->
       (biu_repair_lambda * benign_transient_bius) :
          (good_bius'=good_bius+1) &
          (benign_transient_bius'=benign_transient_bius-1);

  [] (symmetric_transient_bius>0 & good_bius<N) ->
       (biu_repair_lambda * symmetric_transient_bius) :
          (good_bius'=good_bius+1) &
          (symmetric_transient_bius'=symmetric_transient_bius-1);
```

```
  [] (asymmetric_transient_bius>0 & good_bius<N) ->
       (biu_repair_lambda * asymmetric_transient_bius) :
          (good_bius'=good_bius+1) &
          (asymmetric_transient_bius'=asymmetric_transient_bius-1);

endmodule


// RMU error model: same thing modulo renaming
module spider_rmus =
    spider_bius[N=M,
                good_bius=good_rmus,
                benign_transient_bius=benign_transient_rmus,
                symmetric_transient_bius=symmetric_transient_rmus,
                asymmetric_transient_bius=asymmetric_transient_rmus,
                benign_permanent_bius=benign_permanent_rmus,
                symmetric_permanent_bius=symmetric_permanent_rmus,
                asymmetric_permanent_bius=asymmetric_permanent_rmus,
                biu_trans_lambda=rmu_trans_lambda,
                biu_perm_lambda=rmu_perm_lambda,
                biu_repair_lambda=rmu_repair_lambda,
                biu_benign=rmu_benign,
                biu_symmetric=rmu_symmetric,
                biu_asymmetric=rmu_asymmetric]
endmodule


// Abbreviations
formula symmetric_bius =
   symmetric_transient_bius+symmetric_permanent_bius;

formula asymmetric_bius =
   asymmetric_transient_bius+asymmetric_permanent_bius;

formula bad_bius = symmetric_bius + asymmetric_bius;


formula symmetric_rmus =
   symmetric_transient_rmus+symmetric_permanent_rmus;

formula asymmetric_rmus =
   asymmetric_transient_rmus+asymmetric_permanent_rmus;

formula bad_rmus = symmetric_rmus + asymmetric_rmus;



//
// Maximal fault assumption:
// - number of good BIUs > number of bad BIUs
// - number of good RMUs > number of bad RMUs
// - no asymmetric BIUs or no asymmetric RMUs
//
label "functional" =
```

```
   (good_bius > bad_bius) & (good_rmus > bad_rmus)
 & (asymmetric_bius=0 | asymmetric_rmus=0);


// No component faulty
label "all_good" = (good_bius=N) & (good_rmus=M);

// At most one symmetric faulty component
label "one_fault" =
   (good_bius>=N-1) & (good_rmus>=M-1)
 & (asymmetric_bius=0) & (asymmetric_rmus=0)
 & (((symmetric_bius=1) & (symmetric_rmus=0))
    | ((symmetric_bius=0) & (symmetric_rmus=1)));


// For expected times
rewards "total_time"
  true: 1;
endrewards
```

Properties for the previous model

```
// Probability that no component fails over a 10hour run
P=? [ G<=10 "all_good" ]

// Expected time to the first component failure
R{"total_time"}=? [ F !"all_good" ]


// Probability that the system does not fail over a 5hour run
P=? [ G<=5 "functional" ]

// Probability that the system does not fail over a 10hours run
P=? [ G<=10 "functional" ]

// Probability that the system does not fail over a 100hour run
P=? [ G<=100 "functional" ]


// The probability of system failure over a 1hour run is less than 10^-9
"all_good"=>P<=1e-9 [ F<=1 !"functional" ]


// Probability of system failure after a one hour run is less than 1e-9,
// even if we start from a start with a single benign component failure.
"one_fault"=>P<=1e-9 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start from a start with a single benign component failure.
"one_fault"=>P<=1e-8 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start from a start with a single benign component failure.
```

```
"one_fault"=>P<=1e-7 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start from a start with a single benign component failure.
"one_fault"=>P<=1e-3 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start from a start with a single benign component failure.
"one_fault"=>P<=1e-4 [ F<=1 !"functional" ]

// Probability of system failure after a one hour run is less than 1e-8,
// even if we start from a start with a single benign component failure.
"one_fault"=>P<=1e-5 [ F<=1 !"functional" ]
```

Appendix B

Model of SPIDER's Broadcast Service

```
mdp

// Messages: encoded in the range [0 .. 7]
//  data1 .. data3 are data messages
//  pe_error, source_error, no_majority: special messages
//  bad_message: manifest fault
const int data1 = 0;
const int data2 = 1;
const int data3 = 2;
const int pe_error = 4;
const int source_error = 5;
const int no_majority = 6;
const int bad_message = 7;

// Error states of each component
const int fault_free = 0;
const int benign_faulty = 1;
const int symmetric_faulty = 2;
const int asymmetric_faulty = 3;

// Transition probabilities for BIU error model
const double biu_benign = 0.1;
const double biu_symmetric = 0.1;
const double biu_asymmetric = 0.1;
formula biu_nofault = (1 - biu_benign - biu_symmetric - biu_asymmetric);

// Transition probabilities for PE error model
const double pe_benign = 0.1;
const double pe_symmetric = 0.1;
formula pe_nofault = (1 - pe_benign - pe_symmetric);

// Transition probabilities for RMU error model
const double rmu_benign = 0.1;
const double rmu_symmetric = 0.1;
const double rmu_asymmetric = 0.1;
formula rmu_nofault = (1 - rmu_benign - rmu_symmetric - rmu_asymmetric);

// Global variables = communication links
// each link is an integer variable
global pe1_biu1: [0..7] init 0;
global pe2_biu2: [0..7] init 0;
global pe3_biu3: [0..7] init 0;

global biu1_rmu1: [0..7] init 0;
global biu1_rmu2: [0..7] init 0;
global biu1_rmu3: [0..7] init 0;
global biu2_rmu1: [0..7] init 0;
global biu2_rmu2: [0..7] init 0;
```

```
global biu2_rmu3: [0..7] init 0;
global biu3_rmu1: [0..7] init 0;
global biu3_rmu2: [0..7] init 0;
global biu3_rmu3: [0..7] init 0;

// source data: what PE1 will broadcast
// it must be valid data
//
// NOTE: prism does not allow me to specify that source must
// be equal to data1, data2, or data3 here.
const int source = data3;


// Majority vote for BIU1 when it receives three messages
// in biu1_rmu1, biu1_rmu2, biu1_rmu3
formula vote1 =
   (biu1_rmu1=biu1_rmu2) ? ((biu1_rmu1!=bad_message) ? biu1_rmu1 : biu1_rmu3)
 : (biu1_rmu2=biu1_rmu3) ? ((biu1_rmu2!=bad_message) ? biu1_rmu2 : biu1_rmu1)
 : no_majority;

// Same thing for BIU2
formula vote2 =
   (biu2_rmu1=biu2_rmu2) ? ((biu2_rmu1!=bad_message) ? biu2_rmu1 : biu2_rmu3)
 : (biu2_rmu2=biu2_rmu3) ? ((biu2_rmu2!=bad_message) ? biu2_rmu2 : biu2_rmu1)
 : no_majority;


// ERROR MODEL FOR PE1
module pe1_error
  pe1_faulted: bool init false;
  pe1_error: [0..2] init fault_free;

  [] (!pe1_faulted) ->
        pe_benign:    (pe1_error'=benign_faulty) & (pe1_faulted'=true)
      + pe_symmetric: (pe1_error'=symmetric_faulty) & (pe1_faulted'=true)
      + pe_nofault:   (pe1_error'=fault_free) & (pe1_faulted'=true);

endmodule

// PE1
module pe1
  pe1_ctrl: [0..4] init 0;
  pe1_out: [0..7] init 0;

  // broadcast
  [] (pe1_ctrl=0) & (pe1_error=fault_free) -> (pe1_ctrl'=1) &
                                              (pe1_biu1' = source);
  [] (pe1_ctrl=0) & (pe1_error=benign_faulty) -> (pe1_ctrl'=1) &
                                                 (pe1_biu1'=bad_message);
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
                                                    (pe1_biu1'=data1);
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
                                                    (pe1_biu1'=data2);
  [] (pe1_ctrl=0) & (pe1_error=symmetric_faulty) -> (pe1_ctrl'=1) &
```

```
                                                    (pe1_biu1'=data3);

  [pe1_broadcast] (pe1_ctrl=1) -> (pe1_ctrl'=2);

  // wait for message from BIU1
  [biu1_send] (pe1_ctrl=2) -> (pe1_ctrl'=3);

  // consume the message: copy it into pe1_out
  [] (pe1_ctrl=3) -> (pe1_ctrl'=4) & (pe1_out' = pe1_biu1);

endmodule


// ERROR MODEL FOR BIU1
module biu1_error
  biu1_faulted: bool init false;
  biu1_error: [0..3] init fault_free;

  [] (!biu1_faulted) ->
        biu_benign :    (biu1_error' = benign_faulty) & (biu1_faulted'=true)
      + biu_symmetric : (biu1_error' = symmetric_faulty) &
                        (biu1_faulted'=true)
      + biu_asymmetric: (biu1_error' = asymmetric_faulty) &
                        (biu1_faulted'=true)
      + biu_nofault:    (biu1_error' = fault_free) & (biu1_faulted'=true);

endmodule


// BIU1
module biu1
  biu1_ctrl: [0..6] init 0;

  // wait for broadcast from pe1
  [pe1_broadcast] (biu1_ctrl=0) -> (biu1_ctrl'=1);

  // Fault-free case: consume the message and forward it to all rmus
  [] (biu1_ctrl=1) & (biu1_error=fault_free) & (pe1_biu1!=bad_message) ->
              (biu1_ctrl'=2) & (biu1_rmu1'=pe1_biu1) &
              (biu1_rmu2'=pe1_biu1) & (biu1_rmu3'=pe1_biu1);
  [] (biu1_ctrl=1) & (biu1_error=fault_free) & (pe1_biu1=bad_message) ->
              (biu1_ctrl'=2) & (biu1_rmu1'=pe_error) &
              (biu1_rmu2'=pe_error) & (biu1_rmu3'=pe_error);

  // Benign fault: send manifest bad message to all rmus
  [] (biu1_ctrl=1) & (biu1_error=benign_faulty) ->
              (biu1_ctrl'=2) & (biu1_rmu1'=bad_message) &
              (biu1_rmu2'=bad_message) & (biu1_rmu3'=bad_message);

  // Symmetric fault: send the same arbitrary message to all rmus
  [] (biu1_ctrl=1) & (biu1_error=symmetric_faulty) -> (biu1_ctrl'=2) &
              (biu1_rmu1'=data1) & (biu1_rmu2'=data1) & (biu1_rmu3'=data1);
  [] (biu1_ctrl=1) & (biu1_error=symmetric_faulty) -> (biu1_ctrl'=2) &
              (biu1_rmu1'=data2) & (biu1_rmu2'=data2) & (biu1_rmu3'=data2);
```

```
[] (biu1_ctrl=1) & (biu1_error=symmetric_faulty) -> (biu1_ctrl'=2) &
            (biu1_rmu1'=data3) & (biu1_rmu2'=data3) & (biu1_rmu3'=data3);

// Asymmetric fault: send different messages to the rmus
// TODO: add more combinations
[] (biu1_ctrl=1) & (biu1_error=asymmetric_faulty) -> (biu1_ctrl'=2) &
            (biu1_rmu1'=data1) & (biu1_rmu2'=data2) & (biu1_rmu3'=data3);
[] (biu1_ctrl=1) & (biu1_error=asymmetric_faulty) -> (biu1_ctrl'=2) &
            (biu1_rmu1'=data1) & (biu1_rmu2'=data2) & (biu1_rmu3'=data2);
[] (biu1_ctrl=1) & (biu1_error=asymmetric_faulty) -> (biu1_ctrl'=2) &
            (biu1_rmu1'=data1) & (biu1_rmu2'=data1) & (biu1_rmu3'=data2);
[] (biu1_ctrl=1) & (biu1_error=asymmetric_faulty) ->
            (biu1_ctrl'=2) & (biu1_rmu1'=bad_message) &
            (biu1_rmu2'=data1) & (biu1_rmu3'=data3);
[] (biu1_ctrl=1) & (biu1_error=asymmetric_faulty) -> (biu1_ctrl'=2) &
            (biu1_rmu1'=pe_error) & (biu1_rmu2'=data1) & (biu1_rmu3'=data2);

// Synchronization: allow RMUs to read the message
[biu1_broadcast] (biu1_ctrl=2) -> (biu1_ctrl'=3);

// wait for replay from all RMUs
[rmu_relay] (biu1_ctrl=3) -> (biu1_ctrl'=4);

// consume the three messages from the RMU and compute the
// majority
[] (biu1_ctrl=4) & (biu1_error=fault_free) ->
             (biu1_ctrl'=5) & (pe1_biu1'=vote1);

[] (biu1_ctrl=4) & (biu1_error=benign_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=bad_message);

[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data1);
[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data2);
[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data3);
[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=pe_error);
[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=source_error);
[] (biu1_ctrl=4) & (biu1_error=symmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=no_majority);

[] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data1);
[] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data2);
[] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=data3);
[] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=pe_error);
[] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
             (biu1_ctrl'=5) & (pe1_biu1'=source_error);
```

```
  [] (biu1_ctrl=4) & (biu1_error=asymmetric_faulty) ->
               (biu1_ctrl'=5) & (pe1_biu1'=no_majority);

  [biu1_send] (biu1_ctrl=5) -> (biu1_ctrl'=6);
endmodule


// RMU1
module rmu1
  rmu1_ctrl: [0..3] init 0;

  // Wait for data from BIU1
  [biu1_broadcast] (rmu1_ctrl=0) -> (rmu1_ctrl'=1);

  // consume the message and forward it to all BIUs
  [] (rmu1_ctrl=1) -> (rmu1_ctrl'=2) & (biu1_rmu1'=biu1_rmu1) &
                      (biu2_rmu1'=biu1_rmu1) & (biu3_rmu1'=biu1_rmu1);
  [rmu_relay] (rmu1_ctrl=2) -> (rmu1_ctrl'=3);

  // done
  //  [] (rmu1_ctrl=3) -> (rmu1_ctrl'=3);
endmodule

// RMU2 and RMU3 are like RMU1, modulo renaming
module rmu2 = rmu1[rmu1_ctrl=rmu2_ctrl,biu1_rmu1=biu1_rmu2,
                   biu2_rmu1=biu2_rmu2,biu3_rmu1=biu3_rmu2]
endmodule


module rmu3 = rmu1[rmu1_ctrl=rmu3_ctrl,biu1_rmu1=biu1_rmu3,
                   biu2_rmu1=biu2_rmu3,biu3_rmu1=biu3_rmu3]
endmodule


// BIU2
module biu2
  biu2_ctrl:[0..3] init 0;

  // Wait for data from RMUS
  [rmu_relay] (biu2_ctrl=0) -> (biu2_ctrl'=1);

  // consume the messages + vote
  [] (biu2_ctrl=1) -> (biu2_ctrl'=2) & (pe2_biu2'=vote2);
  [biu2_send] (biu2_ctrl=2) -> (biu2_ctrl'=3);

  // done
  // [] (biu2_ctrl=3) -> (biu2_ctrl'=3);
endmodule

// BIU3: same as BIU2 modulo renaming
module biu3 = biu2[biu2_ctrl=biu3_ctrl,biu2_send=biu3_send,
                   biu2_rmu1=biu3_rmu1,biu2_rmu2=biu3_rmu2,
                   biu2_rmu3=biu3_rmu3,pe2_biu2=pe3_biu3]
endmodule
```

```
// PE2
module pe2
  pe2_ctrl: [0..2] init 0;
  pe2_out: [0..7] init 0;

  // wait for message from BIU2
  [biu2_send] (pe2_ctrl=0) -> (pe2_ctrl'=1);

  // consume the message
  [] (pe2_ctrl=1) -> (pe2_ctrl'=2) & (pe2_out' = pe2_biu2);

  // done
  // [] (pe2_ctrl=2) -> (pe2_ctrl'=2);
endmodule

// PE3: same as PE2 modulo renaming
module p3 = pe2[pe2_ctrl=pe3_ctrl,pe2_out=pe3_out,
                biu2_send=biu3_send,pe2_biu2=pe3_biu3]
endmodule


label "done" = pe1_ctrl=4 & pe2_ctrl=2 & pe3_ctrl=2;

label "all_good" = (pe1_out=source)  & (pe2_out=source) & (pe3_out=source);

label "unanimous" = (pe1_out=pe2_out) & (pe1_out=pe3_out);

label "nonfaulty_agree" = (pe2_out=pe3_out);

label "nonfaulty_valid" = (pe2_out=source) & (pe3_out=source);
```

Properties for this model

```
// The broadcast protocol terminates with probability 1.
"init"=>P>=1 [ F "done" ]

P>=1 [ G ("done"=>"all_good") ]

P>=1 [ G ("done"=>"unanimous") ]

Pmin =? [ G("done" => "unanimous") ]

Pmax =? [G("done" => "unanimous")]

Pmin =? [G ("done" => "nonfaulty_agree")]

Pmax =? [G ("done" => "nonfaulty_agree")]

Pmin =? [G ("done" => "nonfaulty_valid")]

Pmax =? [G ("done" => "nonfaulty_valid")]
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 05 - 2011	Contractor Report	

4. TITLE AND SUBTITLE
Probabilistic Analysis of Distributed Fault-Tolerant Systems

5a. CONTRACT NUMBER
NNL10AB32T

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dutertre, Bruno

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
534723.02.02.07.30

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR-2011-217090

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
This report was prepared by SRI International under subcontract to Honeywell International, Inc., Minneapolis, MN, under NASA contract NNL10AB32T.
Langley Technical Monitor Paul S. Miner

14. ABSTRACT

An approach is documented for analyzing probabilistic properties of fault-tolerant distributed systems using the PRISM model checker.

15. SUBJECT TERMS
fault models, fault tolerance, markov models, probalistic model checking, reliability analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	49	(443) 757-5802