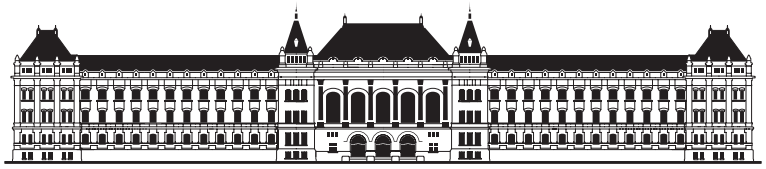

Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group

Critical Architectures Laboratory
Spring Semester 2018/2019

Dependability Modeling

Syllabus and Exercises
v0.11

Authors: Kristóf Marussy
(marussy@mit.bme.hu)
Attila Klenik
(klenik@mit.bme.hu)
András Vörös
(vori@mit.bme.hu)

March 6, 2019

Contents

1 Introduction

In addition to functional requirements, systems and their architectures must also satisfy numerous extra-functional requirements. *Dependability*, i.e., the ability to provide service in which reliance can be justifiably placed is one of the most significant categories of such requirements. In IT services Service Level Agreements (SLA) capture the desired dependability attributes, while safety-critical systems must not exceed the Tolerable Hazard Rates (THR) prescribed by standards.

Stochastic modeling is a mathematically precise approach to reason about the dependability attributes of the design alternatives of systems.

In this syllabus, we will briefly review some concepts in formalisms for dependability modeling, and the analysis of stochastic models. We also introduce the tools needed for the laboratory. In the last section, students may find the exercises.

Before the laboratory, students are advices to review the related materials of the Systems Engineering, Software and Systems Verification, and Formal Methods courses.

2 Modeling formalisms and tools

When ensuring the satisfaction of extra-functional requirements in systems engineering, we can calculate dependability measures either by *simulation* or *analytic solution* of stochastic models. The models capture the dependability attributes of the architecture of the system.

Simulation supports a broader range of behaviors. For example, it can easily incorporate arbitrary probability distributions. However, the interpretation of simulation results requires care, because we must ensure that the simulation was run for the appropriate number of cases and amount of time.

Analytic solutions always provide accurate answers up to user-specified tolerance. To make the analysis tractable, solvers typically handle more constrained stochastic modeling languages than simulators.

In the next subsections, we discuss two commonly used stochastic modeling formalisms that are especially amenable for analytic solutions. We also illustrate the use of the Storm probabilistic model checker on these formalisms.

After introducing our running example, we turn to *Fault Trees* (FT) [Sta+02] for reliability analysis. Fault trees are a non-state-based formalism, which means we cannot explicitly model the state of system components other than their failures. This assumption greatly simplifies their analysis.

The Storm toolset also supports *Dynamic Fault Trees* (DFT), which enable some stateful modeling, for example, cold, warm, and hot spares. Analysis with such constrained state spaces is possible at a modest additional complexity.

In contrast, *Continuous-Time Markov Chains* (CTMC) are state-based models. Instead of describing the system with a state graph of state nodes and transition edges, we use the more compact representation offered by the PRISM language. Hence we will able to model state as variables and transitions as programming language statements acting on the variables.

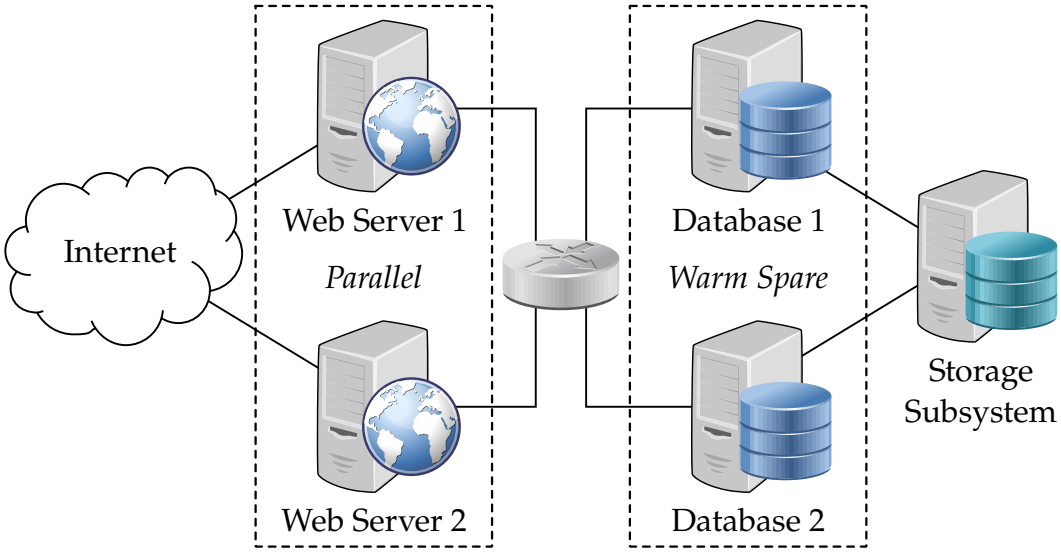

Figure 1: Example web service infrastructure

2.1 Running example

We illustrate reliability modeling with fault trees and Markov chains using the web service infrastructure in Fig. 1.

The example infrastructure contains 2 web servers, 2 database servers, and a shared storage subsystem. The web servers are in a parallel redundant configuration, i.e., both web servers handle requests at the same time unless one of them has failed. Database Server 2 is a warm spare for Database Server 1. If the first server is operating correctly, no queries are directed to the second one. We will not examine the redundancies in the storage subsystem and will instead consider it as a basic unit.

The dependability attributes of the components are as follows:

Component	**Failure rate**	**Dormancy factor**
Web Server	$0.05\ \frac{1}{\text{day}}$	1.0
Database	$0.01\ \frac{1}{\text{day}}$	0.8
Storage Subsystem	$0.03\ \frac{1}{\text{day}}$	1.0

Table 1: Dependability attributes for the running example

The *domancy factor* of the database server reduces the failure rate of the warm spare database. When a database is not currently responding to queries its failure rate is only $0.8 \cdot 0.01\ \frac{1}{\text{day}}$.

2.2 Dynamic fault trees

Dynamic fault trees are a generalization of fault trees. In addition to basic events and standard Boolean logic gates, they offer *priority gates*, *sequence enforcers*, *spare gates* and *functional dependencies* to express some state-based fault tolerance strategies.

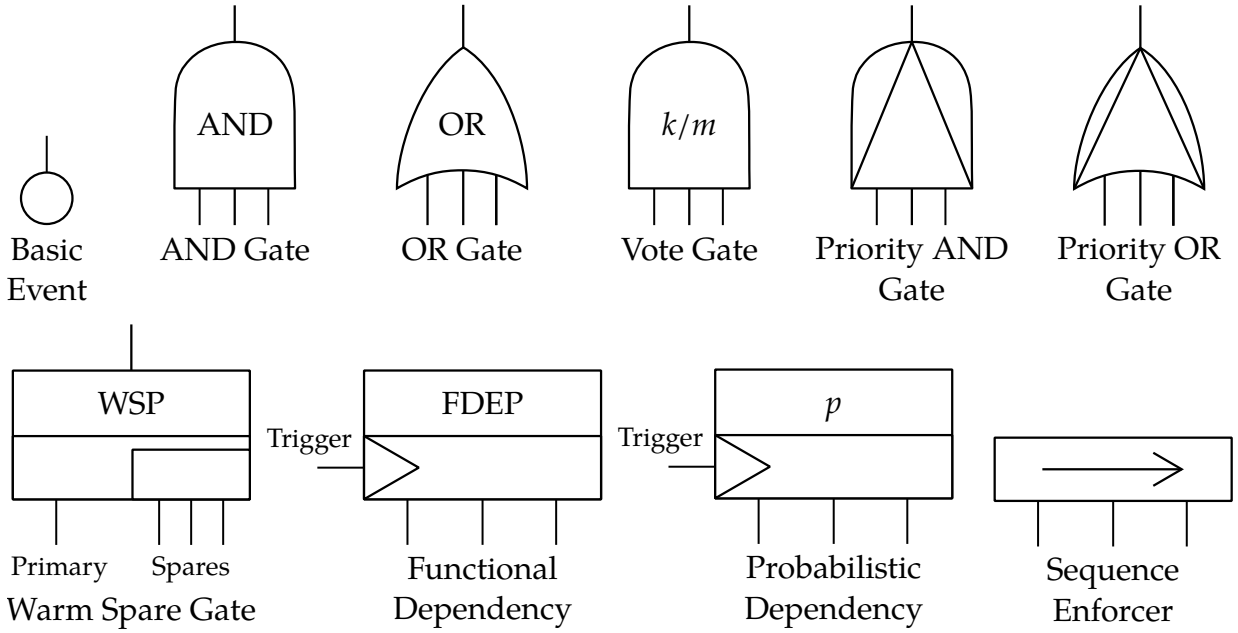

Figure 2: Symbology for dynamic fault tree gates

2.2.1 Elements and notation

In this laboratory, we will use the textual Galileo[1] language to describe dynamic fault trees. Fig. 2 shows the graphical notations corresponding to DFT elements.

The *top level event* (Galileo: **toplevel** ‹*name*›;) correspond to whole system failure. This event can be decomposed into failures of elementary components, which are called *basic events*, by a series of *gates*. The basic events and gates supported by the Storm-DFT[2] [VJK16] tool, which we will use in the laboratory, are as follows:

- **Basic events** (Galileo: ‹*name*› **lambda**=‹λ› **dorm**=‹*dormancy*›;) describe the independent failures that may occur in the system. The time first failure since mission start is exponentially distributed with a rate of λ, i.e., $\mathbb{P}(T_{fail} < t) = 1 - e^{-\lambda t}$. When the basic event is serving as a warm spare, its failure rate is reduced to $dormancy \cdot \lambda$. The dormancy factor may be omitted if it is equal to 1.
- **AND gates** (Galileo: ‹*name*› **and** ‹$input_1$› ‹$input_2$› ...;) fail if all their inputs fail.
- **OR gates** (Galileo: ‹*name*› **or** ‹$input_1$› ‹$input_2$› ...;) fail if any of their inputs fail.
- **Vote gates** (Galileo: ‹*name*› ‹*k*›**of**‹*m*› ‹$input_1$› ‹$input_2$› ...;) must have exactly m inputs. They fail if at least k of their m inputs fail. If one wishes to avoid specifying the number of inputs, alternative textual notation **vot**‹*k*› is available.
- **Priority AND gates** (Galileo: ‹*name*› **pand** ‹$input_1$› ‹$input_2$› ...;) fail if all their inputs fail in the specified order, i.e., $input_2$ must fail after $input_1$. By default, priority AND gates are inclusive, and fail when $input_1$ and $input_2$ fail simultaneously. If desired, inclusiveness can be explicitly specified by the **pand-inc** keyword. The **pand-ex** keyword specifies

[1] https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm#Editing%20in%20the%20Textual%20View

[2] http://www.stormchecker.org/documentation/usage/running-storm-on-dfts.html

exclusive priority AND gates, which survive simultaneous failure. Inclusive and exclusive gates can be graphically distinguished with the ≤ and < symbols, respectively.

Most commonly, priority AND gates represent failure avoidance strategies. If the failure avoidance strategy connected to $input_1$ is working correctly, failure of $input_2$ does not affect the rest of the system, but if $input_1$ fails first, the failure of $input_2$ are let through. Inclusive and exclusive variation can specify the response of the gates to common-mode failures of the avoidance strategy and the monitored component.

- **Priority OR gates** (Galileo: ‹*name*› **`por`** ‹$input_1$› ‹$input_2$› ... ;) only fail if $input_1$ fails before any other input. By default, priority OR gates are inclusive, i.e., fail if $input_1$ fails simultaneously with some other input. Inclusive and exclusive priority OR gates can be specified with the **`por-inc`** and **`por-ex`** keywords, respectively.

- **Warm spare gates** (Galileo: ‹*name*› **`wsp`** ‹*primary*› ‹$spare_1$› ‹$spare_2$› ... ;) maintain a pool of spares. When the primary input fails, its first free spare is *claimed*. When the claimed spare fails, the gate claims a new one (from left to right) until the pool of spares is exhausted. If the last spare fails, the gate itself fails. Spares assigned to a spare gate may not have any common events, but a spare can be assigned to multiple spare gates. However, a spare can only be claimed by at most one spare gate at a time. Thus we can model contention between subsystems for shares spares. Rates of basic events in unclaimed spares are multiplied by their dormancy factors.

  While Storm-DFT accepts the keywords **`csp`** and **`hsp`** for cold and hot spares, respectively, they are treated as synonyms for warm spares.

- **Functional dependencies** (Galileo: ‹*name*› **`fdep`** ‹*trigger*› ‹$input_1$› ‹$input_2$› ... ;) cause their inputs to fail (provided they have not already failed) when their trigger event fails. Every input except the trigger must be a basic event. The outputs of functional dependencies are "dummy outputs", because the event *name* will never actually fail.

- **Probabilistic dependencies** (Galileo: ‹*name*› **`pdep`**=‹*p*› ‹*trigger*› ‹$input_1$› ‹$input_2$› ... ;) act similarly to functional dependencies. However, their inputs only fail with probability p the failure of the trigger. The probabilistic dependency has a single underlying random choice. If it causes one of its inputs to fail, all of them will fail, too.

- **Sequence enforcers** (Galileo: ‹*name*› **`seq`** ‹$input_1$› ‹$input_2$› ... ;) ensure that their inputs only fail in the specified order, i.e., $input_2$ will never fail unless $input_1$ has already failed. Similarly to the dependencies, their outputs are "dummy".

The fault tree must not contain any loops, because an event cannot transitively depend on itself. However, limited forms of circular dependencies can be introduced with functional and probabilistic dependencies [Sta+02, Fig. 8-7]. It is not necessary to connect "dummy" outputs to the tree in Storm-DFT. However, they may be connected to an OR gate without a change in semantics.

We recommend choosing meaningful names for event and gates for two reasons as follows: Firstly, the names aid in the comprehension of the fault tree. Secondly, single-character identifiers may cause a puzzling parse error in Galileo models. Pay attention when referring to fault tree elements, because unresolved references, e.g., typos in names, may lead to silent failures and corrupt the analysis output.

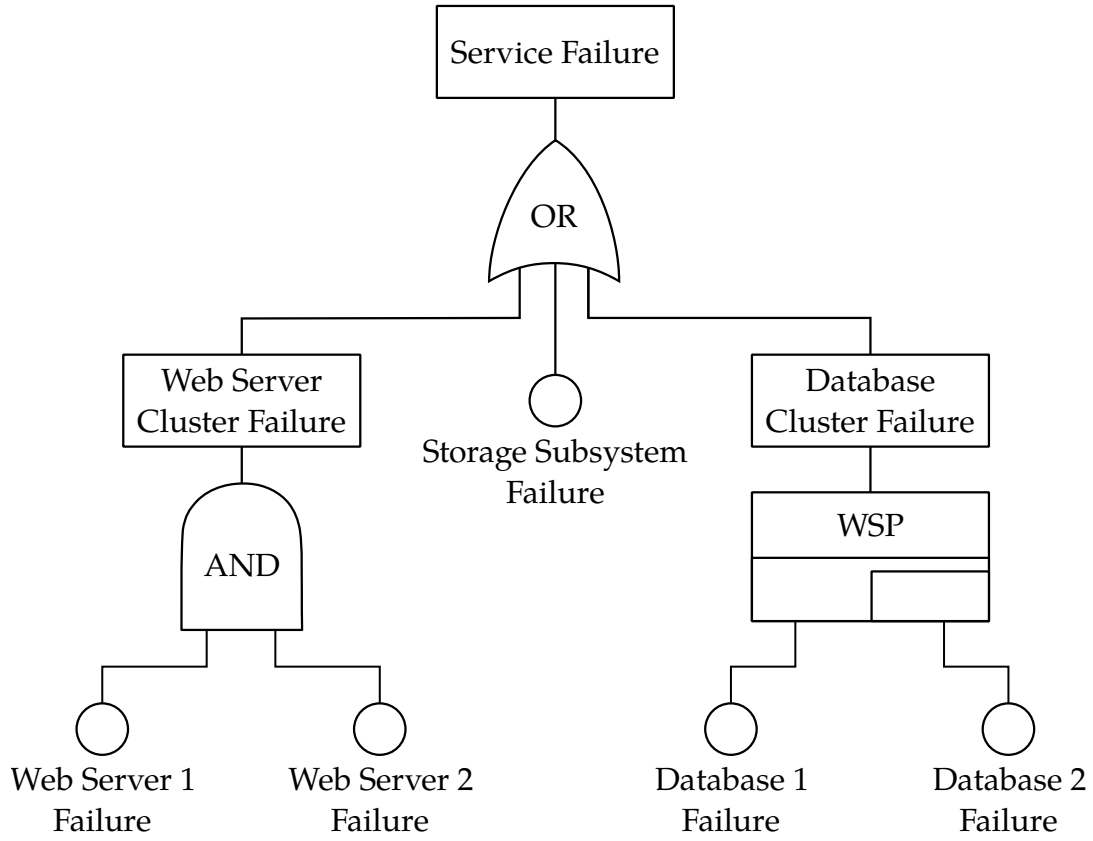

Figure 3: Example web service fault tree

2.2.2 Fault tree construction

As an example, we construct a dynamic fault tree model for the system presented in Section 2.3.1. The reader is invited to follow along in Fig. 3, which shows the finished model using the graphical syntax of DFTs (see Fig. 2 for a legend). For a more thorough introduction to fault tree construction, we refer the interested reader to the presentation by Ericson [Eri99].

First, we start with the top level failure event "Service Failure". Events on the second level of the tree, which correspond to subsystem level failures, are connected with an OR gate. Either of those subsystem-level failures triggers system failure.

The event "Web Server Cluster Failure" is decomposed into the two basic events "Web Server 1 Failure" and "Web Server 2 Failure" by an AND gate corresponding to parallel redundancy.

The event "Database Cluster Failure" use decomposed using a warm spare gate. "Database Server 1 Failure" is the primary event, while "Database Server 2 Failure" serves as a spare. Until the gate claims the spare, its failure rate is multiplied by the dormancy factor.

"Storage Subsystem Failure" is a single point of failure (SPOF) of the service.

Cut sets of the fault tree are sets of basic events the simultaneous failures of which cause the top level event to fail. *Minimal cut sets* are cut sets no proper subsets of which are cut sets. In Fig. 3, minimal cut sets are as follows:

- {Web Server 1 Failure, Web Server 2 Failure}
- {Database 1 Failure, Database 2 Failure}

- {Storage Subsystem Failure}

We may see that a SPOF is a minimal cut set of exactly one element.

We can encode the DFT using the Galileo language as follows:

```
toplevel service;
service or web_cluster database_cluster storage; // Service Failure
web_cluster and web_1 web_2;                     // Web Server Cluster Failure
web_1 lambda=0.05;                               // Web Server 1 Failure
web_2 lambda=0.05;                               // Web Server 2 Failure
database_cluster wsp database_1 database_2;      // Database Cluster Failure
database_1 lambda=0.01 dorm=0.8;                 // Database 1 Failure
database_2 lambda=0.01 dorm=0.8;                 // Database 2 Failure
storage lambda=0.03;                             // Storage Subsystem Failure
```

Listing 1: Example Galileo model

The usual file extension for Galileo models is `dft`. If the following, we will assume that the code in Listing 1 is saved as `example.dft`.

2.2.3 Analytic solution

Mean-time-to-first-failure (MTFF) analysis determines the expected time of failure of the top level event of the DFT. Formally, let the random variable $T_{toplevel}$ denote the time-to-first-failure (TFF) of the top level event. Then the MTFF is the expected value $\mathbb{E}[T_{toplevel}]$.

We can perform MTFF analysis by invoking `storm-dft` with the `--expectedtime` option. The `--dftfile` option specifies the path of the DFT.

```
$ storm-dft --dftfile example.dft --expectedtime
Storm-dft 1.3.1 (dev)

Date: Tue Feb 26 17:54:58 2019
Command line arguments: --dftfile example.dft --expectedtime
Current working directory: /home/meres

-------------------------------------------------------------
Model type:     CTMC (sparse)
States:         10
Transitions:    22
Reward Models:  none
State Labels:   2 labels
   * failed -> 1 item(s)
   * init -> 1 item(s)
Choice Labels:  none
-------------------------------------------------------------
Times:
Exploration:    0.007s
Building:       0.000s
Bisimulation:   0.000s
Modelchecking:  0.006s
```

```
Total:          0.014s
Result: [16.86431329]
```

Listing 2: MTFF analysis with Storm-DFT

Strorm-DFT has determined that the MTFF is 16.86431329 days. The unit of days comes from Listing 1, where we specified the failure rates in $\frac{1}{\text{day}}$. In general, the unit of MTFF is always the reciprocal of the unit of the failure rates. Pay special attention that each failure rate is specified with the same unit, otherwise the analysis results with be incorrect.

The answer is accurate up to the relative numerical precision of the DFT solver, which is set to 10^{-6} by default. This means the true MTFF of the system is between $(1 - 10^{-6}) \cdot \text{MTFF}$ and $(1 + 10^{-6}) \cdot \text{MTFF}$. The precision can be changed by the `--precision` option, for example, passing `--precision 1e-8` on the command line sets the precision to 10^{-8}.

Another possible analysis determines the probability of failure at specific point of time t since mission start. Formally, the reliability function $r(t) = 1 - \mathbb{P}(T_{toplevel} < t)$ captures the probability of correct operation.

Let us determine the reliability at $t = 10$ days. In Storm-DFT, the `--timebound` option initiates reliability analysis.

```
$ storm-dft --dftfile example.dft --timebound 10.0
Storm-dft 1.3.1 (dev)

Date: Tue Feb 26 18:08:48 2019
Command line arguments: --dftfile example.dft --timebound 10.0
Current working directory: /home/meres

---------------------------------------------------------------
Model type:     CTMC (sparse)
States:         10
Transitions:    22
Reward Models:  none
State Labels:   2 labels
   * failed -> 1 item(s)
   * init -> 1 item(s)
Choice Labels:  none
---------------------------------------------------------------
Times:
Exploration:    0.001s
Building:       0.000s
Bisimulation:   0.000s
Modelchecking:  0.007s
Total:          0.009s
Result: [0.3790103367]
```

Listing 3: Reliability analysis with Storm-DFT

The output is the probability of failure. Thus $r(10\,\text{days}) \approx 1 - 0.3790103367$ up to the relative precision of 10^{-6}. Pay attention again to the units. The time bound passed on the command line is interpreted in days, because failure rates were specified as $\frac{1}{\text{days}}$.

To determine the reliability at multiple, evenly spaced points of time, we may use the

`--timepoints` ‹*start*› ‹*end*› ‹*increment*› command line option.

For further information on the supported command line options of `storm-dft`, please run the command **`storm-dft --help`**. We especially recommend reading through the `general`, `dft`, and `dftIO` sections of manual.

2.3 Continuous-time Markov chains

Markov chains are state-based stochastic models, which are popular in dependability and performability evaluation. Compared to fault trees, they make relatively few assumptions about the behavior of the system. In particular, failure processes (transitions) can only depend on the active state of the systems, but not on the past trajectory of the system, or the time spent so far in the active state.

In this laboratory, we will use the PRISM language for describing CTMCs. In contrast with the low-level specification of stochastic models in terms of individual states and transitions, PRISM enables a concise description with variables and commands that update them. Thus we can write stochastic models as if they were ordinary, imperative programs.

2.3.1 Example model

The PRISM model for the web service from Section illustrates below some of concepts of the PRISM language. We will also highlight some features specific to the Storm model checker.

For a complete description of the language, we refer to the PRISM Manual[3], especially to the section on CTMCs[4].

```
ctmc

const int web_cluster_size = 2;
const int db_cluster_size = 2;
const double web_fail = 0.05;
const double db_fail = 0.01;
const double dorm = 0.8;
const double storage_fail = 0.03;

module web_cluster
    web_up : [0..web_cluster_size] init web_cluster_size;
    // Failure rates of parallel redundant web servers are summed.
    <> web_up > 0 -> web_up * web_fail : (web_up' = web_up - 1);
endmodule

module db_cluster
    db_up : [0..db_cluster_size] init db_cluster_size;
    // The dormancy factor reduces the failure rates of spare database servers.
    <> db_up > 0 -> db_fail * (1 + (db_up - 1) * dorm) : (db_up' = db_up - 1);
endmodule
```

[3] http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction
[4] http://www.prismmodelchecker.org/manual/ThePRISMLanguage/CTMCs

```
module storage
    storage_up : bool init true;
    <> storage_up -> storage_fail : (storage_up' = false);
endmodule

// The system fails if one of the modules fails entirely.
formula failed = (web_up = 0 | db_up = 0 | !storage_up);

// A single unit of reward is gained per unit time in every state.
rewards "elapsed"
    true : 1.0;
endrewards
```

Listing 4: Example PRISM model

The `ctmc` keyword in the first line denotes that we are describing a CTMC. If we omit it, PRISM interprets the model as a Markov Decision Process (MDP) instead.

Constant declarations introduced with the `const` keyword can hold common parameters.

The model consists of modules, which are defined within `module`. . . `endmodule` blocks. Modules contain *local variables* and *guarded commands.*

Local variables can be read by any module, but can be only modified by their containing module. Variable definitions are of the form ‹*name*› : ‹*type*› `init` ‹*initial*›;, where *type* may be either a bounded integer type [‹*lower*›. . .‹*upper*›] (both bounds are inclusive) or the Boolean type `bool`. In the initial state of the CTMC, the variable is initialized to the value *initial.*

Markovian commands are of the form <> ‹*guard*› -> ‹*rate*› : (‹*var*›' = ‹*value*›);. If *guard* is satisfied in the current state of the CTMC, a transition with rate *rate* is added to the state obtained by setting the local variable *var* to *value*. The prime sign (') denotes the assignment updates the variable in the next state of the CTMC. The expression *value* can refer to the current (old) state of the variable without the prime sign. Multiple assignments can be combined by the & operator, e.g., (‹var_1›' = ‹$value_1$›) & (‹var_2›' = ‹$value_2$›). Transitions with the same guard, e.g., <> ‹*guard*› -> ‹$rate_1$› : ‹$assignments_1$› + ‹$rate_2$› : ‹$assignments_2$›;, can be also combined with the + operator.

In PRISM compatibility mode, which is enabled with the `--prismcompat` command line option, the Storm tool also accepts Markovian commands in CTMCs written as *probabilistic commands.* These commands are introduced by the [] symbols instead of <>. Optionally, they can have a label, which is written as [‹*label*›]. The PRISM manual relies on this alternative notation. However, we do not recommend it, because the notation preferred by Storm emphasizes the difference between CTMCs and discrete-time model more.

Formula definitions, which are introduced by the `formula` keyword, hold common expression that can be reused throughout the model and its properties.

Rewards structures associate *reward rates* to the states of the CTMC. The amount of reward gained (or cost spent) is a state equals the reward rate multiplied by the amount of time spend in the state. Reward structures are defined within `rewards`. . . `endrewards` blocks. While names of modules, variables, and formulas are unquoted, rewards names must be surrounded by double quotes. Each reward structure can have multiple reward elements, which are written as ‹*guard*› : ‹*value*›;. In states where the predicate *guard* holds, *value* amount of reward is gained per unit of time. Values for reward elements with overlapping guards are summed.

The usual file extension for CTMCs described using the PRISM language is `.sm` (Stochastic Model). In the following we will assume the model from Listing 4 is saved as `example.sm`.

2.3.2 Reward analysis

In this laboratory, we will perform reward analysis using the Storm probabilistic model checker[5] [Deh+17], a model model checker for probabilistic systems. Storm supports a wide range of algorithms, formalisms, and languages, including DFTs and the PRISM language.

Queries to be answered by analytic solutions of PRISM models can be expressed using the PRISM property specification language. We recommend reading through the relevant parts of the PRISM manual[6], especially the section on reward-based properties[7].

Reward-based queries are of the form `R{`*‹reward›*`}=?[`*‹prop›*`]`, where *reward* is a reward structure defined in the model (see Section 2.3.1) and *prop* is a reward property. Some examples of the supported reward properties are as follows:

- **Reachability rewards** "`F` *‹pred›*" calculate the expected accumulated (integrated) reward until a state satisfying the predicate *pred* is reached.
- **Cumulative rewards** "`C <=` *‹time›*" calculate the expected accumulated reward until *time* units of time elapse since starting the model from its initial state.
- **Instantaneous rewards** "`I =` *‹time›*" calculate the expected reward rate *time* units of time after starting the model from its initial state.

  Notice that $\int_0^t$ `R{"rew"}=?[I =` τ`]` $d\tau$ = `R{"rew"}=?[C <=` t`]`, i.e., the cumulative reward can be obtained by integrating the instantaneous reward.
- **Steady-state rewards** "`S`" calculate the expected reward rate in steady state of the model, when a sufficient amount of time has allowed the transient behaviors to pass. We may write $\lim_{t\to\infty}$ `R{"rew"}=?[I =` t`]` = `R{"rew"}=?[S]`, i.e., the steady-state reward is the limit of the instantaneous reward.

As an example, we may calculate the MTFF in Listing 4 by accumulating `"elapsed"`, the reward rate of which equals 1.0 in every state, until the formula `failed` holds. This query can be written as `R{"elapsed"}=?[F failed]`.

PRISM properties can be passed to Storm using the `--prop` command line option.

```
$ storm --prism example.sm --prop 'R{"elapsed"}=?[F failed]'
Storm 1.3.1 (dev)

Date: Fri Mar  1 00:56:22 2019
Command line arguments: --prism example.sm --prop 'R{"elapsed"}=?[F failed]'
Current working directory: /home/meres

Time for model input parsing: 0.000s.
```

[5] http://www.stormchecker.org/
[6] http://www.prismmodelchecker.org/manual/PropertySpecification/Introduction
[7] http://www.prismmodelchecker.org/manual/PropertySpecification/Reward-basedProperties

```
Time for model construction: 0.012s.

-------------------------------------------------------------
Model type:     CTMC (sparse)
States:         12
Transitions:    20
Reward Models:  elapsed_time
State Labels:   3 labels
   * deadlock -> 0 item(s)
   * (((web_up = 0) | (db_up = 0)) | !(storage_up)) -> 8 item(s)
   * init -> 1 item(s)
Choice Labels:  none
-------------------------------------------------------------

Model checking property "1": R[exp]{"elapsed"}=? [F (((web_up = 0) | (db_up = 0)
   ) | !(storage_up))] ...
Result (for initial states): 16.86431329
Time for model checking: 0.000s.
```

Listing 5: Reliability analysis with Storm

Storm has determined that the MTFF of the system is 16.86431329 days, which matches our results from Section 2.2.3. Considerations from that section regarding the careful choice of units and the relative numerical tolerance of the solver (10^{-6} by default) apply here, too.

Occasionally, we may specify a property that cannot be handled by the default analysis engine of Strorm. In this case, an alternative equation solver may be selected by passing the `--eqsolver` option. For example, `--eqsolver eigen` sets Eigen[8] as the equation solver.

For further information on the supported command line options of `storm`, please run the command **`storm --help`**. We especially recommend reading through the `general`, `io`, and `build` sections of manual.

3 Exercises

In this laboratory we will study the construction and analysis of stochastic models for evaluating the dependability of IT systems.

To complete this laboratory, you will need

- the Storm probabilistic model checker, version 1.3.1-dev, and its Storm-DFT tool,
- a text editor for creating model files,
- a spreadsheet program or a scripting environment (e.g., Python with `matplotlib`) for basic calculations and plotting.

We installed Strom 1.3.1-dev, various text editors, and Python to the virtual machine available from the course homepage[9]. As a spreadsheet program, we recommend online office suites, such as Google Sheets.

[8] http://eigen.tuxfamily.org/index.php?title=Main_Page

[9] https://inf.mit.bme.hu/edu/courses/kalab

For installing Storm to your own computer, see the Strom documentation[10]. Storm does not presently support Windows. It must be built from source on Linux and OS X, which may take a long time and may sometimes require manual intervention. Therefore, if you want to use Storm on your own physical machine, please compile and install it in advance.

3.1 Documentation requirements

We kindly ask students to document every design decision, if any, necessitated by the lack of information in the specification or the exercises.

Similarly to other sessions, please upload the documentation in Markdown to the GitHub Wiki of your repository. Any supplementary files, including stochastic models, may be placed into a directory named `dependability-analysis` in the repository itself.

The documentation should contain, at the very least,

- stochastic models (`.dft` and `.sm` files) produced for each task,
- (informal) descriptions of the models and their corresponding design decisions,
- results of analytic solutions of models, as well as and other computations,
- evaluation and explanation of the results.

After completing the laboratory, feel free to include any comments on the tasks and the syllabus. Your comments will aid us greatly in improving the session for future iterations.

3.2 Case study

The system shown in Fig. 4 will serve as our case study throughout the laboratory.

The case study consists of two web servers (Web-1, Web-2), two domain controllers (DC-1, DC-2), a database server (SQL), and an application server (App). The servers are connected to a network switch (Sw). The web servers are in a parallel redundant configuration. Domain controller DC-1 is the primary one, while DC-2 is a warm spare.

We will assume that failure and repair times are exponentially distributed. The dependability attributes of components are as follows:

Component	Mean time to failure	Mean time to repair	Dormancy factor
Web server (Web)	20 days	24 hours	0.8
Domain controller (DC)	25 days	24 hours	0.8
Database server (SQL)	30 days	48 hours	1.0
Application server (App)	35 days	72 hours	1.0
Network switch (Sw)	60 days	12 hours	1.0

Table 2: Dependability attributes for the case study

[10] `http://www.stormchecker.org/documentation/installation/installation.html`

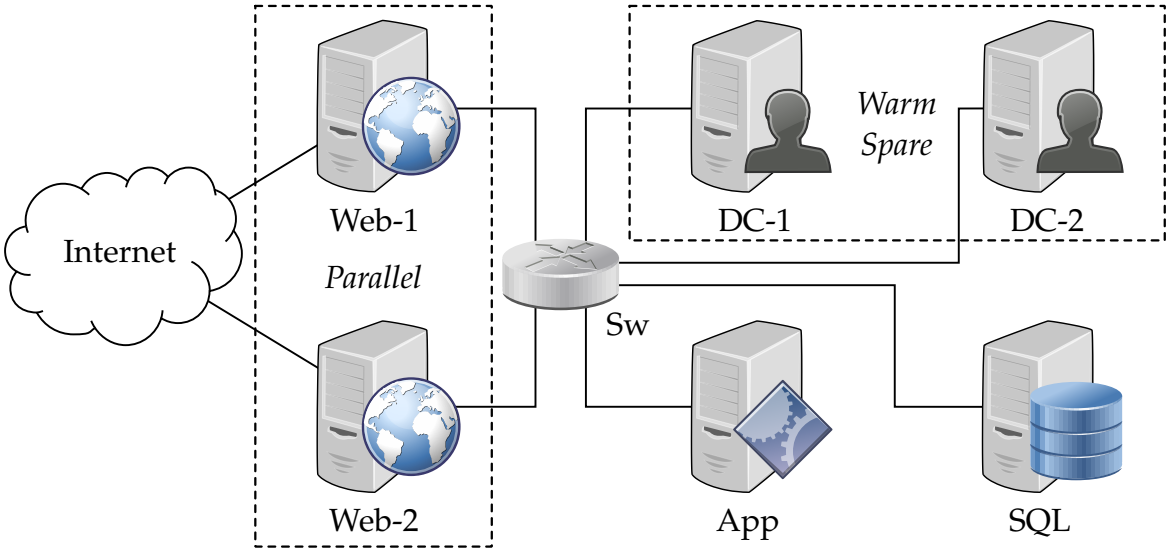

Figure 4: Architecture of the case study

3.3 Dynamic fault tree tasks

1. Create a DFT model for case study from Section 3.2.
2. What are the minimal cut sets of the DFT? What are the single points of failure?
3. Determine the MTFF of the system.
4. Determine the values of the reliability function $r(t) = 1 -$ probability of failure at time t for $t = 1, 2, \ldots, 30$ days. Present the values on a plot, where the x axis is time, and the y axis is the probability of correct operation.
5. Modify your model so that the web servers are put into a warm spare configuration instead of parallel redundancy. How does the MTFF change?
6. Modify the DFT to allow the spare domain controller also be used as a web server. Hence, there should be two spare web servers, one of which may also be a spare domain controller. How does the MTFF depend on the order in which the spares for Web-1 are claimed? Which order should be used in production?
7. The spare domain controller is activated by fault tolerance agents that are replicated across the primary web server, the application server and the SQL server. When two of the three replicas vote for enabling the spare domain controller, it takes over the responsibilities of the failed primary controller. The fault tolerance agents have a failure rate of $0.003\,\frac{1}{\text{day}}$. Agents also fail if their host server fails. You do not have to model similar agents for the spare web servers. Determine the MTFF of the system.

3.4 Continuous-time Markov chain tasks

8. Create a CTMC reliability model for case study from Section 3.2, ignoring any modifications to the system in Section 3.3. Use PRISM modules to structure you model.
9. Determine the MTFF of the system. Check whether the results match DFT analysis.

10. Extend your model to obtain an availability model by adding component repairs. What is the steady-state availability of the system?

11. The daily cost of operation is $5 for a web server, $6 for a domain controller and $3 for a spare domain controller. Determine the total cost for the first 10 days of operation, as well as the steady-state hourly cost.

12. How does the steady-state availability and cost change if the number of web servers is increased to 3? How do the metrics change if we increase the number of domain controllers to 3 instead? All new domain controllers are added as warm spares. If you could add only a single server to the system, which modification would you prefer?

Acknowledgement

Emberi Erőforrások Minisztériuma

Version 0.11 of this document was supported by the ÚNKP-18-3 New National Excellence Program of the Ministry of Human Capacities.

References

[Deh+17] Christian Dehnert et al. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *CAV 2017*. LNCS 10427. Springer, 2017, pp. 592–600. DOI: 10.1007/978-3-319-63390-9_31. arXiv: 1702.04311 [cs.SE].

[Eri99] Clifton A. Ericson II. "Fault Tree Analysis". In: *COP 4331 - Processes for Object-Oriented Software Development*. University of Central Florida, 1999. URL: http://cs.ucf.edu/~hlugo/cop4331/ericson-fta-tutorial.pdf.

[Sta+02] Michael Stamatelatos et al. "Dynamic Fault Tree Analysis". In: *Fault Tree Handbook with Aerospace Applications*. Version 1.1. NASA Office of Safety and Mission Assurance, NASA Headquarters, 2002, pp. 97–108. URL: https://kscddms.ksc.nasa.gov/Reliability/Documents/Fault_Tree_Handbook_with_Aerospace_Applications_August_2002.pdf#page=110.

[VJK16] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. "Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates". In: *SAFECOMP 2016*. LNCS 9922. Springer, 2016, pp. 253–265. DOI: 10.1007/978-3-319-45477-1_20. arXiv: 1604.07474 [cs.SE].