VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

TVORBA SPOLEHLIVOSTNÍCH MODELŮ PRO POKROČILÉ DIGITÁLNÍ SYSTÉMY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARIO WANKA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

TVORBA SPOLEHLIVOSTNÍCH MODELŮ PRO POKROČILÉ DIGITÁLNÍ SYSTÉMY

CONSTRUCTION OF RELIABILITY MODELS FOR ADVANCED DIGITAL SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE AUTHOR	MARIO WANKA
VEDOUCÍ PRÁCE SUPERVISOR	Ing. JAN KAŠTIL

BRNO 2015

Abstrakt

Cílem této práce je simulovat vliv spolehlivosti obvodů detekujících chybu u komponent pokročilých digitálních systémů. Prvně je definována spolehlivost a skutečnosti s ní související a jsou představeny Markovské modely. Tyto jsou využity pro samotný simulátor, který je představený v následující kapitole. Jedná se o ad-hoc řešení a použití tohoto simulátoru je detailně popsáno. Stejně tak je popsáno jeho chování v průzných situacích a s různou konfigurací. Na závěr jsou ukázány a diskutovány výsledky experimentů se spolehlivostí obvodů detekujících chybu pro různé modely. Dle výsledků práce je zřejmé, že zásadním faktorem pro zajištění spolehlivosti systému je krátkodobé maskování chyby a dlouhodobé udržení opravovatelnosti.

Abstract

The goal of this thesis is to simulate the impact of reliability of circuits designed to detect failures in components of advanced digital systems. At first the reliability and terms related are defined and Markov models are itroduced. These are used as logic for simulator, which is introduced in next chapter. This simulator is an ad-hoc solution and it's usage is thoughly described, as well as it's bahviour in various situations and configurations. In the end the results of experiments with reliability of circuits designed to detect failures for multiple models are shown and discussed. By the results of this thesis it is obvious, that the crutial factor for system's reliability ensurance is failure disguis in short-term view and repairing ability in long-term view.

Klíčová slova

Spolehlivost, Markovský model, Systémy zatížené chybou, Stochastický simulační algoritmus, Pravděpodobnost bezporuchového provozu

Keywords

Reliability, Markov model, systems with faults, Stochastic Simulation Algorithm, Probability of failure-free service

Citace

Mario Wanka: Tvorba spolehlivostních modelů pro pokročilé digitální systémy, bakalářská práce, Brno, FIT VUT v Brně, 2015

Tvorba spolehlivostních modelů pro pokročilé digitální systémy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Mario Wanka
18. května 2015

Poděkování

Děkuji vedoucímu své práce za jeho trpělivost, ochotu se kterou přistupoval ke konzultacím a celkový zájem o práci.

Contents

Chapter 1

Introduction

The question of reliability is a crucial aspect of any system in existence. Even a - in theory - flawlessly designed system is no use when we cannot rely on results which may be corrupted by faults of components. So what are the ways to achieve reliability and what is the cost of it? Possible approach is to use redundancy. But with three, five, etc. times more components, the price of such system will rise the same way. Another problem is to find out how strong redundancy is enough, especially considering systems running for years long. This is an ideal case for simulation to take place.

The most precise way to measure reliability is, of course, on the operational system. This is no problem with cheap systems whose failure causes no difficulties. But sending hundreds shuttles with different equipment to space and test which holds for longer time or create different systems for some medical equipment on which life of patients depends and test which will not break is unthinkable. Devices on which life or health of people depends or which damage will be extremly expensive must be reliable and this reliability must be somehow proven.

A good way to prove reliability is to use computer simulations. Its advantages are safety (no crashed shuttles or dead patients), price (huttles again, car crash tests, etc.), speed (atom colision stretched to a minute or forrest growth pushed down to a minute) and possibility to observe things we cannot in real (crash of galaxies). The disadvantage of simulatin is, above all, the problem with model validity. Other disadvantages might be the price of model creation, requirements on performance or inaccuracy of numerical solution. All these factors are discussed in [13].

In case of advanced digital systems which are the topic of this thesis the Marcov models are the optimal simulation system for solving given problem. As described in [10] a Markov model is a stochastic system used to model randomly changing systems as a Moore machine [3], meaning the next state is dependant only on the present state, not on the past progress. As we need fully observable and autonomous system, the Markov chains, described in [8] are used.

Reliability simulation, computed and measured in this thesis, is based on [15] Jan Trávníček's diploma thesis who implemented the proof of concept for this simulator and discussed the advanced digital system simulation in general. My thesis contains brand-new implementation of simulator with slightly extended interface dicussing behaviour of the simulator in various conditions and the impact of particular components results reliability. Above all, the reliability of fault checkers in functional units.

Chapter 2

Reliability and it's models

In general, reliability is an attribute of any system describing probability of success in system's actions. It may be defined as `1 - probability of failure`. In common sense you can say that reliability means how long keeps the system working and/or how often is it (un)broken. However, this is not very accurate and … reliable.

2.1 Reliability

- In research, the term reliability means repeatability or consistency. A measure is considered reliable if it would give us the same result over and over again (assuming that what we are measuring isn't changing!) [16].
- General attribute of object based on ability to perform requested functions with values preserved in defined technical conditions [1].

It is crutial to define what reliability really is and what values we need to observe before we start simulating and measuring.

2.1.1 Basic terms

By [5], the incorrect states of system are `fault`, `failure` and `error`. `Fault` means any system malfunction causing the system to perform in an unintended or unanticipated manner. This foul behaviour might be treated in the system (ie. with redundancy of components) so nothing will happen. In other case may this behaviour be innoticeeable or, with specific conditions met, cause `failure`. `Failure` is the inability of system to perform required functions within specified performance requirements. Such behaviour may lead to an `error` which is a discrepancy between expected and real value, state or behaviour of system.

In simulation we will consider two states of system `failure-free` where system is running as supposed to and `failing` where failure occured and the results are incorrect. All failures within this model are permanent and last until broken component is repaired. Systems with possibility to repair their components are called `renewable` which will be the most of systems simulated. Systems without ability to repair their components are `unrenewable` [6].

2.1.2 Reliability indicators

Basical indicator of reliability is the probability of failure-free run `R(t)`. Complementary value of probability of failure-free run is the probability of systems failure which defines the interval between system launch and any first occured failure. Relation between these two values can be defined as

$$R(t) = 1 - Q(t) \tag{2.1}$$

As described in [7] the two fundamental indicators of reliability are `Mean Time Between Failures - MTBF` and `Mean Time To Repair - MTTR`. With constant failure rate λ defined as the amount of failures per hour can `MTBF` be described as

$$T_f = \frac{1}{\lambda} \tag{2.2}$$

Analogically with μ defined as the amount of repairs per hour `MTTR` is defined as

$$T_r = \frac{1}{\mu} \tag{2.3}$$

2.1.3 Ways to achieve system's reliability

Apparently it's required to achieve the highest reliability possible with, of course, the lowest price possible. The straight way of increasing the reliability is for each component to lower λ - the noumber of failures in time. By lowering λ all other attributes are improved but the price of this process might be very high and there will always be some possibility of failure. This process is called `Fault avoidance`. It means using techniques and procedures which aim to avoid the introduction of faults during any phase of the safety lifecycle of the safety-related system [5].

Other way to manage foul behaviour is to expect faults and handle them. System with redundat `m` of `n` components (where `m` components from `n` must work) is able to hide up to $n - m$ component failures and work correctly. This is called `Fault tolerance` - the ability of a unit to continue to perform a required function in the presence of faults [5]. Other abilities of fault-tolerant systems are fault detections and fault recovery.

2.2 Markov models

With need of fully observable and autonomous system the Markov process(/chain) as the specific model must be used. As described in [2] a Markov process is a stochastic process whose behavior depends only upon the current state of the system. The particular sequence of steps by which the system entered the current state is irrelevant to it's future behavior. Markov state-space models have four main categories:

- Discrete space and discrete time
- Discrete space and continuous time
- Continuous space and discrete time
- Continuous space and continuous time

With two possible states of system - working or broken and time steps defined by real numers the second category - discrete space and continuous time will be used. A model is defined by a set of states and a vector of transitions for each state which is optimal to represent as matrix of transition probabilities. A vector of transitions consists of probabilities for transition from given state to any other. With set of states it is necessary to define which states are broken and which are correct.

Chapter 3

Model implementation

To measure the impact of reliability of checking components on the complete system's reliability we need a simulator to run a Markov model. Such simulator is a part of this thesis and can be found on attached CD. The implementation is based on [15] preserving backward compatibility on program's interface and keeping pseudo-codes of both simulation methods and arcitecture of a program on high level of abstraction. Simulations were executed on pc with Intel Core i5-3470 CPU @ 3.20GHz, 16 GB RAM under x64 Win 8.1.

3.1 Used technologies

With need of minimalistic command line interface, high performance, sophisticated mathematical methods and multi-platform support `Python` [11] as programming language was the choice, specifically `Python 2.7.x`. Application is written by `OOP` paradigm and for more complex mathematical operations uses the `NumPy` library [9]. For model definition the Extensible Markup Language - `XML` [12] is used. This choice was done on the former design and was preserved to keep the backward compatibility with existing models.

With multiple data on output of simulator, mostly sets of noumbers, their visualisation is required. First type of data is the dot graph drawed by `Graphviz` [4]. Dot graph represents states and transitions of matrix of transition probabilities described in 3.3. Second data type on the output is the simulation result. This is a set of pairs giving the time and reliability. To plot these values Gnuplot was used.

3.2 Definition of input xml file

As mentioned in 3.5.1, the first parameter of application is path to xml input file. This file is expected to be in standatd xml [12] format and must contain data defined in this section. Namely elements `architecture`, `correctness`, `repairRules`, `imunity` and `computation`. As requierd by `xml` standard, on first level contains the input file single `root` element which contains elements mentioned above as subelements. Any other data than these five sections and their content are ignored. Data format is based on Jan Trávníček's diploma thesis [?] which is based on work from CSE'2012 conference [14]. The accuracy of simulation results is strightly dependant on values of all components. It is, however, not easy and sometimes impossible to gain them as mentioned in [14].

3.2.1 Architecture

This section as it's name suggests describes components of model. Element `architecture` contains set of component-types each defined in `component` element. This element has two attributes, `name` and `count`. The `name` attribute defines type of modelated component and the `count` attribute gives amount of these components in system. It is convenient to give names so that each component begins with a unique letter. This has no impact on functionality of simulator. `Graphviz` uses first letters of components to build name for state in graph of transition probabilities (state name consists of first letters of names of components and count of working components of given type in state).

In comparisom with the original structure each `component` element has three subelements - `lambda`, `mu` and `rel`. All three elements contain single decimal value (given in decimal or exponential form). `Lambda` value is as previously defined λ, constant failure rate of given component meaning noumber of failures per hour. `Mu` value is as previously defined μ constant repair rate of given component meaning noumber of repairs per hour possible. Last, new, value `rel` defines reliability of checkers inside unit. Default `rel` value expected is 1 when lowered for any component it means that checker may fail to find out that this component is broken by given probability. Probability of correct behaviour of whole system is always lowered by lowered value of checker's reliability.

Complete `architecture` definition can be seen in 3.1. Model shown defines simple NMR system with five redundant functional units `FU`, single multiplexor `MUX` (used to aggregate results of functional units) and single repairing unit `GPDRC`.

Algorithm 3.1: Example of architecture definition

```
<architecture>
    <component name="FU" count="5">
        <lambda> 3.358e-6 </lambda>
        <mu> 9.977e-5 </mu>
        <rel> 1 </rel>
    </component>
    <component name="MUX" count="1">
        <lambda> 0.00000015 </lambda>
        <mu> 0.0000999 </mu>
        <rel> 1 </rel>
    </component>
    <component name="GPDRC" count="1">
        <lambda> 7.388e-7 </lambda>
        <mu> 9.9e-5 </mu>
        <rel> 0.8 </rel>
    </component>
</architecture>
```

3.2.2 Correctness

Section `correctness` defines set of states considered correct by the definition of the model. This means that a state is correct when sufficient amount of components works correctly. Element `correctness` contains set of subelements `correct` where each of these subelemets defines single minimal correct state. Minimal means, for example, that when `correct` defines three of five functional units are needed for system to work correctly the simulator takes all five, four and three working units as correct state. This feature is implemented to save work with creating model so it's not necessary to write down all states explicitly (it is assumed that the system cannot be broken with more components correctly running while running correctly with fewer correctly running components).

Each `correct` element contains a set of `component` subelements each defining the amount of components of given type required for that state to be correct. `Component` element contains two attributes - `name` and `count`. The `name` attribute gives a name of a component type previously defined in `architecture`, the `count` attribute gives the minimal amount of components needed for that state to be correct as described above. Values recieved from `correctness` section are checked by comparism to `architecture` - states defined as correct must exist in model's architecture (it is not possibe to define components that does not exist or higher amount of components as correct then the model contains).

On example 3.2 is shown a possible definition of `correctness` for previous example of NMR system. System is correct when multiplexor `MUX` and three or more functional units `FU` are forrect. It would be possible to write down other two states with four and five correct `FU`'s but as explained above these records are redundant and would make no difference on the model definition.

Algorithm 3.2: Example of correctness definition

```
<correctness>
    <correct>
        <component name="MUX" count="1"/>
        <component name="FU" count="3"/>
    </correct>
</correctness>
```

3.2.3 Repair rules

This section defines rules by which may the simulator generate repairing transitions. Xml element for repair rules-section is named`repairRules`. It's content consists of a set of `component` elements and a single `priorRules` element. Each `component` element contains two attributes - `name` and `count`. The `name` attribute defines component-type from architecture needed for system to make repairs and the `count` attribute defines the amount of these component's needed to work. With no `component` given the system needs no exact component to repair other components. This can mean that, for example, repairing component is outside of the modelated system. On the other hand with `component count` higher than `count` defined in `architecture` there will never be enough repairing components working and no repairs will be done. Both these extremes are correct.

The `priorRules` element contains non-empty set of rules (always at least one) by which a single component may be picked to be repared in each state of system. With fully defined rules there is always a deterministic way to find the component to be repaired. On the other hand in case when more components have for given state the same priority one of them will be picked by pseudo-random decision. Important is that at any time none or one component may be repared (or broken - up to sigle action in one time point). Single rule is defined by `priorityRule` element with attribute `value`. This `value` is an integer giving priority of rule where rules with higher priority will be considered first. When more rules have same `value` of priority their order is undefined and a pseudo-random decision is used to pick one of them first. It is appropriate that the rule with lowest priority has unique `value` and no required component defined because this rule is used as default rule with no needed componets running in situations when no other suitable rule can be found. `PriorityRule` element contains two sets of subelements. The first set consists of `component` elements and the second of `priorityVal` elements.

Element `component` contains two attributes - `name` and `count`. The `name` attribute gives the name components type the `count` attribute the exact amount of given component-type units needed to be correctly working for applying rule that is being processed. This means that a rule is picked when all components mentioned have exactly that amount of working units as defined in the rule. The second element type `priorityVal` is used to define priority of components within given rule. `PriorityVal` has single attribute - `name` which defines name of components type. Within `priorityVal` is a single integer number defining priority for given component. This priority defines order in which components will be picked for repair. First suitable component found will be picked. Non-unique component-priority values are processed the same way as non-unique rule-priority values this means that from components with the same priority one will be picked pseudo-randomly. Naturally the `name` attribute is checked for being defined in `architecture` in both `component` and `priorityVal`.

It is not possible to gain more components then defined in `architecture` by repair. Rule defining such action is correct and will be processed, however never picked so it is pointless to set such rule. Apropriate definition of repairing rules is crutial for system's reliabilty and necessary for expected system's behaviour. For example 3.3 is shown a possible way to define reraire rules for previously defined system. When some `FU` is broken, one of the explicit rules is picked. The same behaviour would be achieved with defining rule for every state with any broken component or just the last basic rule. This would define priority of each component and say to repair anything broken in given order.

Algorithm 3.3: Example of repair rules definition

```
<repairRules>
    <component name="GPDRC" count="1"/>
    <priorRules>
        <priorityRule value="5">
            <component name="FU" count="1"/>
            <component name="MUX" count ="1"/>
            <priorityVal name="FU">1</priorityVal>
        </priorityRule>
        <priorityRule value="4">
            <component name="FU" count="2"/>
            <component name="MUX" count ="1"/>
            <priorityVal name="FU">1</priorityVal>
        </priorityRule>
        <priorityRule value="3">
            <component name="FU" count="3"/>
            <component name="MUX" count ="1"/>
            <priorityVal name="FU">1</priorityVal>
        </priorityRule>
        <priorityRule value="2">
            <component name="FU" count="4"/>
            <component name="MUX" count ="1"/>
            <priorityVal name="FU">1</priorityVal>
        </priorityRule>
        <priorityRule value="1">
            <priorityVal name="MUX">2</priorityVal>
            <priorityVal name="FU">1</priorityVal>
        </priorityRule>
    </priorRules>
</repairRules>
```

3.2.4 Imunity

Imunity section defines components that cannot be broken. This means that all components listed will always work correctly and system with all units imune will be abolutely reliable. Element `imunity` contains set of `component` subelements. `Component` element contains two attributes - `name` and `count`. The `name` attribute defines components-type name from `architecture` and the `count` attribute the amount of units of given type that will be unbreakable during simulation. For model with no imune components the section `imunity` is left empty.

For example 3.4 the definition of `imunity` is shown. Namely unbreakable repairing unit `GPDRC`.

Algorithm 3.4: Example of imunity definition

```
<imunity>
    <component name="GPDRC" count="1"/>
</imunity>
```

3.2.5 Computation

This last section gives informations for simulation run. Element `computation` contains two subelements - `time` and `samples` both containing a single integer value. The value of `time` attribute defines simulation length by setting the number of time points computed. It is important to use the same time unit for computing λ and μ values for all components. In examle 3.5 the miliseconds are used. In next chapter will be shown the impact of various units and differences in simulations.

The second subelement of `computation` - `samples` - gives the amount of output values stored in the output file. The amount of `samples` must be lower than the length of simulation defined in `time`. It is impossible to output more values than the simulator counts. The length of simulation and the number of samples combined gives the length of single simulation step. The length of one step is equal to division of time by noumber of samples. For this time the simulator runs and counts and when it hits the end of step the overall probability of system's correctness is written down. For example 3.5 are fourty milion milliseconds given divided into houndred steps. With these settings simulator will count approximately eleven hours of simulation time with one step taking approximately 6.66 minutes.

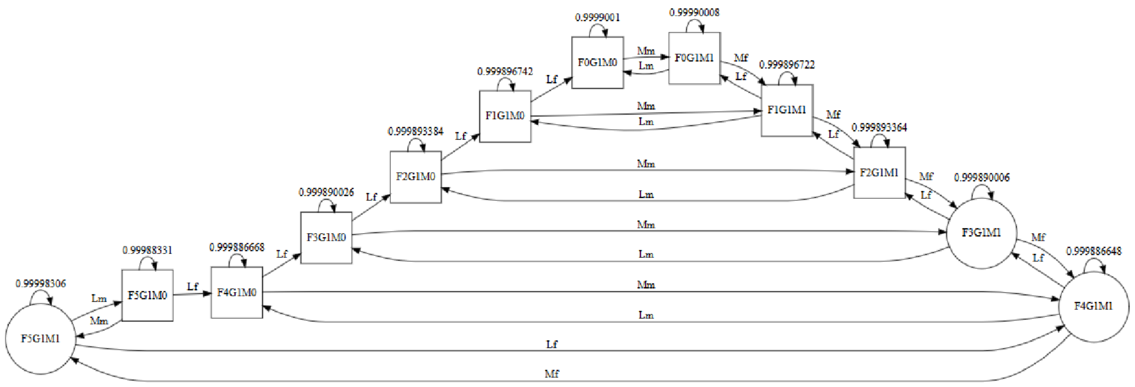
Algorithm 3.5: Example of computation definition

```
<computation>
    <time>40000000</time>
    <samples>100</samples>
</computation>
```

3.3 The matrix of transition probabilities

After reading and parsing all input data the matrix of transition probabilities must be created. This matrix contains probabilities of transitions to other states for each state and is used by both numerical and ssa algorithm. To generate such matrix the set of all states possible must exit.

Generating statespace is done with modified depth first search algorithm [17]. With complete architecture on stack as initial state the top state is popped and expanded to stack by decresing the amount of each unit per newly expanded state. To prevent redundancy in generated states when a new state is picked from the top of the stack a human-readable hash is generated and saved. When expandinding new states their hashes are compared to the set of existing hashes and new states, that are duplicit, are ignored. Processed state after giving all child states is appended to list which stands for the final statespace. This list of states is immutable and position of state is used as it's index. The first state contains no broken unit and all the other states are expanded form this one. Compared to standard depth first search algorithm we need to find all possible states so the algorithm is not stopped until there is any element on the stack.



Graph 3.1: Grapg of matrix of transaction probabilities for model defined in 3.2

With statespace as a vector of states the matrix of transition probabilities is a cartesian product computed as this vector squared. Than for every row representing a state all possible transitions are generated. First is tried to break each component and if this action leads to a legal state the probability of transition 3.1 is written into a row on position of generated state. After generating all breaking-transitions single transition for repair is chosen if given state is repairable. For such state the repairing rules are walked through in order of their priority and by the first suitable a component is chosen to be repaired. This is done by writing μ value of component being repaired on row's index of state with repaired component.

$$trans\_probability = \lambda_{component\_broken} * num\_of\_working\_comps\_before\_breaking \tag{3.1}$$

The size of matrix is equal to the square of statespace state which grows exponentially with the amount of components with most cells equal to zero (because there is a transaction only to adjectant states). With such matrix it becomes hard to work after having more than a few states. To provide a way to read the matrix more easily it may be shown as a graph of states and transactions between them. Such graph contains the amount of states equal or less the the statespace (units imune that wont break will preserve the generation of doubled amout of states). The graph of transaction probabilities for model of NMR system defined above can be seen in 3.1. A state is defined by first letters of names of component type and amount of these components working. A state where the system is working correctly is dwawn as circle on the other hand the state where system is broken is drawn as square. Arrows between states show transitions with probability of given transition above.

Language used to describe transition graph is `dot` [4]. After computing the matrix of transaction probabilities a .dot file describing this is automatically generated and stored as path_of_output.dot next to simulation output file. Before the simulation is started the simulator calls system's utility to translate `dot` to `pdf`. If `dot` command is not present or cannot be executed, warning is raised and simulator continues to begin the simulation.

3.4 Simulation methods

For simulation computation there are two implemented methods. In theory, both of them give the same results what, however requiers correct model setting. Both methods, their advantages and disadvateges, will be described in this section.

3.4.1 Numeric

The numeric method for solving Markov model is based on periodical matrix multiplication. For every time point the matrix of transition probabilities is multiplied with row vector. Each value in vector gives probability for model to be found in that state. Sum of all values in vector meaning sum of all transition probabilities is always equal to one. Sum of probabilities of all states defined as correct equals the probability of system's failure-free run as defined in 2.1.2. This value is the desired result of simulation.

Before computation the vector must be initializet to single state by setting the probability of model to be found in this state to one. By default the simulation begins in fully working state but this may be changed. After this initialization the simulation starts looping multiplying actual vector with matrix in every step. The result of such multiplication is a new vector for time $t + 1$, used in next iteration. When sample time is reached the probability of system's failure-free run is counted and written into results.

3.4.2 Stochastic simulation algorithm

The basical principle of ssa is to literally simulate behaviour of given system. This is done by randomly switching to next state after random time both by probabilities given by the matrix of transition probabilities. Single run of simulation is single experiment with Markov chain with continuous time. Result of such experiment is information that system is in given time whether working correctly or not. This information is recorded in the end of each simulation step as well as in the numeric method. To gain requested probability it is necessary to repeat this cumputation. After sufficient amount of repetitions the final probability of system's failure-free run per time is counted as arithmetical mean of values for given time.

It is a little bit tricky to obtain the sufficient value of repetition. With the number of repetitions `R` is the relative error `E` of simulation described in 3.2. On this relation can be seen that to increase the result accuracy by ten time it is necessary to repeat the simulation hundred more times [13].

$$E = \frac{1}{\sqrt{R}} \tag{3.2}$$

3.5 Simulation process

With all fundamental theoretical parts of simulation described it is time to describe the complete run of simulation script in more practical way.

3.5.1 Simulator usage

As mentioned before the application is used via command line. The file to execute is main.py which calls all other modules. Expected parameters are: path to input xml file (described earlier), path of output file to be generated and the name of simulation method (numeric, ssa), for ssa method the number of iterations as required parameter and number of paralel processes as optional parametr (default is one and paralelism is not implemented yet but interface is prepared fot thes option).

Simulation output is simple text file with values showing probability of failure-free run of system in time. Amount of values is given in input xml by samples. Format of a sample (row) is one integer number showing simulation time and decimal number showing probability of systems's failure free run for given time. These two numbers are separated with space and ended with new-line. Second possible output is a graph showing transitions between states representing matrix of transition probabilities. This graph, however, is generated for small-enought amount of states only. The limit of states for graph generation is set to 128.

3.5.2 Model preparation

After params parsing with `Params` class from `inout.py` module, the input data are loaded. All informations are obtained from XML file described above and then parsed into internal representation. This is done in class `Data` from `inpout.py` module which has methods to parse input XML and stores gained data in dicts and tuples as data-object's attributes.

The next step is matrix creation. This is done by making an instance of `Matrix` from module `matrix.py`. The object's initialization takes care of all necessary actions as state-space generation and matrix creation. In addition `matrix` object contains all supportive methods for any action related to matrix. With params read, data parsed and matrix prepared, the last step before simulation is to create `Simulator` object from `simulator.py` module. `Simulator` class wraps simulation computation which is started with method `run`.

3.5.3 Simulation run

When simulation is run `Simulator` picks chosen method and aggregates results recieved writing them into output file.

When Numeric method is chosen then, as described in 3.4.1, row vector set to initial state is created. The main simulation loop beginns cycling over simulation steps. Each step is a nested while loop cycling over each time point computing one matrix computation with `numpy.dot` method. After finishing nested loop, an actual probability of system's failure-free run is computed and saved as sample into result dict.

Computation of `ssa` method is significantlymore complicated. At first the lambda-vector is created. By original algorithm are these values supposed to be computated during simulation. With optimalization described in [15] these values are pre-computed. Whole simulation is being repeated in wrapping for cycle that many time set in input parameter. The simulation itself runs in a while cycle to the end of simulation time. Algorithm skips time points where nothing happens and resolves actions of component breaking and repair and records probability of system's failure-free run after time of one simulation step. To skip simulation of system which is definitly broken the simulator writes default zero value from such moment on.

Chapter 4

Simulation results

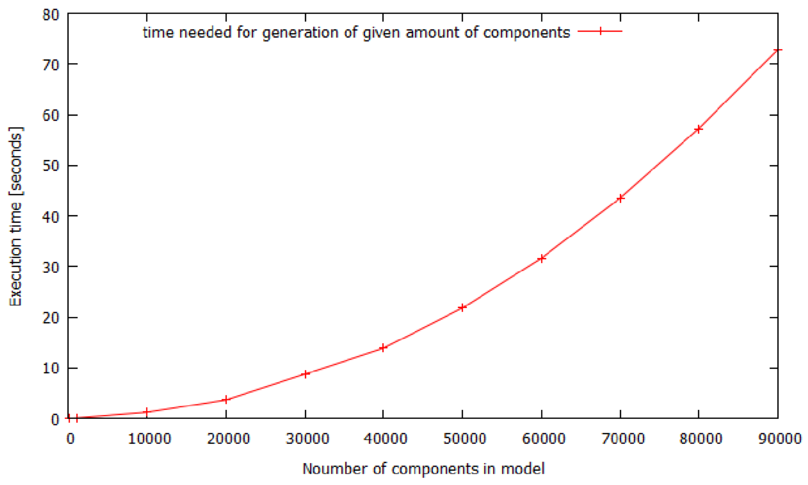
With model defined and working it is time to proceed to simulation itself and description of experiments done. Upcoming chapter is supposed to be the core of this thesis. First some statistic about program execution will be mentioned like the length of script run or impact simulation time on simulation methods. Another topic to be discussed is the difference between both simulation methods impact of model size and simulation length and accuracy of their results. After making all these variabilities clear it is possible to start experimenting with component's reliability. For all graphs showing reliability of system both simulation methods were used. The numerical results are always drawn with simple line, the ssa results with points.

4.1 Execution durations

The first, practical issue for any simulation is indeed the length of simulation run. To measure time within the script a standard python module `profile` was used. In simulation there are two significant parts that may take noticeable time and it is possible to measure them separately. Generation of statespace and matrix of transition probabilities, dependant on the amount of components, is the first one. The second is the simulation itself which is more tricky. Even not considering the fact of having two methods the simulation itself is dependant on size of the model and it's length which adds another dimension into results.

4.1.1 Matrix generation duration

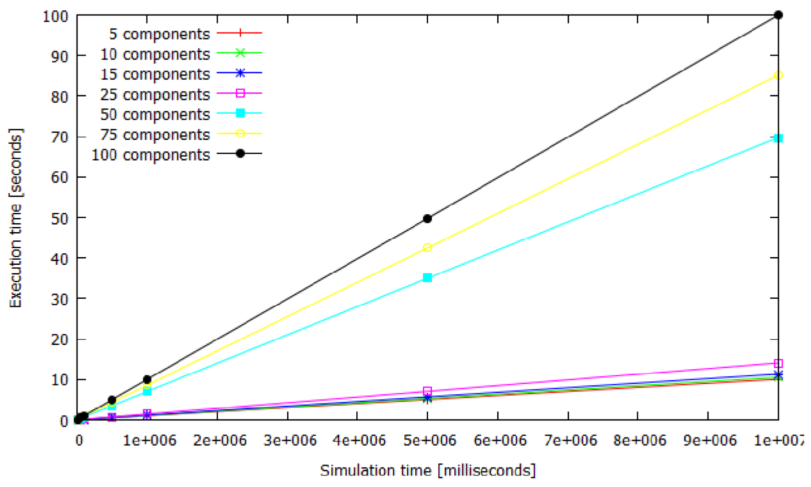
As mentioned above the execution time of matrix generation is directly dependant on noumber of components. In graph 4.1 is shown the dependancy of time in seconds on noumber of components. The time needed for matrix generation is rising exponetialy, however the absolute value of time grows to one minute for biggest models computable on machine mentioned in 3. The capabilities of machine allow the model to limitly grow to one hundred thousand states. That is multiply more than the simulator is able to compute. The time of practically used models is between fractions of second and single seconds and so it is not necessary to observe this anymore.



Graph 4.1: Matrix generation time

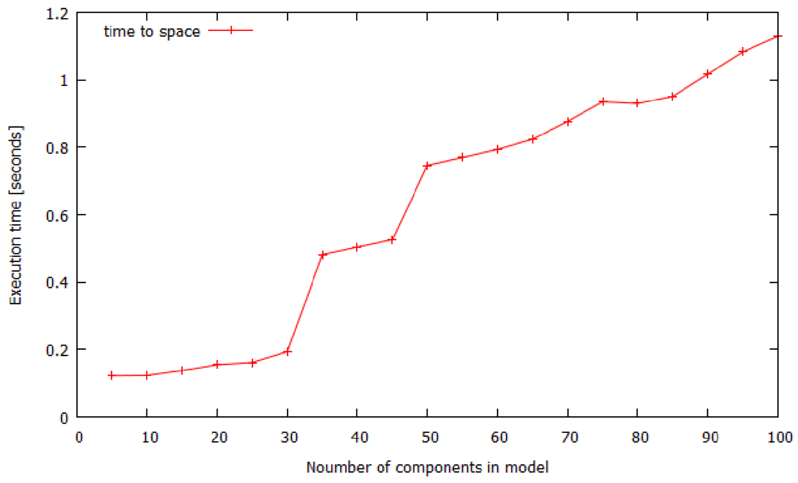
4.1.2 Impact of model size and simulation length

It is obvious how both model size and simulation length influence the length of execution of simulation. The bigger model we use the longer it takes to compute one step and the more steps have to be computed the more time it takes. The unpredictible and interesting information is the real execution time. Model used for this experiment is the one defined above in 3.2 with `FU` units making the tested amount of units and correctnes moved to be greater than the half of the amount `FU` units. Used simulation time unit is milliseconds. The progress of execution duration for numerical method can be seen on the graph 4.2. One line indicates one model and it's execution duration in time. The size of the model is mentioned in the key of the graph.



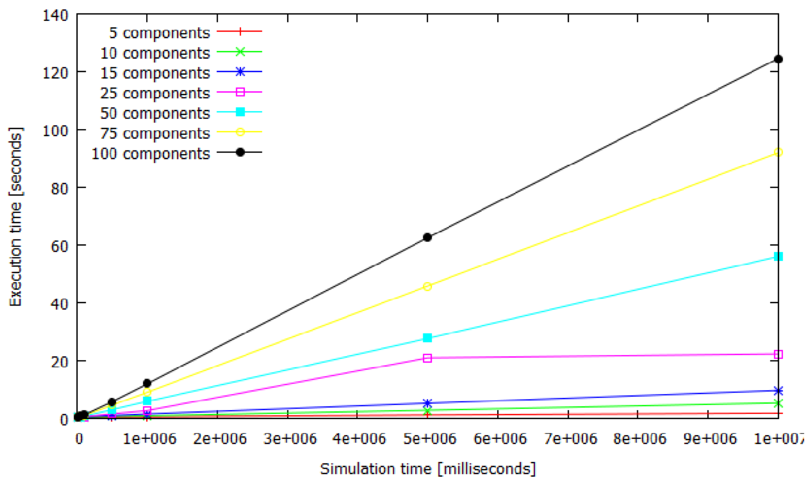
Graph 4.2: Numeric simulation times

As expected, with growing simulation time, the execution duration grows lineary, each time unit means one multiplication of matrix and nothing else happens. What is more interesting is the inconherent growth of execution time with regard to the size of used model. The time of execution is partially linear. A more detailed graph is shown on 4.3. Here was the time measured again for single simulation length, namely hundred thousand milliseconds, and more different models in sizes from five to hundret components. This behaviour is most probably caused by numpy's attitude to computers memory or computation optimalization moving in levels of matrix size. To prove the source of this behaviour more research would be needed but since it has no impact on simulation results it will stay only mentioned.



Graph 4.3: Detail of simulation time dependance on size of statespace

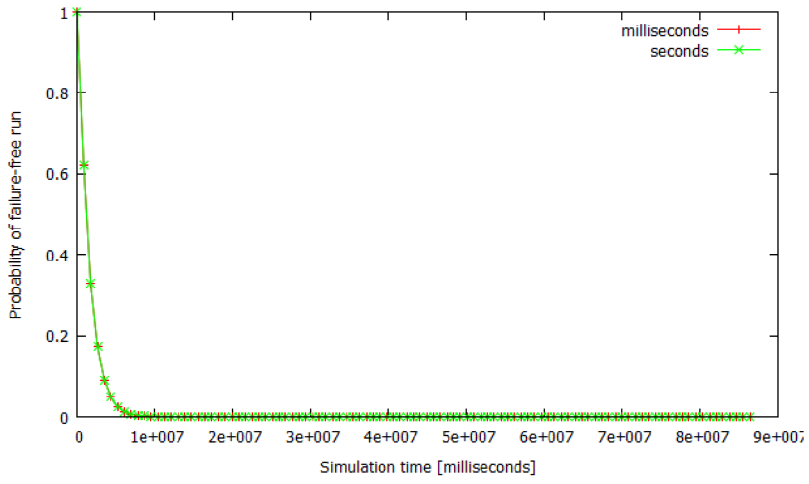
As the stochastic simulation algorithm uses no sophisticated computation methods with regard to the size of model an even growth of execution duration can be seen. However with more dynamic simulation computation is the growth not that stable in time. This stability may be dependat even on simulation result and is discuddes in 4.1.4.



Graph 4.4: Ssa simulation times

4.1.3 Impact of various time units on simulation

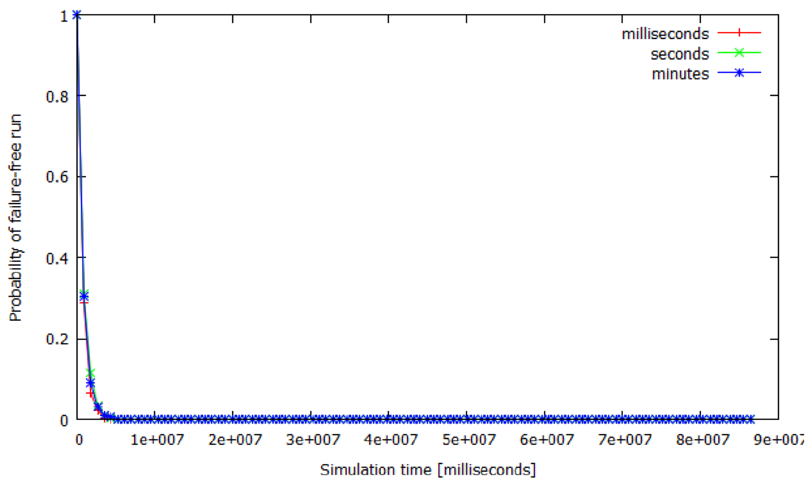
While modeling systems that operate in units of millisecond it is expected to keep this resolution and simulate system with that precision. On the other hand it is required to experiment wint models running for years or decades which is computationaly very exacting. Considering the definition of λ and μ 2.1.2 as number of failures and repairs in one hour it might be possible to simulate the system in lower resolution with the same results and faster simulation execution. Experiments with various time units use model defined above without imunity for repairing `GPDRC` unot and with simulation time of one day.



Graph 4.5: Simulations with different time units for numeric method

As can be seen on graph 4.5, results for numerical method seem to be perfectly equal. The difference between both results before the graph gets steady is about one and half hundreth of percent. Considering possible inacurracy caused by model definition and other factors influenting result such difference is unimportant. Result of third experiment with minutes is not shown because with current model the numeric method fails to compute with too high values of λ and μ. For another enlargement of simulator time units redefinition of λ and μ would be required.

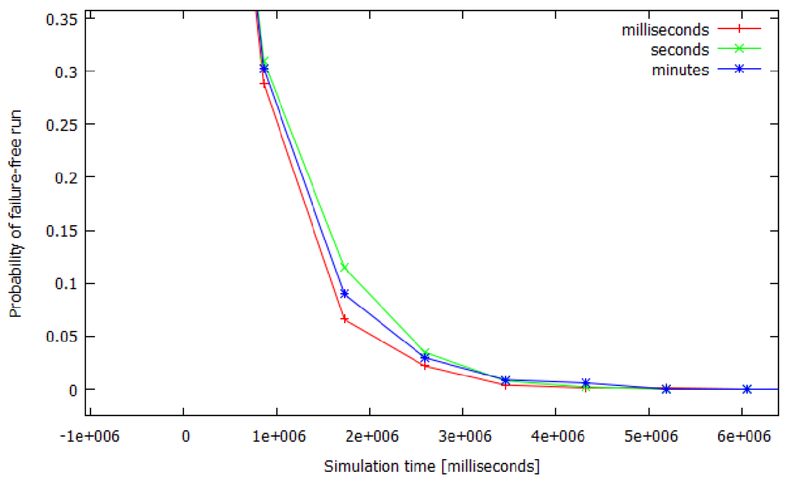
This is however not necessary. As shown in 4.1.2 the execution time of numeric method grows lineary with simulation length. This was confirmed again for the execution of model counting by milliseconds ran for nearly five minutes model counting by seconds ran for 0.36 sec. and model counting by minutes (nonsense result has no inpact on execution duration) ran for 0.054 sec.



Graph 4.6: Simulations with different time units for ssa method

More interesting results came from experiment 4.6 with ssa method. The ssa simulation results, alike the numeric, differ a little but still inconsiderably. The detail of difference is shown on graph 4.7. From the nature of ssa method the results differ more than at numeric computation but still the highest difference nears 5%.

As the ssa algorithm resolves just actions im model and skips time where nothing happens the execution time of all the models was similar. Namely 1.32 sec. for model counting by milliseconds, 1.35 sec. for model counting by seconds and 1.32 sec. for model counting by minutes. The size of simulation time unit has no effect on the execution duration for ssa method.

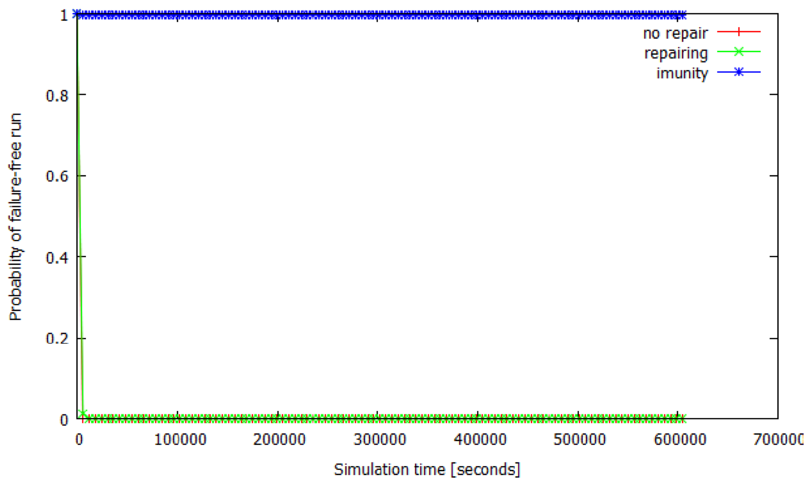


Graph 4.7: Detail of simulations with different time units for ssa method

With inconsiderable influence on results distinct improvement of numeric simulation and no effect on ssa the second may be declared as optimal time unit for experiments with greater simulation time. This modification is necesarry even for ssa to keep the same models for both methods.

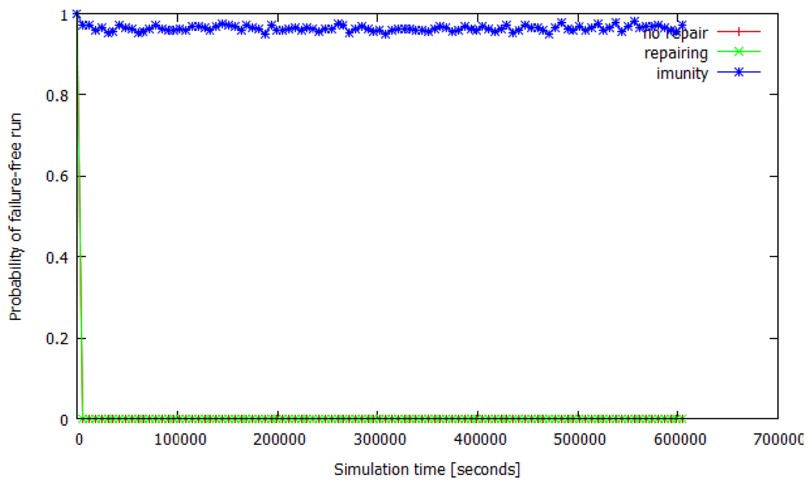
4.1.4 Impact of model behaviour on simulation length

The last remarkable aspect of executions is the relation between simulation and simulator behaviour. It would be expectable that events within simulation will have no effect on it's execution. This is true for the numeric method. On graph 4.8 is shown progress of reliability for model without repairing unit, a model with repairing unit that can be broken and model with imune repairing unit. All three model ran for about 2.5 sec. with difference of tenths of seconds.



Graph 4.8: Long term simulation with different repairs by numeric method

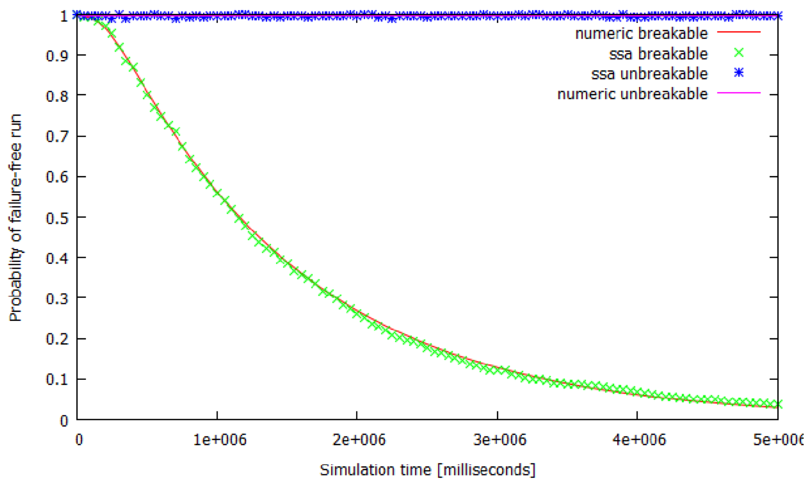
The simulations with ssa method shown on graph 4.9 with approximately same results has extremely variable execution times. The first two models both breaking quickly have execution time about one second. Model without repairs finished in 0.8 sec., model with breakable repairs in 1.25 sec. In opposite of fast execution for models which break stand simulation of model of system that keeps working like model with imune repairing unit. Execution of this model took 440 sec.



Graph 4.9: Detail of simulations with different time units for ssa method

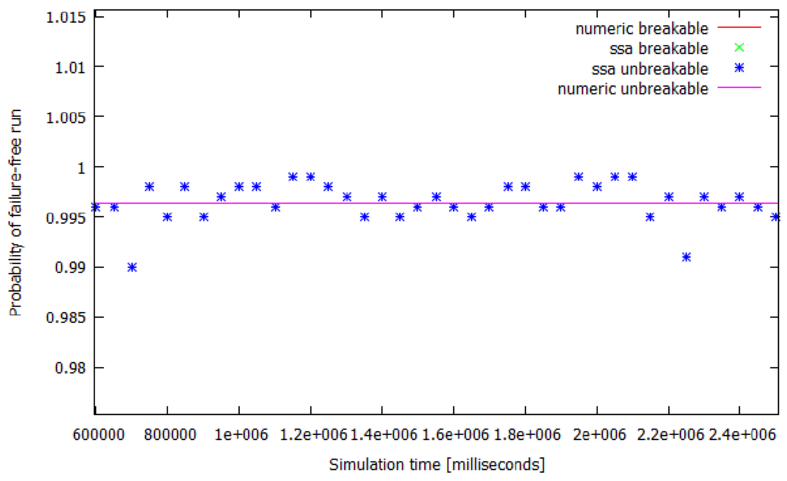
4.2 Difference between numerical and ssa methods

As each method uses different ways to compute results of simulation it is expectabe to gain not exactly equal results from both methods. Graph 4.10 shows results of both methods on different models. The unbreakable results are from model defined above, the breakable use the same model but with imunity section left empty so the repairing units will stop working sooner or later.



Graph 4.10: Comparism of results of simulation methods

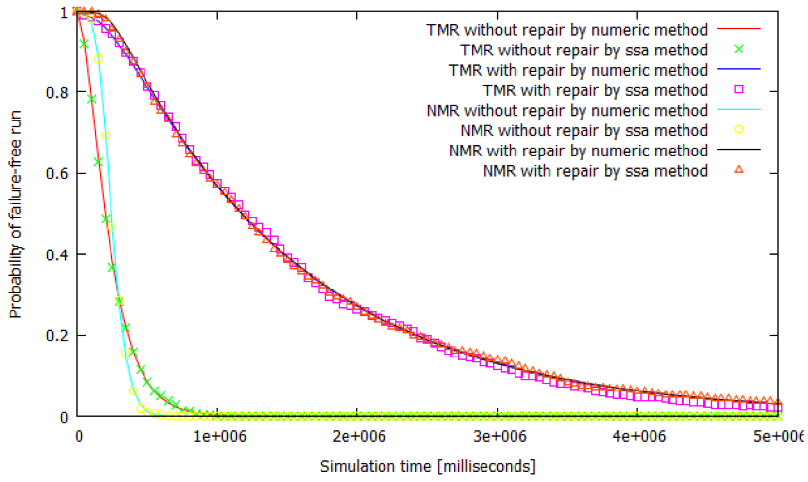
As can be seen, ssa results approximate numeric results with no big difference. After thousand repetitions of model simulation with ssa method is the deflection from numerical results up to one tenth of percent and ssa results oscilate around numerical. This behaviour is shown on detail 4.11 from graph 4.10. As the result of ssa method is an aritmetical mean 3.4.2 its graph will always be fuzzy. With sufficient amount of repetitions becomes this volatility inconsiderable.



Graph 4.11: Detail of comparism between simulation methods

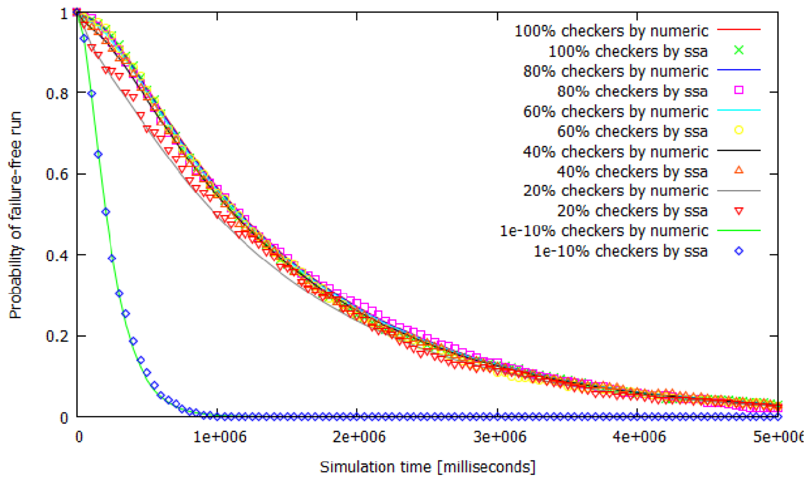
4.3 Influence of component reliability

With knowladge of simulator's behaviour it is time to discuss the results of experiments with various reliability level for different systems. Models used in upcomming set of experiments are based on NMR model with five functional units defined above 3.2. First system to test is `TMR` - Triple Modular Redundancy (a `NMR` with tree redundant functional units) then a fifteen unit `NMR` - N-Moduler Redundancy. Goal of these experiments is to show what system's redundancy is need for which reliability and if the FU's unit redundancy is enought to keep the system working.



Graph 4.12: Comparism of endurance of systems with growing redundancy

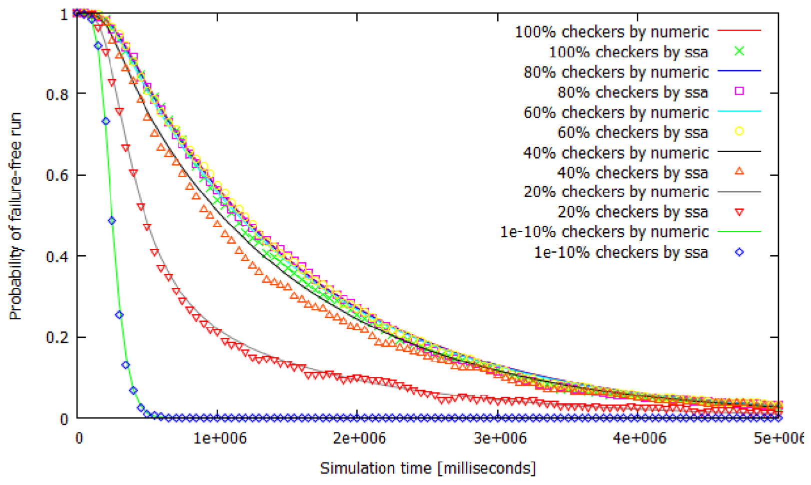
On graph 4.12 are shown the results of both `TMR` and `NMR` model ran without repair and than with it. It can be seen that with full reliability of checkers are results of both models for equivalent situations similar. Regardless of repair the `NMR` model keeps giving correct results for a slightly longer time but than break never the less. Let us see what happens after adding the aspect of functional unit's checkers reliability to these systems for now behaving similarly.



Graph 4.13: Impact of FU's checkers reliability on TMR system

As first, the `TMR` model was tested. This experiment consists of repeated simulations by both numeric and ssa method constantly decreasing the functional unit's checkers reliability. The value of reliability began on 100% and decreased by 20% for each simulation run. The last value was not 0% as this is forbidden by simulators definition but was limity converging to zero, namely 1e-10%. Results of these ten experiments can be seen on graph 4.13.

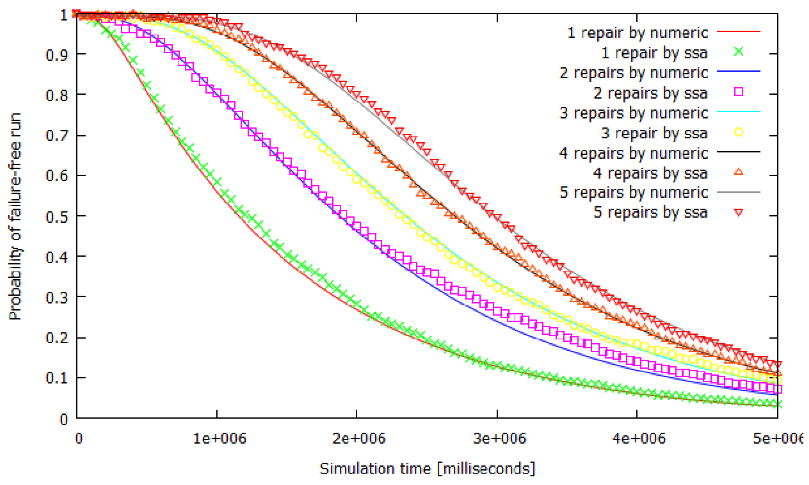
Until there is any reliability of checkers left the impact is almost inconsiderable. With repairing unit unbroken the lower checker's reliability is constantly slightly decresing the probability of system's failure-free run. This constant lowering can be seen on absolutaly unreliable checkers where the repairing unit almost stops repairing - not knowing the funcional units are broken.



Graph 4.14: Impact of FU's checkers reliability on 15-unit NMR system

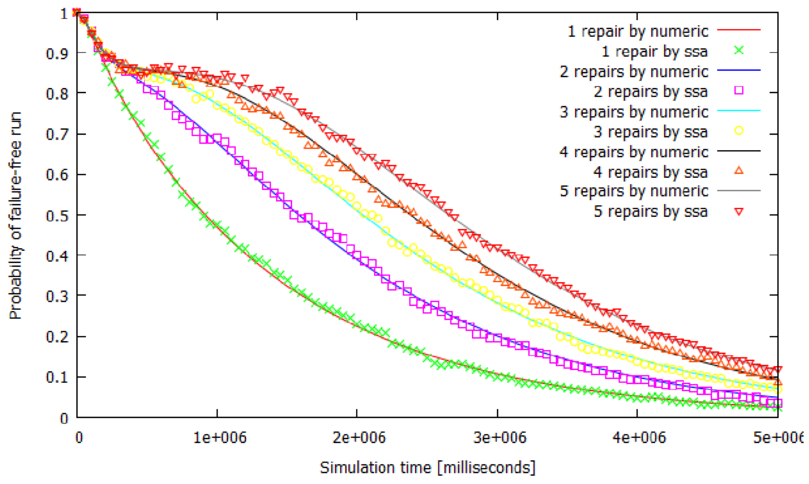
The same set of experiments was done for the `NMR` model, results of this second set of experiments is shown on graph 4.14. Progress of behaviour of both `TMR` and `NMR` is relatively similar but few crutial diferences can be seen. The most distinctive difference is the continuous decresing of system's probability of failure-free run with the decreasing of checkers reliability. This is caused by repairing unit's unability to manage to repair such amount of broken units. Based on this experiment we could say that - from long-term view - the higher functional unit's redundancy in system with unreliable checkers means deterioration of system's probablity of failure free run. On the other hand, another important difference can be seen. In the beginning of simulation time the `TMR` system drops is't probability before or gets steady for a while what does not happen for the `NMR` system. From this short-term view, the higher functional unit's redundancy prevents the system's probability drop by keeping enough units working. This difference is more clearly shown on graphs 4.15 and 4.16 and will be discuseed later more in detail.

As can be seen on graphs 4.13 and 4.14 increasing the redundancy of functional units increases the probability of system's failure-free run in short-term view a little but with any amount of these units sooner or later the system will inevitably break. This point of breaking may be the time when the repairing unit itself breaks and the system works just for the time until enough components break. Another set of experiments was done to measure the impact of repairing units amount. System chosen was NMR with five functional units and one to five repairing units. Results measured in this second set of experiments are shown in graphs 4.15 and 4.16.



Graph 4.15: Impact of repairing units amount on system with high FU's checkers reliability

In graph 4.15 can be seen the set of results from exteprimments with functional unit's checkers reliability equal to 80%. The short-term probability of system's failure-free run is kept high by sufficient redundancy of funcional units which is able to disguise faul results from units that are broken. With adding repairing units the time before the system starts to break increases logaritmically - with more repairing units the time rise by a single new repairing unit is lowering. Even with decreased reliability of checkers is the system with redundancy of repairing units able to keep working. The same can be seen on graph 4.16. The level of relevancy of functional units reliability was set to 20% which is extremely low.



Graph 4.16: Impact of repairing units amount on system with low FU's checkers reliability

However, it is obvious that lowering reliability of functional unit's checkers constantly decreases the the reliability of system as mentioned above. While not knowing that functional units are broken the repairing units will not repair them as frequently as necesary and so the system will return errors more often. In similar time as the system with more reliable checkers starts the system with unreliable checkers to break. From these facts it is obvious that reliability of checkers has no impact on system's long-term probability of failure-free run, on the other hand the redundancy of funcional units is necesary for system to work correcly in a short period of time. Set of redundant units is able to give correct result with some units broken as defined in 2.1.3. For a long period of time it is, however, necessary to ensure the repair of units. For a breakable repairing unit this is again possible with redundancy. The amount of repairing units needed is dependant on demanded possibility that at least one repairing unit will work in time. The value can be counted from repairing component's lambda value 2.1.2 and the amount of repairing units as

$$R_{rep} = 1 - (\lambda * amount\_of\_repairing\_units) \tag{4.1}$$

Chapter 5

Conclusion

After inheritng a work with some progress the biggest challange was to understand what and how it does. First step was the math behind simulator, namely the theory of reliability and the ways to compute and simulate it as Markov models. Equiped with this knowledge the second challenge was to understand simulator application from [15]. This application was supposed to be extended and used for experiments but as it was mainly a proof of concept for simulation methods it was appropriate to implement it from scratch.

The brand new verion is writen by object oriented paradigm and keeps only the the abstract concept (read input, make matrix, run simulation) and simulation algorithms which are optimalized and their implementation is not a part of this thesis. For requested experiments few extensions were done above all the possibility to define reliability of fault-checkers for each unit type. Other modifications are for example the change of internal data storage or lambda-transition value fix (the old version did not took in consideration that posibility of breaking single component of set is it's lambda multiplied by the amount of components in this set).

With this application as minimal implementation needed for this thesis there are many possibilities to improve it. From the technical point of view the ssa simulation may be coumputed in more paralel thread as the computation is repeated for hundred or thousand times and the results are agregated. From the conceptual product point of view the possibility of starting in somehow broken state of system might be interesting.

Possibilities of usage of the simulater are unlimited and other researches may be done. The results of research made in this thesis will be used for a paper.

References

[1] *ČSN 010102*. 1993.

[2] Butler, R. W.; Johnson, S. C.: *Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach.* Langley Research Center - Hampton, Virginia, 1995.

[3] Cohen, D. I. A.: *Introduction to Computer Theory.* Prentice-Hall, 1997, iSBN 978-0-471-13772-6.

[4] Graphviz developers: Graphviz [online]. `http://www.graphviz.org/`, [cit. 2015-05-18].

[5] Harzati, V.: Difference between Fault, Failure and Error, [online]. `https://vikashazrati.wordpress.com/2008/10/30/fault-failure-error/`, [cit. 2015-05-07].

[6] Hlavička, J.; Racek, S.; Golan, P.; aj.: *Číslicové systémy odolné proti poruchám.* Praha: Vydavatelství ČVUT, 1992, iSBN 80-01-00852-5.

[7] Kopetz, H.: *Design Principles for Disributed Embadded Applications.* Springer, 2011, iSBN 978-1-4419-8236-0.

[8] Meyn, S.; Tweedie, R. L.: *Markov chains and stochastic stability.* Cambridge: Cambridge university press, 2009, iSBN 978-0-521-73182-9.

[9] Numpy developers: NumPy [online]. `http://www.numpy.org/`, 2008 [cit. 2015-05-03].

[10] Pukite, P.; Pukite, J.: *Modeling for reliability analysis: Markov modeling for reliability, maintainability, safety and supportability analysis of complex systems.* New York: IEEE Press, 1998, iSBN 0-7803-3482-5.

[11] Python community: Python [online]. `http://www.python.org/`, [cit. 2015-05-03].

[12] Quin, L.: Extensible Markup Language (XML), [online]. `http://www.w3.org/XML/`, [cit. 2015-05-04].

[13] Rábová, Z.; Janoušek, V.; Peringer, P.; aj.: *Modelování a simulace.* Brno: VUT, 1992, iSBN 80-214-0480-9.

[14] Straka, M.; Kaštil, J.; Kotásek, Z.: *Methodology for Reliability Analysis of FPGA-based Fault Tolerant Systems. In CSE'2012 International Scientific Conference on Computer Science and Engineering.* The University of Technology Košice, 2012, 146–153 s., iSBN 978-80-8143-049-7.

[15] Travnicek, J.: *Tvorba spolehlivostních modelů pro pokročilé číslicové systémy*. FIT VUT v Brně, 2013.

[16] Trochim, W. M.: Theory of Reliability [online]. `http://www.socialresearchmethods.net/kb/reliablt.php`, 2006-10-20 [cit. 2015-05-02].

[17] Zbořil, F.; Zbořil, F.: *Základy umělé inteligence*. Brno: VUT, 2012.